1 INTRODUCTION

The language Prolog is doubly cursed. It is considered to be flawed by many but for differing reasons. Its failure to reach the declarative ideals of logic programming [Llo84] make many consider it flawed in conception. The logic programming community has devoted considerable effort to developing new languages which better meet these ideals. Despite this shortcoming, many have found Prolog the language of choice for a variety of application areas including expert systems, reasoning, database interfaces and natural language understanding. However, many also consider Prolog practically flawed because they perceive it as inherently inefficient. It is seen as only suitable for uses where execution speed is less important, such as prototyping and experimental applications. It is the contention of this thesis that this latter view is mistaken. Prolog programs executed using currently available implementations may be much slower than equivalent C or Pascal programs but this is not inherent in the language. Prolog can be efficiently implemented. This thesis presents such an implementation, a high performance Prolog compiler for the MIPS RISC architecture.

This work does not consider Prolog's theoretical flaws and indeed ignores logic programming and its declarative ideals. It instead considers how to maximise the performance of a Prolog implementation. Prolog may be a stepping-stone soon to be made obsolete by the evolution of new logic programming languages. However such languages, at least in pragmatic terms, must be steps in the dark if we do not understand how to implement Prolog efficiently. It is essential that we know which Prolog features can be implemented efficiently and which features cannot.

Parallel machines may dominate the future of computing. Prolog's amenability to parallel implementation may result in parallel versions of Prolog being very successful. This thesis ignores any issues related to parallelism. If we do not understand how to implement Prolog efficiently on a single processor, we cannot expect to understand how to build implementations for many processors.

The blame for the inefficiency of existing Prolog implementations has been laid on the large semantic gap between Prolog and the machine architecture used to execute it. The better performance of implementations of languages such as Pascal or C is attributed to them being closer, both in philosophy and operations, to the machine architecture. This view has motivated the development of high-level machine architectures with instructions much closer to the basic operations needed to implement Prolog. This closes the semantic gap. Both seminal and pre-eminent among machine architecture designed for Prolog is David H. D. Warren's abstract machine, the WAM [War83]. The instructions of the WAM perform complex operations such as unification. This is in complete opposition to a recent strong movement in computer architecture design, the development of reduced instruction set computers (RISC). The tenet of this movement is that machine architectures with very simple regular instructions are the most efficient way to exploit current hardware technologies. Although this tenet is far from established, the recent successes of RISC architectures must cast considerable doubt on movement in the opposite direction. This concern has led to architectures which attempt to offer better support for Prolog but with an instruction set simple enough to obtain the benefits of a RISC such as Mills' LOW RISC [Mil89] and the BAM from Berkeley [Hol90]. The simulated performance results for the BAM are particularly promising, suggesting such an approach has great merit.

Even if machine architectures designed for Prolog are technically successful, pragmatic obstacles may prevent their adoption. Many of the existing Prolog machines can execute only Prolog and so, in most environments where support for multiple languages is essential, they are useful as a co-processor for a general-purpose processor. This may make them less attractive. A sufficient market may not exist to support a machine designed for a particular language, particularly a language such as Prolog which is yet to be widely adopted. Even if the market exists, it may not provide the resources necessary for Prolog machines to keep pace with the aggressive improvement in hardware technologies. These factors leave Prolog implementations for existing general-purpose machines attractive even if their performance falls short of that possible with special-purpose hardware.

The lessons learnt from attempting to implement Prolog efficiently on a general-purpose machine are still relevant to at least some Prolog hardware. This is

1.

particularly so where a general-purpose architecture is extended to support Prolog, such as Berkeley's BAM. The benefits of particular extensions may change significantly with the use of more sophistication in the software implementation.

Rather than closing the semantic gap this work attempts to bridge the semantic gap better by improving compiler technology. Existing implementations examine a Prolog program incremently by viewing each predicate or even each clause separately. The main conclusion of this thesis is that there is great advantage for a Prolog compiler in examining the program in its entirety.

The focus of this thesis is the implementation of Parma†, a Prolog compiler for the MIPS RISC architecture. Parma was constructed to examine compilation techniques which could yield high performance from Prolog programs. Parma's implementation is described and, in particular, the critical global analysis phase which Parma employs to examine the whole program is described in detail. The performance of Parma on a benchmark set of programs is presented and the utility of the compilation techniques employed is assessed by examining the improvements they yield on the benchmark set.

Much of this work has already been presented to the Prolog community [Tay89], [Tay90], [Tay91] and [Tay91a].

1.1 PREVIOUS WORK

The first Prolog implementation was an interpreter [Rou72]. By an interpreter we mean an implementation which executes the program clauses in an internal form little different to their external form. Typically, the internal form of the program clauses is similar to a parse tree.

† Parma is named both for *Macropus parma* the small critically endangered wallaby of coastal NSW forests and for *Parma microlepis* the pugnacious pomacentrid fish of NSW coastal reefs. Those desperate for an acronym can interpret it as Prolog for A Risc Machine Architecture.

In most aspects an interpreter is the best way to implement Prolog. Interpreters can be implemented cheaply and easily. They can be made highly portable between machines. They can load new programs quickly, expediting the important edit-test cycle during development. It is easy to provide a comprehensive debugging environment in an interpreter. Predicates which modify the program at run-time, such as *assert*/1 and *retract*/1, are easy to implement in an interpreter.

Prolog interpreters have only one drawback - poor performance. They are relatively slow at executing programs and can also be expensive in the amounts of memory they use. If the performance of interpreters was sufficient, new Prolog implementation techniques would be of minor interest. Unfortunately the performance of interpreters is inadequate and much work has been devoted to remedying this. This is not to say interpreters are useless. Most implementations include interpreters for use in executing predicates which can be dynamically modified, for providing debugging environments and for when the user requires quick program loading. Indeed, with the recent improvement in general machine performance, an interpreter is the most appropriate tool for some applications, for example education.

Subsequent Prolog interpreters [Bat73], [Bru76], [Sze77] improved on the techniques used in Roussel's experimental implementation producing more practical implementations. The first Prolog compiler was constructed by David H.D. Warren for the DECsystem-10 [War77]. It produced an order of magnitude speed improvement over the existing interpreters. It also was more space-efficient than the existing interpreters which was very important on the memory-poor machines of the time. Although Warren's compiler was a landmark establishing conclusively that Prolog was a practical language, it has had surprisingly little direct influence. It did not see widespread use, was not ported to other machines and had no direct descendants.

Warren's compiler compiles Prolog clauses firstly to the instructions of an abstract Prolog machine. These abstract instructions are subsequently translated to DECsystem-10 instructions. This abstract machine was much closer to Prolog than a conventional machine such as the DECsystem-10. Roughly speaking each symbol in a Prolog clause produces one abstract instruction. Single instructions perform operations such as unification and clause indexing.

Warren refined and generalised the abstract machine employed by his compiler in [War83] to produce what is widely-known as the Warren Abstract Machine or WAM. The paradigm of the abstract machine has dominated research in Prolog implementation since. Not only has the WAM been widely used and considered but there have been variants of the WAM, extensions to the WAM and independently developed abstract machines of similar function. In the following discussion we will use the name WAM to refer to the set of abstract machines similar to the Warren Abstract Machine and the remarks should also apply to most Prolog abstract instruction sets of similar functionality and level of abstraction.

The primary motivation for the design of abstract machines was the construction of Prolog hardware. However, paradoxically, they have seen wider application in software. Implementations which first compile Prolog clauses to abstract machine code and then execute using an interpreter have been very successful. SICStus Prolog [Car88] is a typical example. This approach can produce much better performance than an interpreter but, unlike a native-code compiler, still be easily portable between machines. A similar approach is presumably also used in many of the successful commercial Prolog implementations whose internal details are kept proprietary.

Although Prolog machines have seen little commercial production, a number of successful prototypes have been built. These include the PLM [Dob87] built at Berkeley, the KCM [Ben89] built at the ECRC and the PSI-II [Nak87] built as part of the Japanese Fifth Generation Computer Systems project. The performance of these machines was considerably greater than the software implementations available at the time for comparable general-purpose machines. It could be concluded that Prolog hardware based on abstract machines can offer a significant performance advantage. We feel this is misleading because considerable improvement could be made to the software implementations used in the comparison.

In particular, we feel the use of the abstract machines has limited the performance of software implementations. The instruction set of the WAM is far too coarse and too high-level to allow many of the optimisations which are not only possible but extremely important on general-purpose machines. The simplicity and elegance of the WAM, which are great virtues in other respects, become a straight-jacket for implementations wishing to exploit a conventional machine to its full. For example, the dereferencing operation of the WAM model is expensive to implement on a conventional machine but it is integral to the instructions of the WAM. An implementation which wishes to remove the expensive dereferencing operations must go beyond the WAM. A high-performance implementation could still use the WAM as an intermediate language but great transformations must be made below the level of the WAM. The WAM is hiding operations basic to the efficiency of an implementation.

The use of the WAM in software implementations such as SICStus Prolog was not a mistake. These implementations had important goals other than just maximising performance, such as portability. The use of the WAM no doubt helped meet these goals. It is somewhat misleading to compare such implementations with their many goals with Prolog hardware where great effort has been invested in obtaining high performance.

The excellent performance of the Berkeley Abstract Machine (BAM) [Hol90], may bring a fundamental change in the direction of Prolog hardware design. The BAM is a general-purpose machine extended with a carefully chosen set of instructions to support Prolog. The general-purpose subset of the BAM's instruction set is very similar to the MIPS architecture. The BAM's implementors believe that the additional instructions designed to benefit Prolog alone offer significant performance improvement but have minimal impact on the performance of the general-purpose instructions and add only modestly to the cost of the chip.

Much recent work in the area of Prolog implementation is based on the WAM and hence much of it is of little relevance to a native-code compiler, such as Parma, which makes no use of the WAM instruction set or anything similar. However, Parma's memory model is similar to the WAM. As a result work on garbage collection such as [App87] remain very relevant and the techniques described could be employed in a garbage collector for Parma.

Although Parma's memory model is similar to the WAM, both its data and instruction referencing characteristics are significantly different. This means that studies of Prolog memory referencing characteristics and Prolog cache behaviour based on the WAM, such as [Mad89], cannot be safely applied to Parma.

1.1

Evan Tick [Tic87] made a detailed study of the memory referencing characteristics of Prolog architectures. Tick derives an ideal canonical Prolog architecture. This is particularly interesting because this architecture has many similarities to the WAM. Unfortunately, the assumptions made in deriving this ideal architecture are not valid for Parma. The basic problem is that Tick assumes that the predicates of the program will be compiled independently. This is quite a defensible assumption as without it an ideal model would be very difficult to construct. Nonetheless it is not valid for Parma and as a result the code from Parma may make significantly fewer data references than Tick's ideal model.

The only full description of a Prolog compiler published since Warren's seminal work is Peter Van Roy's compiler for the Berkeley PLM [Van84]. It is a detailed technical description of a compiler which produces reasonable PLM code. The issues discussed are of little relevance to Parma because no global analysis is attempted and because of the difference between the PLM instruction set and the MIPS architecture.

There have been a number of papers on aspects of Prolog compilation [Bow86], [Deb85], [Van87], [Car89], [Dem89], [Mar89a]. Many of the improvements suggested in these papers are relevant to, and are indeed incorporated in, Parma. However much of the work in these papers is devoted to how to apply these improvements to the WAM and in many cases necessary extensions to the WAM for these improvements to be applied are described. Not only is this irrelevant to Parma but it is easier to apply the improvements free of the strictures of the WAM and often a more effective generalised form can be applied.

Two recent papers on Prolog compilation are very relevant to Parma. Peter Van Roy's work described in [Van89] and [Van90] essentially parallels the work on Parma. Van Roy is also constructing a Prolog compiler based on global analysis and is producing similar results. Although much of the detail of Van Roy's work is not yet available, it appears that there is considerable overlap between the techniques he has employed and those employed in Parma. Comparison with the final results from Van Roy's work will be interesting and his results should bolster the conclusions of this work.

In contrast to previous work on Prolog compilation there is an extensive body of work on the analysis of Prolog programs. This is discussed at the beginning of Chapter 3.

1.2 LIMITATIONS

Assessment of a programming language implementation is not easy. Ideally subjective concerns should be assessed by distributing the implementation to a large number of users and considering their reports. Objective concerns should be assessed by measurement of the implementation’s performance over a large set of real programs. The resources available for this thesis have not allowed such a full evaluation so the results presented cannot be completely satisfactory. Nevertheless we feel they are sufficient to support the central claims of the thesis.

Much of the work in producing a Prolog implementation lies outside the core necessary for pure Prolog. As a result of limited resources several features outside the pure Prolog core have yet to be implemented in Parma. Parma has no garbage collector, no debugging environment, no floating point arithmetic and many built-in predicates have not yet been implemented. Importantly, although these features have not yet been implemented, provision has been made for them all. For example, bits in Parma’s data representation have been reserved for use by a garbage collector. All these features could be added to Parma without any penalty to the current performance. The performance results presented would therefore remain valid.

Parma has also ignored the cyclic terms that arise when a free variable is unified with a compound term which contains that free variable. As a result unifications involving such terms can loop forever. This is commonly known as the occur-check problem because one approach would be to ensure such unifications always fail. This approach can be expensive so many implementations instead modify their unification algorithms and certain built-in predicates to ensure they do not loop. Although a real implementation should have to address cyclic terms we feel ignoring them does not significantly affect our performance results.

In the work on Parma we have ignored some performance aspects of Prolog. While Parma assumes programs may be large it assumes individual predicates will not be large (more than 100 clauses). Although such predicates are correctly handled their effects on performance are ignored. Such predicates raise a completely different set of issues which are addressed by the deductive database community. The NAIL! system [Mor86] forms an interesting counterpart to Parma. It implements a Prolog-like language but focuses on the performance aspects that result from very large predicates.

Parma also makes no attempt to improve the performance of certain expensive built-in predicates. These include predicates which implement input and output such as *read*/1 and *write*/1, predicates which allow dynamic program modification such as *assert*/1 and *retract*/1 and database predicates such as *recorda*/3 and *recorded*/3. Although it may be possible to improve the performance of these built-in predicates, the issues involved were largely uninteresting because they are not germane to the rest of the work on Parma.

This thesis is also limited in the target machine architecture considered. The only code generator implemented in Parma is for the MIPS RISC architecture. Although other code generators could be provided, portability has been a secondary consideration in the development of Parma. The discussion of the costs and benefits of implementation decisions is largely with respect to the MIPS architecture. It would be interesting to examine the performance of Parma on other target machines. Unfortunately this would have required considerable extra work beyond the scope of this thesis. Discussion of implementation decisions could have been made with respect to other target architectures but this would have been cumbersome, and because of the difficulty in making it comprehensive, of little extra value.

Given that only a single target machine architecture could be incorporated in this work we believe the MIPS RISC architecture was an ideal choice. Prolog can be implemented on this platform without the problems found with some other architectures. For example, Prolog implementation is hampered on the Motorola M68000 because there are insufficient registers to contain the Prolog execution state and the bit operations necessary for tag manipulations are expensive.

The MIPS architecture is available as high-performance scientific workstations and as CPU servers. This type of machine is widely used in the academic, research and development environments that are Prolog's current stronghold. The machines which employ the MIPS architecture provide some of the highest scalar uni-processor performance, both in absolute terms and in performance/price terms, available in commercial machines. This makes the MIPS architecture a very suitable target for a high-performance Prolog implementation.

Ideally the target architecture chosen should be as typical as possible so that the results will correlate well with those that would be obtained for other architectures. However the MIPS architecture cannot be said to be typical. Even other RISC architectures, such as Sun’s SPARC with its register windows, are sufficiently different that Parma’s performance on these architectures could differ significantly from its performance on the MIPS. Although it is not typical, the extreme simplicity and regularity of the MIPS architecture make it attractive as a single target. Its instructions are, to a large extent, a subset of those available on most processors. This should at least minimise surprises when comparing performance to other architectures.

The simplicity of the MIPS architecture is a joy to the compiler writer. The simplicity of the instructions and the single addressing mode makes construction of a good code generator for the MIPS architecture much easier. The simple instruction timing makes assessment of the cost of code sequences very easy. The architecture’s only real gargoyle, instruction delay slots, is hidden from the compiler writer by the assembler.

1.3 MIPS RISC ARCHITECTURE

Although the roots of the RISC philosophy date back at least to the 1960’s and the work of Seymour Cray and others on the CDC 6600, the flowering of the RISC philosophy came at the start of the 1980s with three machines, the IBM 801, Berkeley’s RISC I and Stanford’s MIPS architecture. The MIPS RISC architecture [Kan88] is a commercial development by MIPS Computer Systems from Stanford’s processor. It is an elegant embodiment of one stream of the RISC philosophy.

The MIPS architecture is a 32-bit architecture. It has fixed format 32-bit instructions. It has 32 32-bit integer registers. There is a separate set of 32 32-bit floating-point registers which can also be treated as 16 64-bit floating-point registers.

Only load and store instructions reference memory. Other instructions perform their operations on registers. The load and store instructions have only one addressing mode, register offset. The effective address is calculated by the addition of a signed 16-bit constant to the contents of the specified register.

The MIPS instructions have only three formats and are designed to be highly pipelined. The intention is that the instruction execution rate approaches one instruction per processor cycle. Unfortunately, to make this possible some complexity has to be introduced into the architecture. Load instructions have a latency of one instruction. The instruction immediately following the load instruction cannot use the register which is the target of the load. This one cycle shadow is usually termed a delay slot. Usually it is possible to schedule a useful instruction in the delay slot but sometimes a *nop* (null instruction) must be placed there and a cycle wasted. It can take many cycles for the result of an integer divide, integer multiply or a floating-point instruction to be available. The processor stalls if a subsequent instruction uses the target register of these operations before the result is ready. It is often not possible to schedule useful instructions, which do not depend on the result, to follow such instructions and often they effectively take a number of cycles to execute.

Branch and jump instructions also have a latency of one cycle. The subsequent instruction is executed regardless of whether the branch is taken. Again it is usually possible to schedule a useful instruction in this delay slot but occasionally a *nop* must be placed there and a cycle wasted. Fortunately, instruction scheduling is performed well by the MIPS assembler reducing the burden on the compiler writer.

1.4 DEFINITIONS

The use of nomenclature in the Prolog community is not only sometimes inconsistent with external use but it is sometimes also inconsistent within the community. We make no claim that the nomenclature within this paper is optimal but hopefully with the help of the following definitions it will be at least clear.

term - A Prolog data object. A *term* is either a *constant*, *free variable* or *compound term*.

constant - A basic object with a name which is a sequence of characters. A *constant* is an *integer*, *float* or an *atom*.

integer - An *integer* is a constant whose name is a valid integer.

float - A *float* is a constant whose name is a valid floating-point number.

atom - Any constant which is not an integer or a float.

free variable - An uninstantiated object.

compound term - A structured data object consisting of an atom and a sequence of one or more terms called *arguments*.

arity - The number of arguments of a compound term is its arity.

functor - The pair comprising the name and arity of a compound term is its functor. We will write these separated by a '/'. For example, the functor of the compound term $f(x, y)$ is $f/2$. Many use *functor* to refer to only the name of a compound term. The above definition, at least for our purposes, is more useful.

structure - The memory object representing the top-level of a compound term.

list cell - A structure with functor '.'/2.

clause - A part of a Prolog program consisting of a head and a body.

fact - A clause whose body is either empty or a conjunction of unifications only.

predicate - A maximal set of program clauses the heads of which have the same functor.

1.5 THESIS LAYOUT

This thesis contains five more chapters. The next, Chapter 2, contains a description of both the data representation and the execution model Parma employs. Implementation choices are discussed, some with detailed, although informal, evaluation.

Chapter 3 discusses the Parma global analysis phase. The paradigm for this phase is execution over an abstract domain. The abstract domain is described and justified. The abstract execution algorithm is given. The limitations that global analysis imposes are discussed.

Chapter 4 presents the heart of Parma, its compilation phases. Parma's intermediate language is discussed. The techniques used to translate the Prolog clauses which have been annotated with information by the global analysis stages into this intermediate language are presented. The improving transformations which are applied to this intermediate language are also described.

Parma's performance on a set of benchmark programs is presented in Chapter 5. The value of the compilation techniques implemented is evaluated by comparing Parma's performance with and without each technique. Parma's performance is compared to existing Prolog implementations and the results available for Prolog hardware.

Chapter 6 gives the conclusions that can be drawn from this research and discusses its implications for future work on Prolog implementation.

2 REPRESENTATION

It is more difficult to represent the execution state of a Prolog program than the execution state of a C program. This is a result of Prolog being an untyped language and the more complex flow of control resulting from backtracking. In this chapter we examine the possibilities for representing the execution state of a Prolog program and describe the representation chosen for Parma.

Most Prolog implementors have been content just to describe the execution representations they have chosen. They do not attempt to evaluate or justify their choices. We do not doubt the wisdom of their choices, but, as these decisions are basic to the efficiency of a Prolog implementation, we feel it desirable to examine the choices.

There is a great advantage to be gained if we ensure that the representation of an object fits in a single machine word. This allows objects to be passed about in a single register and hence allows an efficient calling protocol for predicates. The information required to completely describe an object usually will not fit in a single word, e.g. the print name of a constant will not fit in a single word. Instead, the word must contain a reference to memory locations where further information about the object is stored.

Prolog is untyped so it must be possible to distinguish the type of an object from its representation. The approach usually adopted is to reserve certain bits of data words for a field whose value will indicate the object's type. This field is called the *tag*.

Prolog objects can be divided into three obvious categories: constants, free variables and compound terms. Prolog implementations usually subdivide these categories further for efficiency reasons. Extensions to Prolog may require the introduction of other types.

A garbage collection algorithm may require several bits in each data word to be reserved for its use [App87]. This, in combination with the bits used by the tag, may significantly restrict the bits available to store an address in a data word. To put this in perspective, if 26 bits are available of a 32 bit word and word addressing is used then 268 megabytes can be addressed. This may be sufficient for almost all purposes but even if there are currently no Prolog applications which require a larger address space, there certainly will be in the future. So reserving bits for a tag field in a 32-bit word may hurt some applications. Implementors of special-purpose hardware for Prolog [Ben89] and Lisp [Hil86] have avoided this problem by choosing a larger word size of 64 bits and 40 bits respectively. This will cease to be a problem on conventional machines if, as many predict, workstations move to a 64 bit architecture in the near future. As this thesis was being completed it was announced that the soon to be released MIPS R4000 workstation will support 64-bit operations [MIP91].

Another problem with using a tag field to distinguish types is that on some machines, e.g. the MC68020, examining and setting sub-fields of a word are relatively expensive operations [Mul87].

2.1 CONSTANTS

A constant is distinguished by its name. This name can typically be any string of characters. Some implementations include the empty string. The obvious representation for a constant is a pointer to the string of characters that form the constant's name. However if this representation is used, the unification of two constants entails a string comparison. This is a comparatively expensive operation, typically involving many instructions.

This can be avoided by instead employing a table with one entry for each unique constant occurring in the program and using the index or address of the table entry to represent the constant. This means the same representation is used for each instance of a constant and the representation is unique. This allows the unification of two constants to be done with a word comparison, a cheap operation, typically involving only one or two instructions. It makes creation of a new constant a more expensive operation but this is an infrequent operation in most programs. Only in the execution of a few built-in predicates, e.g. *read*/1, are constants created at run-time.

An extensible hash table [Knu73] is an appropriate data structure for the constant table. The table entry can also be used to store other information associated with the constant, e.g. its precedence as an operator. It should be that noted only built-in predicates access the fields of entries in the constant table. Pure Prolog code needs only the index or address of the entry to perform unifications.

2.2 NUMBERS

The representation we have described for constants would make arithmetic operations slow. Arithmetic predicates would need to convert their operands from strings of digits into the internal machine representation for numbers, perform the arithmetic operation, then convert the result to a string and search the constant table for this string.

These overheads can be ameliorated by refinement of the constant table. The conversion of operands from strings to numbers can be avoided if each constant table entry has a field for the numeric value (if any) of that constant. The conversion of the result from a number to a string can be avoided by storing all constants whose name is a valid number in a separate numeric constant table. This numeric constant table would be indexed (hashed) on the numeric value of the constant and the entry field which contains a constant’s name would be calculated only on demand.

This refined representation would still add considerable overheads to arithmetic operations. A search for the result in the numeric constant table, and possibly adding an entry to this table, would still be necessary. This overhead may be acceptable in a program which performs little arithmetic but could slow arithmetic-intensive programs by at least a factor of 2 and probably more.

The pattern based language AWK [Aho79] does not treat integers as a separate type. The implementors presumably believed that arithmetic intensive AWK programs are rare. Prolog has more pretensions to being a general-purpose language than AWK and hence such arithmetic overheads are unacceptable.

A specialised representation for numbers can be used if they are treated as a separate type to other constants. The machine bit patterns for integers and floating point numbers can be employed in their representation. Not only does this avoid conversions to and from strings of digits but it has the added advantage that integers and floats can then be stored in a single word.

Unfortunately, the range of integers and floats expressible must be pruned to allow room in the word for the tag and possibly other fields. Thus, for example, on a machine with 32-bit words integers may be limited to 28 bits. This should be adequate for many programs but will certainly be a problem when translating programs written in other languages which assume availability of, for example, 32-bit integers to Prolog. It also makes it expensive for Prolog implementations to conform to floating point standards which expect 32 bits to be available.

Most implementations treat integers and floats as a separate type not just internally but also, distinguished by syntax, at user level. This results in the integer 1 not unifying with the atom of the same name, i.e. 1 = '1' fails. This is not necessary in the case of integers. If a one-to-one mapping can be established between valid integer names and integer values then an implementation can convert all constants with valid integer names to the internal integer representation and yet recover their original name when needed. The syntax of valid integer names would have to be restricted slightly. Leading zeros and a sign on zero and positive integers would need to be forbidden.

Establishing such a mapping for floats is not feasible. Many real numbers map to the one machine floating-point representation. The original name cannot be recovered from the machine representation. Given that floats must be visible to the user as a separate type it does not seem worthwhile eliminating integers as a separate type. Therefore Parma, like most implementations, not only treats integers and floats as separate types internally but also externally, visible to the user.

2.3 STRUCTURES

The obvious representation for structures is that typically used. A structure of arity n is represented by a vector of n+1 words. The first word is used to indicate the functor of the term. The following n words contain the structure's arguments. Some implementations restrict the arity of structures to below a limit such as 255. Such a restriction allows the arity of the term and the constant table index of the functor name to be packed directly into one word. This restriction is reasonable in that such large terms are unlikely to appear in the text of a human written program. However some sophisticated applications automatically-generate Prolog programs which when executed yield desired results. It is not clear that terms of large arity will not naturally occur in the text of such programs.

Many Prolog implementations do not provide arrays. This is unfortunate as arrays are a natural and efficient data structure for some problems. This results in programmers using structures as single assignment arrays. This is done by using the built-in predicates *functor/3* and *arg/3*. Each argument of the structure corresponds to an array element. Many applications require large arrays so a restriction on structure size is very deleterious in this respect.

The limit on structure size could be removed by using a complete word to store a structure's arity. A structure of arity n would then need n+2 words. The extra costs of this approach in memory space and memory accesses can be avoided by the following more complex scheme.

A table is maintained with an entry for each functor for which structures exist in the current program. Each table entry contains fields indicating the functor name and arity. It may be useful to store other information in each table entry such as the location and type of clauses for the predicate of that form. Functor table indexes are used to indicate the functor of a structure. New functors are encountered at run-time only by built-in predicates such as *functor*/3 and *read*/1. The table must be searched for the new functor and, if not present, it must be added. An extensible hash-table allows this to be done at reasonable cost. This operation is the only significant extra overhead this scheme incurs compared to packing the name and arity directly into the first word of a structure. The cost is minor and it would seem advisable for implementations that do not provide arrays to use this scheme to allow structures of unrestricted arity. Prudence and engineering principles would suggest that even implementations with arrays might be wiser not to limit structure arity. Parma uses this scheme.

2.4 LISTS

Lists are very common and often are the predominant data structures in Prolog programs. By convention lists are constructed using terms of the functor '.'/2 for list cells and the constant '[]' as the list terminator. Thus the list of the first three letters of the alphabet would be represented by this term: '.'(a, '.'(b, '.'(c, []))). This representation is clumsy both to read and to write so most implementations add syntactic sugar to their parser and output predicates, allowing them to be read and written in a more convenient form. The previous list might be written as [1, 2, 3].

The normal structure representation results in three words being used for each list cell, one word each for the functor ('.'/2), the *car* and the *cdr*. More efficient representations are possible and because of the ubiquity of lists many implementations treat structures with functor '.'/2 specially.

A common improvement is to allocate a separate tag for structures of functor '.'/2 allowing the word indicating the functor to be removed. This reduces the memory required for a list cell to two words. The gains from this optimisation are more modest than would be expected from its wide use. For example, it allows Parma to reduce the critical piece of code in the *nreverse* benchmark, the recursive clause in *append/3*, from eight to seven instructions. The improvement in execution time of roughly 12% for *nreverse*, which does nothing except manipulate lists, is presumably larger than for most programs. The reduction in total memory use will be more variable but programs where maximum memory use is reduced more than 15% should be rare.

The costs are harder to assess. The cost of allocating a tag to lists is highly dependent on the allocation of other tags. There will be little or no cost if there is a free tag value. On the other hand the need for an extra tag value may force the tag field to be increased in size, which could have substantial costs. For example, the advantages of a two-bit tag-field over a larger tag field are described in Section 2.7.

The separate tag for list cells means the built-in predicates which access structures must have extra code added. The two most important of these are *functor/3* and *arg/3*. Not only is it undesirable that these two predicates be slower as they are frequently used but it is also undesirable that they be larger as calls to them are often compiled to in-line code. It is hard to assess the overall performance penalty from allocating a separate tag to list cells. However the gains seem worthwhile for implementations such as Parma where a separate tag value is available and high performance is a primary goal. It may be wiser for implementations with other aims to avoid the extra complexity of a separate tag for list cells.

2.5 COMPACTING LISTS

Lists are even more important in the language Lisp. Lisp implementors have employed more compact list representations which use less than two words per list cell. These representations are collectively called cdr-coding. Optimum performance from cdr-coding results in memory use approaching one word per list cell. A variety of schemes have been described and implemented [Li86]. Basically these schemes take advantage of the fact that most commonly the cdr of a list cell is a reference to another list cell and often this list cell will occur at an address immediately following, or at least close to, the first list cell. In these common cases, cdr-coding schemes use bits in the car field to indicate the cdr field implicitly. This allows list cells often to be reduced to a single word. However, cdr-coding makes it more expensive to determine the cdr of a list cell. The primary use of cdr-coding has been on Lisp machines where extra hardware can be employed to ameliorate this cost. Without the benefit of hardware assistance the cost is formidable.

The simplest cdr-coding scheme we can envisage, without global analysis, changes the MIPS code required to obtain the cdr of a list cell from one instruction to three (if cdr-coded) or four (if not cdr-coded) instructions. This means a data access has been possibly avoided but at the expense of two or three extra instruction accesses. This increase in total memory accesses is not necessarily detrimental because of the different characteristics of instruction and data caching. Precise analysis is difficult but we can gain a notion of the tradeoffs involved by making some plausible assumptions.

Suppose that the data cache miss penalty is five cycles, no instruction cache misses are caused by the extra instruction accesses resulting from cdr-coding, no paging occurs and list compression is such that 80% of cdr references are removed. Cdr-coding reduces total execution cycles only if 50% of the avoided cdr references would have provoked, directly or indirectly, a data cache miss. This is very unlikely.

Clearly if a substantial amount of paging occurs then cdr-coding could produce significant net benefits. Assessments involving paging are still harder to make but, given current trends in physical memory sizes, it seems unlikely that many applications would have the necessary characteristics to benefit from cdr-coding.

So far we have only considered the costs and benefits of examining lists which may be cdr-coded. Lisp cdr-coding schemes such as [Li86] also entail significant extra costs in list cell creation. The techniques used cannot readily be applied to Prolog because they depend on the car and cdr being known when the list cell is constructed. Typically this is not the case in Prolog programs. The car is usually known but the list cell is often constructed with the cdr as an unbound variable. Typically this variable is instantiated by a subsequently (often recursively) called predicate. Often the list cell construction could be delayed until the cdr was known but this would entail copying and could preclude last call optimisation discussed in Section 4.7 which has substantial benefits.

The Berkeley PLM [Dob 87] cleverly extends the WAM to support cdr-coding. The representation used is simple. One bit is reserved in all data words. This cdr-bit is set only in words that are the actual cdr of a list cell. This means that the cdr-bit not being set in the cdr word of a list cell implies that the cdr is actually a list cell whose car is stored in that word. The PLM's cdr-coding scheme has a basic weakness: only list cells whose cdr is known at compile-time are compressed. Lists which are created during execution one element at a time, as most are, are not compressed at all. This is very different to the cdr-coding schemes used for Lisp which are carefully designed to compress lists as they are created at run-time. For example, the PLM scheme would achieve no list compression at all on the simple *pri2* benchmark, whereas any of the LISP schemes should obtain maximal list compression.

It is hard to assess the benefits of PLM cdr-coding. Touati and Despain[Tou87] report PLM list compaction rates for a number of benchmarks and try to estimate the consequent benefits. Useful compaction rates, in the range 35% to 83%, are obtained from roughly half the benchmarks. However, they note that these compaction rates may be atypically high because of the number of static lists, mainly of input data, in the benchmarks. It is also interesting to note that, as described in Section 4.3, Parma would create such lists at compile-time if they do not contain variables. This makes irrelevant the reduction in the number of memory accesses during the creation of these lists that cdr-coding produces. This would reduce significantly the benefits for cdr-coding reported in [Tou87], halving the compaction rate reported for many of the benchmarks.

There is no reason to doubt that a cdr-coding scheme, which is much more aggressive than that of the PLM, such as [Li86], could be adapted to Prolog and obtain excellent compaction rates. The difficult question is not whether good compaction rates can be obtained but whether overheads low enough to be viable can be obtained. Touati and Despain believe that the costs of implementing cdr-coding in the PLM hardware are sufficiently low but they do not believe that a similar scheme would be viable on a conventional machine. We certainly could find no cdr-coding scheme which would be useful in Parma. However, it is possible that more sophisticated approaches to cdr-coding could be successful on a conventional machine.

The overhead cdr-coding incurs during the creation of lists can be avoided by not compacting lists during normal execution but rather employing a garbage collector which compacts lists when it is invoked. This leaves the overhead of determining the cdr when examining lists. Our previous naive analysis suggests this alone is still sufficient to make cdr-coding unviable. This overhead could be much reduced if at compile-time it could be determined whether a particular list would always be or not be compacted when encountered at a particular program point. This seems impossible if compaction is done only by the garbage collector as no information on compaction would be available at compile-time. However, if it is known at compile-time where compaction occurs, then global analysis may be able to determine for a useful fraction of program-points whether a list will always be compacted or always not. Similar global analysis could also detect that a particular list can always be created compacted. We cannot even guess if global analysis can reduce cdr-coding costs sufficiently to be viable and still retain a useful degree of compaction.

An alternative approach is to introduce a new vector data type at the user level. Global analysis could be employed to transform uses of lists to uses of this vector data-type with consequent saving in memory use and accesses. This approach is much less flexible than the more aggressive cdr-coding schemes but has the advantage that the vector data type may be useful to users in other ways.

2.6 FREE VARIABLES

It is not the representation of a single free variable that is difficult. All that is necessary for this is a value which can be distinguished from bound types. The difficultly lies in the nature of Prolog's logical variable. Free variables can be unified together without being bound. We will refer to the state of such variables as aliased. A unification which binds a variable must also bind all the variables to which it is aliased. The unification of two unbound variables is not a common event in most programs. It accounts for only a small proportion of unifications. A variable representation must also provide the capability for backtracking. There must be some mechanism to allow the effects of unifications to be undone.

The WAM variable representation is an elegant solution filling the above constraints. It has a reference type distinguished by a separate tag. Words with this tag contain the address to another word. The unification of two free variables is achieved by changing one to refer to the other using this reference type. Subsequent unifications follow the chain of references and bind only the last variable in the chain. As a consequence it may be necessary to follow an arbitrary length chain of references to obtain the value of a variable.

Whenever a unification changes a free variable, either in setting it to refer to another free variable or in binding that variable, it records the addresses of the free variable on a stack named the *trail*. When backtracking, the WAM undoes the effects of previous unifications by popping addresses from the trail and setting the word at each address to the unbound value.

There is a choice when unifying two free variables of which variable to set to refer to the other. This choice is important. If the referred-to variable is deallocated before the last use of the referring variable, then the result, often called a dangling reference, is unfortunate. The WAM uses two rules to determine this choice and ensure that dangling references do not arise. If one variable is on the global stack and the other variable is on the local stack then the local variable should refer to the global variable. If both variables are on the same stack then the more recently created variable should refer to the old variable. Both these rules can be implemented by single comparison of variable addresses if the local and global stacks are placed in the appropriate order in memory.

The WAM does not allocate a separate tag for free variables nor does it have a particular unbound value. Instead it represents a free variable by a self-referring reference. This representation has the advantage that loading a variable's memory location into a register will produce the desired result regardless of whether the variable is bound or not. This is not true of other representations. As each free variable contains a different value, initialising numbers of free variables can be more expensive than if each free variable contained the same value.

The WAM variable representation is an elegant and efficient solution but it has a pervasive cost. It may be necessary to follow an arbitrary length chain of reference to obtain the value of any variable. This operation, usually termed dereferencing, is expensive to implement on the MIPS and many other conventional machines.

In practice, long reference chains form only very rarely [Tou87]. Global analysis of the program tracing the formation of reference chains can allow a compiler to remove many of the dereferencing operations [Tay89].

This may not be possible if modules of a program are compiled independently. The user can assist type analysis of a module by supplying entry and exit type declarations for predicates external to the module. It is not feasible for the user to similarly declare the reference chains of arguments on entry and exit. Reference chains are beyond the purview of the user. Concern about poor results from reference chain analysis within modules prompted adoption of an alternate variable representation in Parma.

The representation is similar to the WAM in that there is a reference type which is used to link aliased variables. Unlike the WAM, aliased free variables are kept in a circular list. When two free variables are unified the two circular lists are joined into one. When a unification binds a free variable the circular list is followed to set all variables to the bound value. Similarly to the WAM, a single unbound value is represented by a self-reference. This is natural as this is a circular list of one element.

Unlike the WAM the effects of past unifications cannot always be undone solely by setting a number of variables to unbound (a self-reference). It must be possible to restore a circular list of variables which has been destroyed either by binding or by merging with another circular list. This requires the trail entry must contain two fields: the former value of the variable and its address. This value-trailing is, of course, more expensive than the WAM's address-trailing.

Ensuring dangling references do not result from the allocation of variables is more difficult with Parma's scheme. When a local variable is deallocated, which usually occurs on exit from a predicate, a routine is called to check if that variable is free and aliased to other variables. If so, it is deleted from the circular-list of aliased variables. This is an expensive operation. However it is not necessary if the variable is known to be bound, or known not to be sharing with variables other than local variables being deallocated at the same time. This will very frequently be the case, providing global analysis is yielding precise results.

The advantage of Parma's representation is that variables do not need dereferencing. This saving is unfortunately matched by the added cost of binding a variable which involves following a circular list binding each variable. The result is that Parma's representation offers no advantage if the compiler has no information about variables.

As unification of two free variables rarely occurs, in many cases it will not be necessary to follow a circular list to bind a variable. Parma gathers information on the formation of lists of aliased variables and this allows many binding loops to be removed. This is similar, although somewhat simpler and faster, to the reference chain analysis which an early version of Parma's global analysis phase [Tay89] used for the WAM representation. At least for small programs, the code Parma produces is almost identical to that which resulted from earlier versions of Parma which used the WAM representation and reference chain analysis. For example, the small benchmarks in [Tay90] produce very similar code and, as a result, very similar performance.

Trailing operations are more expensive with Parma's variable representation but the information from Parma's analysis phase allows most of these operations to be removed. The programs in [Tay90] have almost all trailing operations removed.

The advantage of Parma's representation arises in cases where the compiler knows that a variable is bound but has no reference chain information. In this case the WAM's representation would require a dereferencing operation whereas Parma's representation does not. This is precisely the situation which will commonly arise when compiling a module with user declaration of entry and exit types of external predicates. This allows calculation of good information regarding the mode of variables but only poor information regarding reference chains.

Parma's representation has a further advantage where there is no information regarding a variable in cases where execution should fail or produce an error if the variable is unbound. A dereferencing loop must be produced for the WAM representation whereas Parma need only examine the variable's tag. This is likely to be necessary anyway. This case arises with the arguments for many built-in predicates and is thus significant. This saving is particularly important if calls to the built-in predicate are to be compiled in-line.

The adoption of Parma's variable representation, with its extra complexity, on the basis of conjectured but not demonstrated need over the simpler and well-used WAM variable representation cannot be justified as an engineering decision. It was chosen because it was interesting in its own right and because it was felt that even if it provided no advantage it could not be sufficiently disadvantageous to imperil the results from Parma. In hindsight the difficulty of ensuring dangling references do not occur outweighs the benefits. Variable representation is an area worthy of further examination.

2.7 TAGGED INTEGER ARITHMETIC

The integer arithmetic instructions of the MIPS and indeed most conventional machines expect their operand registers to contain only the two's-complement representation of the operand. As the internal Prolog representation of an integer contains other information a number of extra operations may be necessary to perform arithmetic. The following steps may be necessary:

- check the operand types by examining their tags
- extract the operand values from their tagged representation
- perform the arithmetic operation
- convert the result to tagged representation checking for overflow.

These steps could change an integer addition from one MIPS instruction to eleven. However appropriate choice of representation can remove some or all of these steps. If efficiency is important the choices need to be examined carefully.

We will examine in detail the representation of integers. We assume that the integer representation consists of only two contiguous fields: the tag field and the value field containing the integer in two's complement form. Other fields with constant value, such as bits reserved for the garbage collector, can be considered part of the tag field for our purposes. These constant fields need not be contiguous but we can see no rewards from them not being so. This leaves two possibilities. The tag can be in either the high (most significant) bits or the low (least significant) bits of the word. The value chosen for the integer tag is also significant, in particular, a zero tag is useful.

[Ste87a] discusses in detail a scheme for reducing the tag manipulation and type checking overheads for arithmetic in a LISP implementation. They propose a scheme where the tag is in the high bits of the word and is the sign-extension of the integer. Thus positive integers have a zero tag and negative integers a tag with all bits set. They assign tag values such that the sum of two non-integer tag values, with possibly a carry-in, cannot result in an integer tag value without an overflow resulting. In the case of addition, this allows the type and overflow checking to be combined in a single operation after the addition. This scheme may not be applicable to Prolog if the garbage collector requires a zero field in the integer representation.

Table 2.1 presents the code necessary to perform some of the common arithmetic operations for each of the possible integer representations we have described. We have assumed that it is necessary for arithmetic overflow to cause an exception. Many, perhaps most, Prolog implementations allow overflow to occur silently but we feel this is unacceptable. The code only checks the type of operands in the *sign extended* case.

Table 2.2 summarises the cost, in execution cycles, of the code for each integer representation both with and without type-checking. We have assumed that only multiply and divide operations take more than one cycle to execute. We have assumed a constant multiply takes 5 cycles, a general multiply 10 cycles and a general divide 20 cycles. We have assumed, where a general multiply or divide is involved, that the type checking of operands can be placed in the shadow of the multiply/divide.

2.7

Tag	$R = X \pm C$	$R = X \pm Y$	$R = X * C$	$R = X * Y$	$R = X / Y$
Low zero	$R = X \pm C_1$	$R = X + Y$	$R = X * C$	$T_1 = X \gg N$ $R = X * Y$	$T_1 = X / Y$ $R = T_1 \ll n$
Low non-zero	$R = X \pm C_1$	$T_1 = X \pm Y$ $R = T_1 - V$	$T_1 = X * C$ $R = T_1 - C_2$	$T_1 = X \gg_a N$ $T_2 = Y \gg_a N$ $T_3 = X * Y$ $T_4 = T_3 \ll N$ $R = T_3 \mid V$	$T_1 = X \gg_a N$ $T_2 = Y \gg_a N$ $T_3 = X / Y$ $T_4 = T_3 \ll N$ $R = T_3 \mid V$
High zero	$T_1 = X \ll N$ $T_2 = T_1 \pm C_1$ $R = T_2 \gg N$	$T_1 = X \ll N$ $T_2 = Y \ll N$ $T_3 = T_1 \pm T_2$ $R = T_3 \gg N$	$T_1 = X \ll N$ $T_2 = T_1 * C$ $R = T_2 \gg N$	$T_1 = X \ll N$ $T_2 = Y \ll N$ $T_3 = T_2 \gg_a N$ $T_4 = T_1 * T_3$ $R = T_4 \gg N$	$T_1 = X \ll N$ $T_2 = Y \ll N$ $T_3 = T_1 / T_2$ $R = T_3 \gg N$
High non-zero	$T_1 = X \ll N$ $T_2 = T_1 \pm C_1$ $T_3 = T_2 \gg N$ $T_4 = V_1$ $R = T_3 \mid T_4$	$T_1 = X \ll N$ $T_2 = Y \ll N$ $T_3 = T_1 \pm T_2$ $T_4 = T_3 \gg N$ $T_4 = V_1$ $R = T_3 \mid T_4$	$T_1 = X \ll N$ $T_2 = T_1 * C$ $T_3 = T_2 \gg N$ $T_4 = V_1$ $R = T_3 \mid T_4$	$T_1 = X \ll N$ $T_2 = Y \ll N$ $T_3 = T_2 \gg_a N$ $T_4 = T_1 * T_3$ $T_5 = T_4 \gg N$ $T_6 = V_1$ $R = T_5 \mid T_6$	$T_1 = X \ll N$ $T_2 = Y \ll N$ $T_3 = T_1 / T_2$ $T_4 = V_1$ $R = T_3 \mid T_4$
Sign extended	$R = X \pm C_1$ $T_1 = R \ll N$ $T_2 = T_1 \gg_a N$ if $(T_2 \neq R)$ exception	$R = X \pm Y$ $T_1 = R \ll N$ $T_2 = T_1 \gg_a N$ if $(T_2 \neq R)$ exception	$T_1 = X \ll N$ $T_2 = T_1 * C$ $R = T_2 \gg N$ $T_3 = T_2 \gg_a N$ if $(T_3 \neq X)$ exception	$T_1 = X \ll N$ $T_2 = Y \ll N$ $T_3 = T_2 \gg_a N$ $T_4 = T_1 * T_3$ if $(T_3 \neq Y)$ exception $T_5 = T_1 \gg_a N$ if $(T_5 \neq X)$ exception $R = T_4 \gg N$	$T_1 = X \ll N$ $T_2 = Y \ll N$ $T_3 = T_1 / T_2$ $T_4 = T_1 \gg_a N$ if $(T_4 \neq X)$ exception $T_5 = T_2 \gg_a N$ if $(T_5 \neq Y)$ exception $R = T_3 \gg N$

X, Y are tagged integer variables. C is a constant. V is the integer tag value.
N is the size of the tag field. $C_1 = C \ll N$. $C_2 = V * (C - 1)$. $V_1 = V \ll (32 - N)$.

TABLE 2.1. Tagged Arithmetic: Code

The benefits of placing the tag in the least significant bits spring from a simple relationship between the integer, I, and its tagged representation, I_t, considered as a two's-complement integer: $I_t = I * 2^n + v$ where n is the width of the tag and v is the tag value. This allows addition, subtraction and constant multiplication to be performed directly on the tagged value and the tagged result to be obtained by subtraction of a correcting constant. For example, it allows us to calculate $X = 7 * Y + 3$ as $X_t = 6 * Y_t - (5 * v + 3 * 2^n)$. As in this case, the correcting constant can often be folded with other constants in the expression. A zero tag value is desirable as this makes the correcting constant unnecessary. Otherwise it is desirable that the tag value is as small as possible to reduce the chance of overflow before subtraction of the correcting constant.

The same approach is not successful if the tag is in the high part of the word because

	$R = X \pm C$		$R = X \pm Y$		$R = X * C$		$R = X * Y$		$R = X / Y$	
Type checking	no	yes	no	yes	no	yes	no	yes	no	yes
Low zero tag	1	3	1	5	5	7	11	11	21	21
Low non-zero tag	1	4	2	7	6	9	14	14	24	24
High zero tag	3	4	4	8	6	8	13	13	22	22
High non-zero tag	4	7	4	10	9	14	14	14	23	23
Sign extended tag	3	4	4	4	6	8	12	12	20	20

TABLE 2.2. Tagged Arithmetic: Costs

overflow can occur into the tag rather than causing a machine exception. The costs of checking for this outweigh the benefits.

Precise evaluation of each of the representations cannot be made without knowing the frequency of each of the types of arithmetic operations and the proportion of cases where type-checking will be necessary. We have no figures on the distribution of arithmetic operations. Our intuition is that addition/subtraction of a constant would account for at least half of arithmetic expressions with general addition/subtraction accounting for close to 90% of arithmetic expressions. This makes placing the tag in the low-order bits very attractive. Certainly a *low zero* tag is the best representation.

A *sign−extended* tag could be competitive with a *low non−zero* tag if type-checking of operand is needed in most cases. Parma's global analysis phase seems to perform particularly well with the variables in arithmetic expressions. This allows type-checking to be removed in considerably more than half of the cases. Typically it seems that type-checking will be needed in only 10-20% of cases. This makes the *sign−extended* tag representation unattractive.

2.8 TAG PLACEMENT

There are operations other than arithmetic whose cost depends on the placement of the tag in the data word. A particularly common operation is extraction of an address from within a tagged word. This does not necessarily involve shifting and masking operations. On the MIPS, as the addresses involved are word addresses, they need to be aligned on four byte boundaries to avoid a heavy execution penalty. This means the two least significant bits of every address are not used. This allows a two bit tag to be placed in this position. Some processors such as the IBM ROMP [IBM86], mask off the unneeded bits of word addresses. The MIPS produces an exception if the two least significant bits of a word address are non-zero. This assists the debugging of programs written in C and similar languages. However, it is not necessary to mask off the tag before using the address. Both load and store instructions use a register offset addressing mode. A constant byte offset is added to the address register to obtain the effective register. Providing the tag value is known, this offset can be adjusted to cancel the tag value. This is very useful as otherwise shifting and/or masking operations would be necessary when accessing the arguments of a structure, a frequent operation. This relies on the bits reserved for the garbage collection always being zero except when the garbage collector is actually executing. This is true of several garbage collection algorithms.

The subdivision of constants into separate types results in more than four tag values being necessary. The advantages of a two bit tag in least significant bits can still be obtained by splitting the tag into two fields: a two bit primary tag which distinguishes free variable, constant, list cell and structure and a secondary tag for constants which distinguishes atom, integer and float. This is the representation that Parma employs.

There is a conflict in the allocation of the primary tag value. Some arithmetic operations are cheaper if the integer tag value is zero. Unfortunately some unification operations are cheaper if the reference tag value is zero. This is because branching if a register is equal to or not equal to zero is a special case on the MIPS taking only one instruction rather than two. It is also a special case on some other architectures. Parma has assigned zero to the reference tag but we have not evaluated the choice. These decisions yield the data representation for Parma in Figure 2.1.

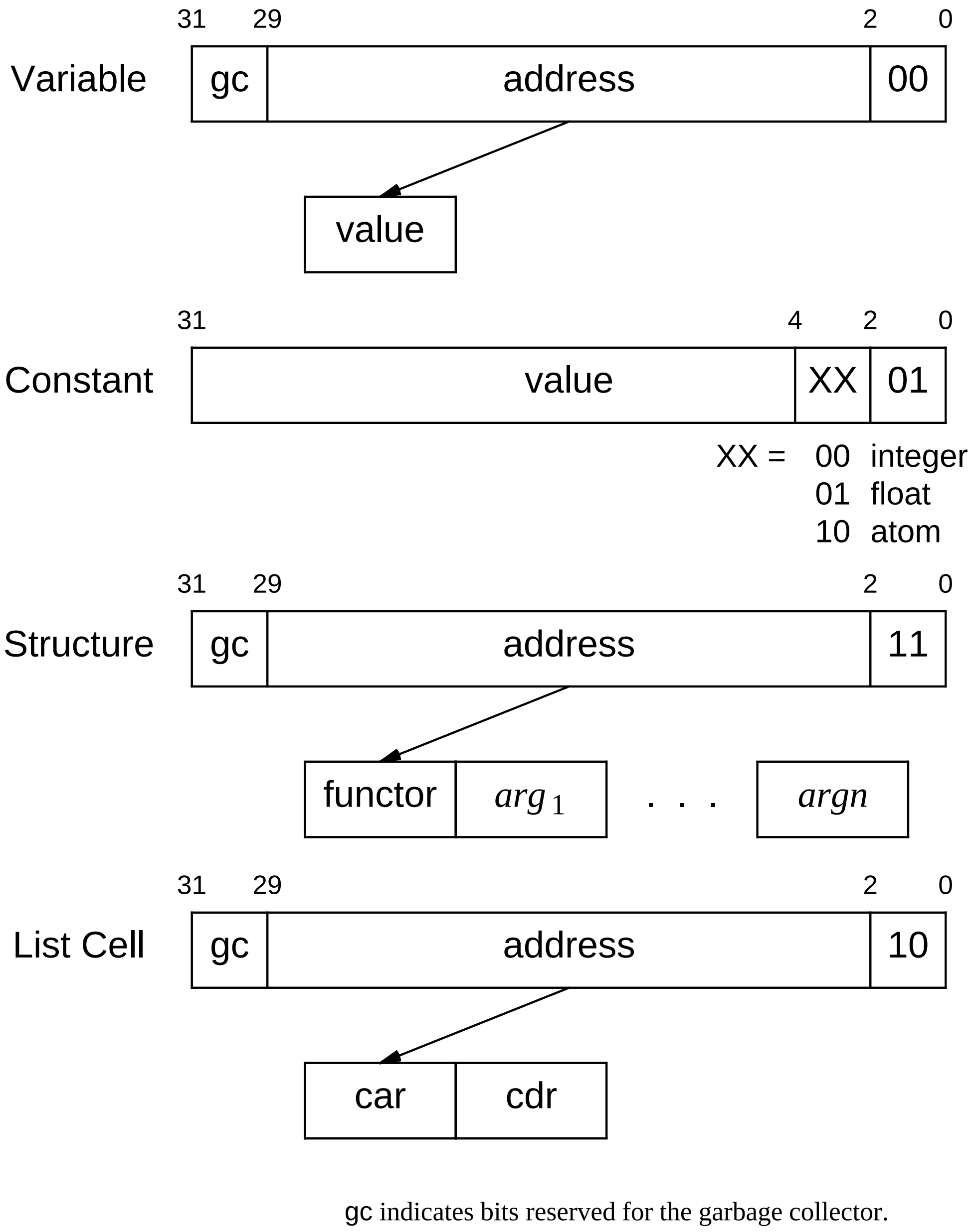


Figure 2.1. Data Representation

2.9 EXECUTION

This section examines how the Prolog execution state can be represented. Backtracking accounts for most of the complexity in representing the Prolog execution state. Without backtracking, the representation of the Prolog execution state can be done in a similar manner to imperative languages such as C.

Calling a Prolog predicate is analogous to invoking a C function. As with C, the possibility of recursion requires the dynamic allocation of memory for local variables and to store return addresses. Again, as with C, a stack is the natural way to provide this. We will term the memory used for this purpose the local stack.

Parma, like the WAM, passes predicate arguments through designated registers. This technique is also used by some recent implementations of imperative languages. For example, the MIPS C compiler employs this technique for up to the first four arguments of C functions. The common alternative is for the caller to push the argument values on to a stack. This is considerably more expensive than placing the values in registers but the extra work is not wasted if it allows the callee to avoid saving the argument values.

Global analysis allows Parma to determine that many predicates are determinate. As a result it often does not need to save argument values. This, combined with the high density of calls in Prolog compared to languages such as C or Fortran and the abundance of registers on the MIPS, makes argument-passing through registers the calling protocol of choice for Parma.

Memory must be dynamically allocated for structures. This could be done on the local stack but this would cause problems with the deallocation of memory. As a structure's lifetime is potentially the length of the program, it would often prohibit the popping of the local stack even on determinate exit from a predicate. This will often result in an unacceptably high memory consumption. Most implementations since [War77], including Parma, avoid this problem by using a separate stack for structures. It is often termed the heap, and although in purpose it is analogous to the heap of Pascal and similar languages, the name is misleading as, garbage collection aside, deallocation occurs only in a last-in-first-out (LIFO) manner on backtracking. The region of memory is also termed the global stack and that is the name we will use. It is occasionally useful to place variables as well as structures on the global stack.

There is little doubt that using a separate stack for structures is justified. The costs of managing an extra stack and allocating a register to be the global stack pointer are far outweighed by the reduced memory consumption.

The records of bindings which allow the effects of unification to be undone on backtracking could be stored in a linked-list interwoven through the global stack. Deallocation presents no problems as bindings are used and then discarded on backtracking in exactly the order that the other contents of the global stack are discarded.

The cost of a linking pointer in each binding record can be avoided by placing the binding records in a separate stack. This stack is usually termed the trail. This does not require an extra register as the pointer to the most recent trail record becomes the trail stack pointer. Parma's analysis allows most trailing operations to be removed but even then the benefits of a separate trail stack should still be significant.

On backtracking the execution state must be restored. Parma implements this in the same way as most Prolog implementations. At each program point, which backtracking may revert, it stores a vector containing the values of the registers necessary to restore the execution state. These vectors of register values are usually termed choice-points.

Choice-points could be placed on the global stack as their life-time is no longer than the other contents of the global stack. The execution of a cut can make the life-time of a choice-point short, and if structures have been created since the choice-point it would not be possible to deallocate the choice-point's memory, effectively leaving a hole in the global stack.

Choice-points could also be stored in the local stack. The local stack cannot be popped on non-determinate exit from a predicate as the variables on the local stack may be needed on backtracking. Thus a predicate which leaves a choice-point would not be able to pop the local stack on exit anyway. Deallocation of the local stack is not affected by placing choice-points on it.

A third alternative is to place choice-points in a separate stack. Unlike placing trail-entries in a separate stack this does not save any space. It would avoid the need in each choice-point frame for a linking pointer to the previous choice-point but the value of the local (or global) stack pointer which would be implicit in the choice-point's location on the local (or global stack) must now be included in the choice-point frame. There is no net gain. Parma places choice-points on the local-stack.

In summary Parma uses three stacks to represent the Prolog execution state:

- **local stack** - variables, choice-points and call return bookkeeping
- **global stack** - structure and occasional variables
- **trail** - binding records

Many implementations, including the WAM, use a similar representation of the Prolog execution state.

3 ANALYSIS

When compiling many languages, knowledge of the rest of a program may allow a particular piece of the program to be better compiled. The belief that this is particularly true for Prolog motivated us to construct a global analysis phase for Parma. The global analysis phase examines the program as a whole, to gather information to assist the subsequent compilation phase in its task.

The obvious empirical approach to information gathering is to observe the program's execution. This is usually termed *dynamic analysis*. Although dynamic analysis is employed by some compilers, it does not see widespread use because it is awkward to implement and the information it yields is of limited value.

Dynamic analysis causes two problems for users. Firstly, if the compiler is to execute the program the user must supply a suitable environment for the program's execution, including what should be typical input. Secondly, it may take a considerable time to execute the program, adding considerably to compilation time.

More importantly, as it is only possible to observe some of the possible executions during dynamic analysis, the information obtained is statistical in character rather than definitive. For example, dynamic analysis may reveal that, at a particular program point, a variable is very likely to be bound, but it cannot reveal that the variable will always be bound at that program point.

Definitive information is much more valuable to a compiler and for this reason many compilers attempt to determine properties of programs by means other than actual execution. This is usually termed *static analysis*. Algorithms which determine a number of properties of Prolog programs using static analysis have been published. The properties determined by these algorithms include: mode declarations [Mel85], [Deb86], [Bru87a], [Man87]; determinateness [Mel85]; variable sharing [Pla84], [Cit88]; obsolete structures [Bru87a]; functional computations [Deb86a]; polymorphic types [Zob87]; ground variables [War88]; reference chains [Mar89], [Tay89]; and trailing operations [Tay89].

There have been many fewer reports of implementations of these algorithms and their performance. The tenor of work on static analysis of Prolog programs has been much more theoretical than practical. The analysis algorithms seem to be more an end in themselves rather than a means to improving Prolog implementation. As a result, although issues such as soundness, completeness and to some extent complexity have been addressed, we are not yet certain that the algorithms can be practically implemented. Even more importantly, we do not know how useful the information produced by the algorithms is.

This deficit of information is being remedied. However the few recent papers that have assessed the benefits resulting from the analysis, e.g. [Mar89], [Muk90], [Tay90] have used as their data set either a small number of programs, or only small programs or even both. As a consequence, although these papers are encouraging, they are limited. The only exception to date is [Van90] in which Van Roy and Despain have used a more realistic set of programs.

The paradigm for much of the work on static analysis of Prolog programs is a technique named abstract execution. The concept of abstract execution was first applied to imperative languages [Cou77]. An abstract domain is chosen in which the symbols represent sets of values in the actual domain. The program is executed over the abstract domain. During this execution variables are assigned not actual values but abstract symbols which represent a set of values that the variable may possibly take at an equivalent point during actual execution. Operations whose actions cannot be determined at compile-time, such as the built-in predicate *read*/1, are handled conservatively by assuming they can produce any value.

The abstract domain is usually chosen to be a complete lattice of finite depth. This allows the results of recursions to be calculated by computing fixpoints in the abstract execution.

Information on the possible values of a variable at a particular program point can be gathered by noting the set of abstract values the variable takes at that program point during abstract execution. The choice of abstract domain determines the nature of the information that can be gathered. There is a trade-off in the choice of the abstract domain. In general, the simpler the abstract domain, the faster the abstract execution and the faster the convergence to fixpoints. The more complex the abstract domain, the more detailed the information that can be gathered. Parma's abstract domain is considerably more complex than that of other published implementations but abstract execution can still be accomplished in a reasonable time.

Parma's analysis phase is implemented by roughly 4,500 lines of SICStus Prolog. The remainder of this chapter will consist of a description of Parma's analysis phase. The correctness of algorithms for analysis of Prolog by abstract interpretation have been well addressed in the literature. We will not refine such a proof to show Parma's analysis algorithm is correct for we feel this would not much increase our confidence in our implementation. We feel that it is currently more important to address performance and practicalities than to extend the theory. As a result our description will be loose and informal concentrating on pragmatics rather than theory.

3.1 LIMITATIONS

It is necessary to apply some restrictions to Prolog so that Parma can apply global analysis both safely and effectively. The flow of control must be visible to the compiler. If all calls to a predicate cannot be determined then the arguments to the predicates must be assumed unknown. This usually results in little useful information being determined about that predicate. Parma's analysis phase depends on, by default, all calls to a predicate being syntactically visible.

Several built-in predicates make this impossible in general. One is the predicate *call*/1 which allows the user to construct and execute a call at run-time. Analysis may determine the functor of the argument to *call*/1 and hence determine the flow of control. In this case Parma abstractly executes a call to the appropriate predicate. The types of the sub-terms are used as the type of the arguments to the call.

It may be feasible to enhance the abstract domain and/or the analysis algorithm to reduce the frequency of cases where the functor cannot be determined, however there will always be some such cases. The only remedy, apart from abandoning global analysis and the information it yields, is to to seek information from the user.

Parma, as yet, does not have a mechanism for indicating the invocations of *call*/1 which it has been unable to determine the functor of the argument. Hence Parma insists that the user declare all predicates which may be invoked by *call*/1. Unless the user also makes an optional entry type declaration for these predicates, it will be assumed they can be called with any argument.

It may be an onerous task to make these declarations but there is some consolation in that it provides possibly useful documentation of the program. The worst aspect is probably that the departure from other Prolog dialects may make the compilation of programs in these dialects with Parma more difficult.

One important use programmers make of *call*/1 is to provide extra meta-predicates. Below is a typical example:

```
write_call(X) :- write(entering(X)), nl, call(X), write(leaving(X)), nl.
```

These meta-predicates tend to be simple and the term which will be passed to *call*/1 is usually an explicit argument to the meta-predicate and hence visible at compile-time. Such a meta-predicate may be used widely in a program and hence the user would have to declare many predicates as being *call*ed.

This problem can be reduced, but not eliminated, by the provision of a a rich set of meta-predicates, thus reducing the need for the user to provide their own. One possibility is that simple meta-predicates could be removed by an initial partial evaluation phase. This has not been provided in Parma, but a partial evaluator is likely to appear in a future version because it can be used for a number of useful transformations. A more pragmatic approach to such simple meta-predicates is to insist that the user use some form of macro facility to implement the meta-predicates. Calls to the meta-predicates would then be expanded in-line before analysis.

Another alternative is to allow users to declare a predicate to be a meta-predicate. This declaration would indicate which of the predicate's arguments are terms which may be passed to *call*/1. This would remove the necessity to declare the predicates passed to *call*/1 by such a predicate. This should be much more convenient for the user. Such declarations have not been provided in Parma and we remain uncertain about the best approach. This is certainly an area deserving further work.

Built-in predicates, such as *assert*/1 and *retract*/1 which modify the program at run-time, also confound analysis. [Deb87a] presents an algorithm for a simple class of programs containing *assert*/1 and *retract*/1. The set of programs to which it can be applied is so restricted that its utility is very limited.

3.1

It is very useful for a compiler to know which predicates can be dynamically modified. This allows static predicates to be treated much more efficiently. For this reason existing compilers require the user to declare any predicate which may be dynamically modified during execution. This is not onerous and provides very useful documentation. Given this precedent there were no qualms in forcing users of Parma to declare predicates which are dynamically modified.

Users can also, optionally, improve the precision of Parma's analysis by declaring the exit type of these predicates and declaring if they are always determinate. Without such declarations Parma must treat the execution of such predicates as unknown and possibly non-determinate. The correctness of such declarations may need to be checked during execution (see Section 3.4.2). Usually *assert*/1 will be used to add facts to the programs but in some cases it will be used to add rules to the program. This poses an additional problem as, like *call*/1, it hides the flow of control. As with *call*/1 the user must declare any predicates called by rules added at run-time.

Parma's analysis algorithm assumes that the value of a predicate on entry is always unifiable with the value of the same argument on exit. This is true for pure Prolog and for most built-in predicates. However some implementations contain built-in predicates which do not meet this criteria. These predicates usually provide some form of destructive assignment, for example SICStus Prolog's *setarg*/3. Such predicates create other problems for Prolog implementations and we feel excluding the support of such predicates is quite acceptable.

3.2 TRANSFORMATION

Parma transforms clauses to a much simpler internal normal form before analysis. This is convenient both for the analysis phase and the subsequent compilation phases. The main benefit to the analysis phase is that it allows the operations of abstract execution to be simplified. This makes it easier to ensure the correctness of the implementation of these operations which is of no small importance in an experimental implementation.

At first sight this transformation is counter-intuitive as it removes somes of the clause structure and makes it more difficult for a human to read the program. However it would be unwise for Parma to rely on clauses being in the typical human written form. Already some applications machine-generate Prolog programs for execution effectively layering a language above Prolog. This seems a promising avenue and these programs should be properly supported.

The normalising transformation removes all unifications from the clause head. The transformed clause head contains only variables none of which occur more than once. Calls in the transformed clause body also contain only variables. These can occur more than once in the call. Normalised clauses contains only simple unifications. These are restricted to three forms: unification of two variables, unification of a variable and a constant and unification of a variable and a structure whose sub-terms are all variables which have not occurred previously in the clause. As an example, here is a clause from the *nreverse* benchmark and its normalised form:

```
nreverse([X|Xs], Zs) :-
        nreverse(Xs, Ys),
        append(Ys, [X], Zs).
```

```
nreverse(V1, Zs) :-
        V1 = [X|Xs],
        nreverse(Xs, Ys),
        V2 = [V3|V4],
        V3 = X,
        V4 = [],
        append(Ys, V2, Zs).
```

Some meta-predicates which in effect provide control structures are removed in the initial transformation. These include '->'/2 which provides an equivalent to if statements and '\+'/1 which provides a form of negation. This is done by adding an internal version of cut named '!!', which cuts only to the surrounding disjunction. Following is an example illustrating this transformation:

```
a :-
        \+ b,
        (c -> d ; e).
```

```
a :-
        ((b, !!, fail) ; true),
        ((c, !!, d) ; e).
```

When designing the normalised form it seemed wiser to err on the side of simplicity even if larger, more verbose transformed clauses resulted. In hindsight the normalised unifications were indeed too simple. The unification of a variable with a list of N items expands into 2N unifications. This expansion slows analysis and compilation significantly. In a re-implementation of Parma we would relax the transformation so there were only two forms of unifications: unification of two variables and unification of a variable and a term which does not contain any variable which has occurred previously. The normalising transformation would attempt to merge unifications. This form would be more compact but would not, we believe, greatly increase the complexity of the handling of unification by the analysis and compilation phases.

3.3 ABSTRACT DOMAIN

The nature of the abstract domain is governed by the demands for information of the subsequent compilation phases. It may be necessary to make the abstract domain more detailed than is required to directly represent the desired final information. Otherwise loss of information during execution may reduce the precision of the final results.

In general, the most desirable piece of information that can be discovered about a variable at a particular program point is that it is always bound or always free. This can allow a unification to be much simplified. It may reveal a clause is determinate allowing choice-point creation code to be removed. It may allow more efficient indexing of a predicate.

It requires only a simple abstract domain to discover this information. Three symbols are necessary indicating free variables (*free*), bound terms (*bound*) and a universal symbol (*unknown*). The inclusion of a fourth symbol for terms containing no free variables (*ground*) yields better information by representing intermediate results more precisely.

Often it can be determined that, at a particular program point, a variable will always have a particular tag. For example, it may be always an integer. This information is useful to the compilation phase as it allows the omission of tag-checking code. For this reason the symbols *atom*, *integer*, *float*, *number* and *constant*, with the obvious meanings, appear in our abstract domain. As will become apparent later, it is useful to also include a symbol, [], for the atom '[]'.

The lattice that these symbols form is depicted in Figure 3.1.

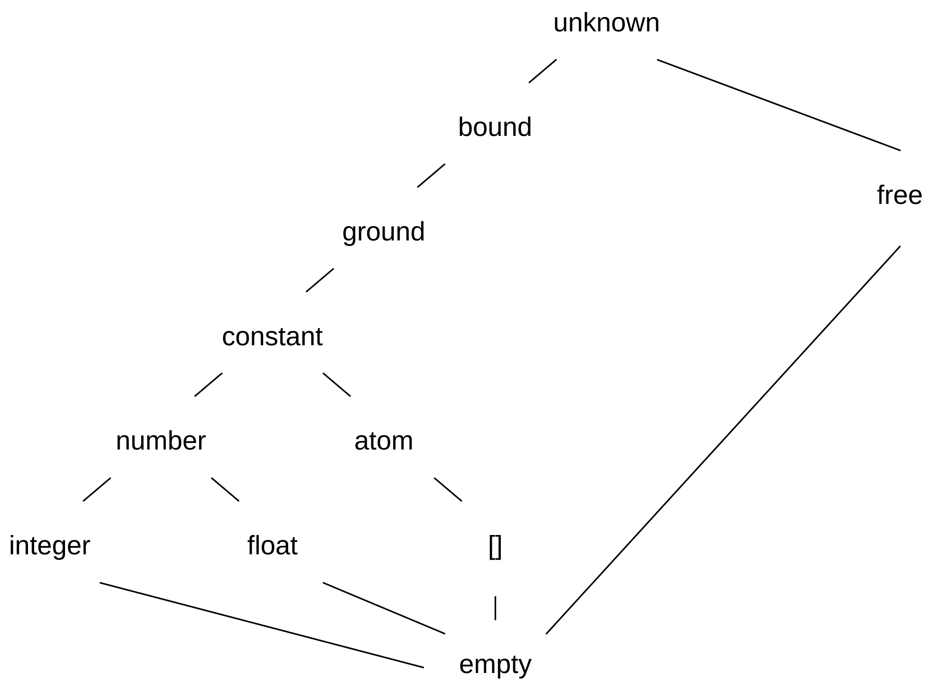


Figure 3.1. Simple Type Lattice

This abstract domain is likely to lose useful information in a program employing structures. There are at least two important cases where this occurs. Firstly, where lists are employed. Secondly, where structures are used to aggregate data analogously to Pascal's *record* data type.

We avoid this loss of information by including a compound symbol in the abstract domain which represents a structure of a particular functor and contains symbols describing the possible arguments of the structure. For example, the symbol *structure*(f(*atom*, *integer*)) represents all structures with functor f/2 and first argument an atom and second argument an integer.

The unrestricted introduction of this compound symbol would make our abstract domain a lattice of infinite depth. Although it may not be essential for the lattice to be of finite depth it certainly makes control of abstract execution much simpler. We retain this property by restricting the nesting of compound symbols to less than a constant bound. A nesting limit of four levels is currently employed in Parma. This limit seems large enough to avoid significant information loss in most programs.

3.3

Many programs implement recursive data types such as lists and binary trees. These cannot be represented precisely in the abstract domain we have so far described. This is unfortunate as the parts of the programs which manipulate these data structures are often those where most execution time is spent and hence the parts where good compilation is most desirable. More ambitious abstract domains have been described which could represent some of these data types[Bru87]. However, as yet, it is not clear that abstract execution can be performed efficiently over such domains. As a primary goal of Parma was to establish the practicality of global analysis we did not attempt a general handling of recursive data types.

Lists pervade Prolog programs and a compiler that handles them poorly will fall well short of optimal performance. As high performance was also a primary goal for Parma we have adopted an ad hoc solution. Parma's abstract domain includes a symbol which can precisely represent lists constructed with the conventional use of structures of functor '.'/2 as list cells. The symbol *list*(*car*, *cdr*) represents the symbols in the infinite sequence:

cdr ’.’(*car*, *cdr*) ’.’(*car*, ’.’(*car*, *cdr*)) . . .

For example, the symbol *list*(*integer*, []) represents precisely all nil-terminated flat lists of integers. The same nesting restriction which applies to *structure* symbols applies to the *list* symbols ensuring that the abstract domain remains a lattice of finite depth.

Earlier versions of Parma omitted the 'cdr' argument of the *list* symbol. It was implicitly assumed to be '[]'. This did not allow the exact representation of difference lists [Ste87]. It was found significant information was lost where difference lists were employed. Difference lists are not prevalent in benchmark programs but skilled Prolog programmers make significant use of them. It can be expected that not only will they appear in real programs but they will appear in precisely those places where efficiency is important. The inclusion of the ’cdr’ argument not only allowed exact handling of difference lists but also slightly simplified abstract execution.

There is further useful information that can be determined about a free variable at a particular program point. It is useful to know if the free variable will never need trailing if bound at this program point. This will often be so and it will allow the omission of the trailing operation. It will be the case if no choice-point can have been created since the free variable was created. The *free* symbol includes a *trail* argument to allow this to be indicated.

The compilation phase can also gain significant advantage if it can determine that a free variable involved in a unification can never be aliased to any other free variable. This allows it to much simplify the code which binds the variable. More generally, it is useless for an abstract domain to have a symbol for free variables but not also have some representation of aliasing information. Without information on aliasing the binding of any free variable potentially binds all other free variables. All information regarding free variables would quickly be lost.

The free variable symbol of Parma's abstract domain includes three values indicating aliasing information: a *must_alias* value, *may_alias* value and a boolean. Any two free variables with the same *must_alias* value must be currently aliased. Similarly free variables with different *may_alias* values will never be aliased at this program point. A third boolean value, *is_aliased*, is included. It is true if the free variable is not aliased to any other free variable. This is redundant, as this could be determined by examining the *may_alias* values of all free variables. It is included to make abstract execution faster. The *unknown* and *bound* symbols also have *may_alias* arguments. This allows more precision to be maintained when unions are formed over the abstract domain.

The combination of aliasing and the *list* compound symbol is awkward because the *car* argument symbol possibly corresponds to multiple sub-terms. There are several possible choices for the meanings of the aliasing information in this context. We have made two simplifying restrictions. The *must_alias* argument of *free* symbols is limited in scope to within the *car* argument. The *may_alias* argument of *free*, *unknown* and *bound* symbols has unlimited scope. As a result there is no way to precisely represent a list of variables which must be aliased to each other. It is also not possible to precisely represent a list of variables which cannot be aliased to each other. We are hopeful that these types will not arise frequently during abstract execution and hence these restrictions will not result in significant loss of precision.

Parma's full abstract domain is, like the simpler subset depicted in Figure 3.1, a lattice of finite depth. A full description of Parma's abstract domain is below:

free(*may_alias*, *must_alias*, *is_aliased*, *trail*, *chain*) - A free variable which must be aliased to the variables indicated by *must_alias*. The only other variables it can be aliased to are indicated by *may_alias*. If *is_aliased* is false it cannot be aliased to any other variables. If *trail* is false the variable does not need trailing when bound. If *chain* is true there is no reference chain to follow when binding the variable.

unknown(*may_alias*) - Any term where any free variables it contains may only be aliased to those indicated by *may_alias*.

bound(*may_alias*) - As **unknown** but cannot be a free variable.

ground - Any term which contains no free variables.

constant, **number**, **float**, **integer**, **atom** and **[]** - Constants and sub-divisions of the constants.

term(*functor*($type_1$, . . . $type_n$)) - any term of form *functor*/*n* whose arguments are in $type_1$, . . . $type_n$, respectively.

list(*car_type*, *cdr_type*) - Any term in *cdr_type* or one or more chained list ('.'/2) cells all with the first argument of each in *car_type* and the second argument of the last in *cdr_type*.

3.4 ABSTRACT EXECUTION

In some operations abstract execution mirrors real execution. The implementation of these operations in Parma also tends to mirror the implementation of the analogous operations in a Prolog interpreter. There are also major divergences from real execution. The most important is in the flow-of-control in abstract execution. There is no backtracking during abstract execution. Instead disjunctions including the clauses of a predicate are executed breadth-first and the union of the results computed. The flow-of-control between predicates is also very different where recursive predicates are involved. The control of abstract execution between predicates is described in the next section. In this section we will describe how Parma implements the steps of abstract execution within a clause. During development it was found that careful implementation of these steps was necessary to reduce the time taken by abstract execution.

The abstract execution of a clause starts with a vector of abstract symbols representing the possible values of each of the arguments of the clause when it is called. It yields a vector of abstract symbols representing the possible values of each of the arguments of the clause when it exits. We will refer to these respectively as the call pattern and the exit pattern. During the abstract execution of a clause a vector is maintained containing the current value, in the abstract domain, of each of the clause's variables. The other part of the execution state is a variable indicating if a choice-point may have been created since execution of the clause began. Conjunctions are just sequentially executed with the execution of each subgoal modifying the execution state appropriately. Disjunctions are also executed sequentially but each disjunct is executed with a copy of the execution state as it was on entry to the disjunction. The result of the disjunction is the union of the execution state results from each of the disjuncts.

The key to the efficient implementation of the low-level operations of abstract execution is the technique used to manage aliasing information. This is particularly so when Prolog is used as the implementation language.

Parma uses Prolog variables to implement the *must_alias* and *may_alias* arguments of the *free*, *unknown* and *bound* symbols . The Prolog variables are unified when it is necessary to make two *may_alias* or *must_alias* values the same because an operation has resulted in these variables, possibly or necessarily, becoming aliased. The built-in '=='/2 can be used to test if two variables may or must be aliased. This representation allows easy propagation of the effects of operations to variables indirectly affected via aliasing.

When an operation changes a variable from *free* to another abstract symbol the effects are propagated by binding the *may_alias* and *must_alias* Prolog variables.

The *must_alias* Prolog variable is bound to the abstract symbol that the variable now has. The *may_alias* Prolog variable is bound to *unknown*. When a *free* symbol is encountered during execution, the *must_alias* argument and then the *may_alias* are inspected. If the *must_alias* argument is bound then the effects of aliasing have produced this value which is used instead of *free*. If the *may_alias* argument is bound then the effects of aliasing mean the symbol *unknown* must be used.

A Prolog variable has also been used by others to represent aliasing during abstract interpretation [War88]. This technique inspires two remarks. It is perhaps a sign of a weakness in Prolog that a somewhat non-intuitive representation is needed for a data structure that would be straightforward to implement in a language such as C. It could also be an indication that there is a natural advantage in implementing an abstract interpreter for a language in the same language.

3.4.1 Unifications

Unifications have been simplified by the initial transformation into three forms. A single piece of code forms the heart of the implementation of each of the forms. This code takes two abstract symbols and yields the result of their unification. The code is not complex, basically just a large case-by-case analysis. Compound symbols can be handled recursively. Aliasing is handled as described above.

The unification code is simplified by delaying some operations until a call or the finish of the clause is reached. A transformation is applied to these abstract symbols to reduce them to a normal form. The unification code ignores the nesting bound for compound symbols. These are pruned to the nesting limit. This is done by replacing *list* or *term* symbols which are too deeply nested with a *ground* or *bound* symbol. The normalising transformation translates any *term* symbol with functor '.'/2 to a *list* symbol. For example:

$$\boxed{term('.'(integer,\ []))} \longrightarrow \boxed{list(integer,\ [])}$$

The implicit assumption is that programmers use terms with functor '.'/2 only to construct lists. Some precision may be lost in the analysis if this is not the case but the results will remain correct.

The normalising transformation also simplifies any instance of a list symbol nested within another *list* as its *cdr* argument. This is replaced with a single *list* symbol. The *car* of the new list symbol is the union of the *car* arguments of the previous list symbols. The *cdr* of the new list symbol is *cdr* of the nested *list* symbol. For example:

list(*integer*, *list*(*atom*, [])) → *list*(*constant*, [])

The final function the normalising transformation performs is the substitution of the appropriate abstract symbol for *free* symbols which have been affected by aliasing.

There are two other important operations performed on abstract symbols. Firstly, a simplified version of the code which unifies two abstract symbols is used to obtain the intersection of two abstract symbols. Another piece of code produces the union of two abstract symbols. More formally, it yields the abstract symbol which is the least upper bound of the two abstract symbols. As with the unification code it is a matter of case-by-case analysis with compound symbols handled by recursive calls. Aside from four operations described above (unification, intersection, union and normalising transformation) the abstract execution code is largely independent of the particular abstract domain.

3.4.2 Calls

Calls have been simplified by the initial transformation so they contain only variables. The first step in the execution of a call is to obtain the current abstract symbols of these variables. The next step in execution of a call to a program predicate is to pass the call pattern to the abstract control code. The abstract control code will take the necessary steps to compute the exit pattern that will result from this call pattern. This will be described in the next section. Predicates which have been declared to be dynamically modified are an exception. In this case this exit pattern is assumed to be unknown.

The user may optionally supply entry or exit declarations for a predicate. These declarations supply for each predicate argument the possible entry or exit values for that argument. The syntax of these declarations is exactly that which we have used in describing the abstract domain except that aliasing and trailing information is omitted. Although aliasing and trailing information could be useful it was thought too likely an avenue for introduction of errors. The abstract symbols in entry declarations are intersected with those in the call pattern to make it possibly more precise before it is passed to the abstract control code. An empty intersection is likely to indicate an error and the user is warned. Unfortunately an incorrect declaration is not guaranteed to produce an empty intersection. Similarly the exit pattern from the call is intersected with the exit declaration.

Currently an incorrect entry or exit declaration may result in unpredictable behaviour during execution. The likely, but unfortunately not guaranteed, result will be an address exception. This will not be acceptable to many users. A production version of Parma would insert sufficient code to produce a useful exception if a user declaration is violated during execution. This would reduce the benefits from such declarations so a user would also be able to specify that their declarations were not to be checked.

Hopefully mode declarations will not often be used. They are primarily intended for users desperate for any increment in program performance. Even if Parma's analysis phase is performing poorly on a particular program it is unlikely that declarations will be needed for more than a fraction of the program's predicates. Our experience so far is that usually there are only a small number of sources of imprecision but their effects may propagate through the entire program. Optionally, after analysis, Parma will indicate predicates which are likely candidates for entry and/or exit declarations. This is based on a simple metric which counts occurrences of *unknown* and *bound* symbols. It is not clear how useful it is in practice.

Calls to built-in predicates constitute roughly 50% of all calls [Tou87]. It was found in the development of Parma that useful gains in precision could be obtained by improving the handling of calls to built-in predicates. For this purpose Parma classifies built-in predicates into several categories:

- Predicates that neither instantiate their arguments nor yield any information about them e.g. *write*/1. Calls to these can be ignored.

- Predicates that don't instantiate their arguments but do yield information about them e.g. *atomic*/1. Parma retrieves their exit pattern from a table. This is intersected with the call pattern to yield the new abstract values of the argument variable.

- Predicates that instantiate some of their arguments e.g. *read*/1. Parma retrieves their exit pattern from a table. This is unified with the call pattern to yield the new abstract values of the argument variable.

- Predicates which are handled poorly by table look-up and are used frequently. These include *arg*/3, '=..'/2 and *is*/2. Each of these is handled by special-purpose code.

3.4.3 Trailing Information

The abstract execution code must manage some extra information to discover the trailing status of variables. An extra element must be added to the execution state indicating whether a choice-point may have been created since the execution of the predicate began. This requires the abstract control code when returning the exit pattern calculated for a call to also return a boolean value indicating whether the execution of the call may have left a choice-point.

It should be noted that predicates which by some definitions would be determinate must leave choice-points. For example the following predicate always succeeds exactly once but must leave a choice-point:

```
a :- write(hello).
a :- write(world), fail.
```

We regard as non-determinate any call which when re-done may succeed or may fail but execute a call to another program predicate or to a built-in predicate which may have side-effects.

During the abstract execution of a clause Parma does not change the trailing status of local variables when a choice-point is created by a disjunction or a call to another predicate. Parma takes this approach because a choice-point is often removed by a cut and it would then be difficult to change back the trailing status of local variables. When a call pattern is passed to the abstract control code Parma sets the trailing argument of any *free* symbols to true if a choice-point may have been created since execution of the clause began. This is overly conservative because the first occurrence of the variable involved may be subsequent to the program point where the choice-point may have been created. Parma greatly improves its precision by the pragmatic measure of checking when constructing the call pattern for variables which have not occurred before. The *free* symbol associated with such variables is not marked as needing trailing.

A more general approach would involve some form of time-stamping which would allow the relative creation time of choice-points and free variables to be determined. This approach was not used, despite its elegance, because it was thought it would be expensive and yield little more information.

Parma propagates trailing information between *free* symbols using the same method used to propagate the effects of aliasing. A Prolog variable is used to represent the *trail* argument of a *free* symbol which, currently, does not need trailing if bound. This Prolog variable will be unified with the *trail* arguments of any other *free* symbols to which this *free* symbol may possibly be aliased. This essentially duplicates the *may_alias* argument. The *trail* argument of the *free* symbol which may need trailing if bound will be the constant *true*. This allows the code for abstract unification to easily propagate changes in trailing status.

3.5 ABSTRACT CONTROL

This section describes how Parma manages the flow-of-control between predicates during abstract execution. This management can be nicely divorced from the abstract execution that occurs within a predicate. It is only invoked when a call to another program predicate is executed.

We will assume in the following discussion that there is only one entry-point to the program, the predicate *main*/0. In practice there may be multiple entry-points including the predicates declared by the user to be invoked by *call*/1. It is a simple generalisation to handle multiple entry-points.

In the absence of recursion the management of abstract control is trivial: at every call execution can be passed to the called predicate. The use of such an approach in the presence of recursive predicates will quickly result in abstract execution looping. This, of course, is a problem seen in many forms through computer science. A solution that also appears in many forms can be used. A fixpoint can be found by iterative refinement. This fixpoint yields the desired information.

Basically, when a call pattern recurs during an abstract execution the results, if any, from past executions of that call pattern are used. We can, by repeated execution, obtain a fixpoint and in the process perform all possible executions of the program. This requires a table to record the results of past execution. The table entries consist of a <Call, Exit, Status> triple where Call is a particular call pattern and Exit is the exit pattern which resulted from last execution of that call pattern. No call pattern occurs twice in the table and so the Call field can be used to index the table. The Status variable is used to indicate if the call pattern has occurred in the current execution.

The simple algorithm in Figure 3.2 is sufficient to manage abstract control using such a table. When a call is met during the abstract execution it is passed to the *manage_call* routine.

Although this algorithm's simplicity is a joy, its efficiency unfortunately is not. In many cases it performs considerable amounts of unnecessary work and as a result was often unacceptably slow. This is because fixpoints are reached at different rates through the program. Much unnecessary execution of parts of the program which have already reached a fixpoint can occur. Clearly this problem had to be remedied.

We found two modifications which in combination were useful. It was found fixpoints were reached faster if the abstract execution code updated the call-exit table to reflect the intermediate results obtained as each clause was executed. This muddied the clean interface between abstract execution and abstract control code but it was worthwhile if the abstract execution code also tried to execute the non-recursive clauses of a predicate before the recursive clauses. The advantage is that by the time a recursive call is met the call-exit table will contain the information computed from the non-recursive cases. The improvement produced was not sufficient to revive the algorithm in Figure 3.2. However, both of these modifications were retained in later versions of the abstract control algorithms.

The next approach we employed was to discover the recursions in the programs. An initial phase extracted the program's call graph. The strongly-connected components of this graph were computed. This reveals non-recursive predicates and sorts other

```
start_execution()
{
        Table = {}
        repeat
            set all status fields in Table to not_done
            Fixpoint_Reached = true
            manage_call(main)
        until Fixpoint_Reached == false
}

manage_call(Call)
{
        if (Call not in Table)
            add entry <Call, fail, not_done> to Table
        if (Table[Call].status != not_done)
            return Table[Call].exit
        else {
            Table[Call].status = started
            New_Exit = abstract_execute(Call)
            if (Table[Call].exit != New_Exit) {
                Table[Call].exit = New_Exit
                Fixpoint_Reached = false
            }
            return New_Exit
        }
}
```

Figure 3.2. Abstract Control - #1

predicates into the sets of predicates with which they are mutually recursive. The algorithm of Figure 3.2 was modified to utilise this information. Call patterns for non-recursive predicates were executed only once and execution only occurred in sets of mutually recursive predicates until a fixpoint was reached.

This approach produced considerable improvement and was satisfactory for many programs. One disadvantage was that it was no longer possible to exploit the technique described in Section 3.1 where abstract execution continues through the built-in predicate *call*/1 if the functor of the *call*ed predicate is revealed by abstract execution. This is because a new recursion could be exposed. Abstract execution was still expensive for some programs so a more precise algorithm was developed in an attempt to improve performance.

It is only useful re-executing a call-pattern if one of the call-exit pattern pairs used in computing its exit pattern has changed. The algorithm in Figure 3.3 exploits this by noting dependencies between call-exit pattern pairs. This is done by maintaining a

```
start_execution()
{
        Table = {}
        manage_call(main)
}

manage_call(Call)
{
        if (Call not in Table)
            add entry <Call, fail, not_done> to Table
        if (Table[Call].status == done)
            return Table[Call].exit
        else if (Table[Call].status == not_done)
        {
            Table[Call].status = started
            while true {
                push <Call, {}> on Stack
                New_Exit = abstract_execute(Call)
                pop <_, Dependencies> from stack
                if (Table[Call].exit == New_Exit)
                    break;
                else {
                    Table[Call].exit = New_Exit
                    for D in Dependencies
                        Table[D].status = not_done
                }
            }
            Table[Call].status = done
            return New_Exit
        } else {
            find the entry for Call on Stack and add any
            more recent entries to its dependencies field
            return Table[Call].exit
        }
}
```

Figure 3.3. Abstract Control - #2

stack of call patterns being currently executed. When a call pattern recurs in the stack then the call patterns that have been stacked since the first instance depend on this call-exit pattern pair. This is noted. If on completion of the execution of the initial instance the exit pattern is changed these dependent call patterns must be re-executed.

Although this algorithm is conceptually more satisfying it only resulted in significant performance improvements in some cases. The execution time of some programs remained unacceptable. The cause appeared to be large sets of mutually recursive predicates. It is possible some refinement of the control algorithm could have produced acceptable execution time, however, we could not produce such a refinement. At this point pragmatism set in and we decided, as is always possible when implementing abstract execution, to trade reduced precision for faster execution.

When gathering information about a particular predicate, we are not interested in the individual call patterns that occurred during abstract execution. Rather we are interested in the union of the call patterns. This suggests a modification to our control algorithm. When a new call pattern is encountered the union of that call pattern is formed with any previous call patterns for the predicate. It is the call pattern resulting from this union that is executed. As a consequence we maintain only one call-exit pair for each predicate. The modified algorithm is shown in Figure 3.4.

This approach is safe but precision is lost. Consider the abstract execution of this program fragment:

```
a :- b(1, A), b(a, B), c(B).

b(X, Y) :- X = Y.
```

The first call pattern for the predicate *b*/2 during the abstract execution of this fragment will be b(*integer*, *free*). The resulting exit pattern is b(*integer*, *integer*). The next call pattern for the same predicate is b(*atom*, *free*). The union of this call pattern with the previous call pattern is b(*constant*, *free*). This call pattern is executed and yields the exit pattern b(*constant*, *constant*). The predicate *c*/1 will be executed with call pattern c(*constant*). The previous abstract control algorithms would have produced the more precise call pattern c(*integer*).

The performance improvement resulting from this algorithm was all that we desired. It yielded good execution times even for programs with very large sets of mutually recursive predicates. However the cost is significant loss of precision in the analysis of some programs. Although we decided to accept this loss of precision there are avenues by which it could be reduced. It seems to be only large sets of mutually recursive predicates that require this approach. An alternative would be a hybrid of the algorithms in Figures 3.4 and 3.3. This would only form the union of call patterns when the predicate involved belongs to a set of mutually recursive

```
manage_call(Actual_Call)
{
        F = functor of Actual_Call
        if (F not in Table)
            add entry <F, fail, not_done> to Table
        else if (Table[F].call subsumes Actual_Call)
            Call = Table[F].call
        else {
            Call = union(Actual_Call, Table[F].call)
            Table[F].status = not_done
        }
        if (Table[F].status == done)
            return Table[F].exit
        else if (Table[F].status == not_done)
        {
            Table[F].status = started
            while true {
                push <Call, {}> on Stack
                New_Exit = abstract_execute(Call)
                pop <_, Dependencies> from stack
                if (Table[F].status == done)
                    return Table[F].exit
                if (Table[F].exit == New_Exit) {
                    Table[F].status = done
                    return New_Exit
                } else {
                    Table[F].exit = New_Exit
                    for D in Dependencies
                        Table[D].status = not_done
                }
            }
        } else {
            find the entry for Call on Stack and add any more
            recent calls to its dependencies field
            return Table[F].exit
        }
}
```

Figure 3.4. Abstract Control - #3

predicates. This may reduce the loss of precision but still yield acceptable execution times.

During abstract execution the only information is that recorded in the call-exit table. The output from the analysis phase is the clauses with the variables at each program point suitably annotated. The annotation of each clause is done by a modified version of the abstract execution code which uses the information in the call-exit table.

3.5

We have not attempted a worst-case complexity analysis of Parma's global analysis algorithms but we have attempted an informal assessment of the average case. The execution time for global analysis of programs ranging from 10 to 1100 lines in size have given no indication that, on average, *Parma*'s analysis time is any worse than linear in the program size. Some of these execution times can be found in Chapter 5. These results leave us confident that analysis of very large programs by Parma is feasible.

There is a very interesting implementation technique which we have not exploited. In [War88] it is suggested that the target program be transformed before abstract execution. This transformation produces a program which can be executed to perform the abstract execution. This they logically term *abstract compilation*. It can be thought of as partial evaluation of the abstract execution code with respect to the program to be executed. The particular level of transformation suggested in [War88] would be of little benefit to Parma. Using profiling information we estimate the gains to be less than 5%. Their abstract compilation is too shallow. More aggressive abstract compilation could produce useful gains for Parma.

Our approach in the development of Parma's analysis phase may seem somewhat patchy and ad hoc. This is because the analysis phase is not an end in itself. The entirety of our work could have been devoted to exploring Prolog analysis algorithms. This was not our objective. We wanted to demonstrate that global analysis was feasible and to show that the information that it yielded could be of substantial benefit to a compiler. The results in Chapter 5 demonstrate that we have done both.

4 COMPILATION

This chapter describes Parma's compilation stage translation of Prolog clauses to MIPS assembly language. Although the input to this stage is Prolog clauses two important transformations have already been made. Firstly, the Prolog clauses have been transformed to an internal normal form (Section 3.2). This much simplifies the task of compiling unifications. Secondly, and more importantly, the clauses have been annotated with the information discovered by the global analysis phase. Although the compilation stage can operate without this information, the heart of the main compilation phase is devoted to taking advantage of this information.

The translation from Prolog clauses to MIPS assembly language occurs in ten separate phases. Figure 4.1. shows the Parma's first four phases. The last two of these phases form the first part of the compilation stage. The first phase of the compilation stage performs source-level transformations. This is followed by the most important phase which translates the Prolog clauses to low-level instructions of an intermediate language. This is where most of the work of compilation occurs.

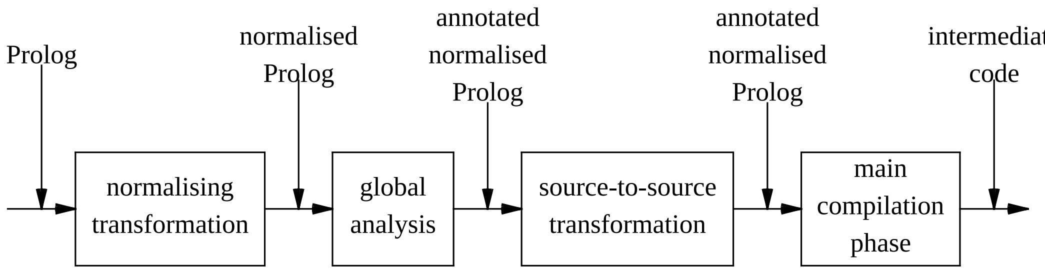


Figure 4.1. Parma's Phases: Prolog to Intermediate Code

The translation from Parma's intermediate language to MIPS assembly language involves eight phases. The operations involved in these are much less Prolog-specific than the operations in the earlier phases. They include well-worn techniques found in compilers for many languages, such as common sub-expression elimination. These phases are described in Section 4.8.

Parma's compilation phase is implemented by roughly 6,000 lines of SICStus Prolog. The execution times of the algorithms employed in this stage are of little interest. The algorithms, including construction of indexing code, are linear in the size of the code produced or, at least, could be with more careful implementation. Unlike the analysis stage, reducing execution time was not a high priority in the implementation of the compilation stage. As a result Parma's overall execution time, at roughly 2-4 lines compiled per second, is slower than most users would like. It is clear that a more careful implementation of Parma's compilation stage would make Parma's execution speed similar to that of compilers for other languages. We will concentrate on a functional description of compilation rather than a precise description of the implementation details.

We will first describe the source-level transformations of the first compilation phase. Next, the intermediate language which is the target of the second compilation phase is detailed. This is followed by a bottom-up description of the important second compilation phase. This occupies most of the chapter. Descriptions of the intermediate code transformations follow. The chapter finishes with an example illustrating the phases of compilation.

It should be noted that predicates which are dynamically modified by built-in predicates such as *assert*/1 and *retract*/1 are excluded from this discussion of compilation. They are treated separately and much less efficiently than static predicates. They are compiled to a stub which calls an interpreter to perform the actual execution.

4.1 SOURCE-LEVEL TRANSFORMATIONS

It is not difficult to contrive source-level transformations which may improve a Prolog program. There are a great variety of such transformations. The problem is determining that the application of a transformation is both safe and will result in improved performance. As a result the only source-level transformations Parma undertakes is a very restricted reordering of conjunctions.

Reordering a conjunction can result in significant improvement in efficiency. It is however, difficult for a Prolog implementation to determine if a particular reordering leaves the program's semantics unchanged. A reordering may change, directly or indirectly, the sequence of calls to built-in predicates with side-effects, such as *write/1* or *assert/1*. This may change the meaning of the program. This is not insurmountable. Simple global analysis can determine that the execution of a particular call will be free of side-effects. This would allow many useful reorderings to be demonstrated as safe in respect of side-effects. However there are other problems.

Reordering a conjunction may result in changes to the instantiation of arguments to calls in the conjunction. This obviously may be unsafe if the program contains cuts, calls to non-logical predicates such as *var*/1, or calls to built-in predicates which require particular argument instantiations. More insidiously, it may be unsafe even in pure Prolog because it can result in non-termination. Worse still, non-termination remains a problem even when a reordering does not change the argument instantiations of any calls. Consider this program:

```
main :- a(1).
a(X) :- b(X), c(X).
b(2).
c(1) :- c(1).
c(2).
```

Swapping the calls to *b* and *c* does not change any instantiations but does change the program's meaning. This is a contrived example but this will also occur, if rarely, in real programs.

In light of the above problems, the only reordering of conjunctions that Parma attempts is the moving of unifications and calls to some built-in predicates and then only in such a way as to not change the sequence of calls or the instantiation of any call, some built-in predicates excepted, during program execution. It may well be appropriate for a compiler, in future, to be more ambitious but as it is difficult and as it is within the reach of the programmer to reorder conjunctions, Parma does not attempt more at present.

The aim of moving unifications within a conjunction is to reduce the work done in the cases when the conjunction fails. This is done by making failure occur as soon as possible during the execution of the conjunction. This may have the secondary benefit of reducing the information needed to be stored and retrieved from a shallow choice-point. Parma's algorithm for generation of indexing code relies on the reordering of the conjunction done at this stage.

There are two types of subgoal of particular interest. The first we call assignments. These are unifications with a variable known to be unbound. Such unifications cannot fail. We include in this category calls to built-in predicates which have no side-effects and which cannot fail given the information known about their argument instantiation. This includes some calls to *is*/2, *arg*/3, *functor*/3 and '=..'/2. The second interesting category of subgoals we call tests. These are unifications of a constant or a functor with a variable known to be bound. We also include in this category calls to built-in predicates which may fail, do not bind any variables and are inexpensive. Important examples are calls to type-checking predicates such as *var*/1 and *atom*/1 and calls to arithmetic comparison predicates such as '=='/2 and '>='/2.

Starting at the bottom of the conjunction, assignments are moved down the conjunction as far as possible. They are not moved over any subgoal (simple or complex) containing a variable which may be bound by the assignment. Assignments are not moved past a call which is the last element of a conjunction where this would imperil the tail recursion improvement described in Section 4.7. It is not guaranteed that demoting an assignment will not produce worse code. Consider, for example, the following clause.

```
a(X, Y) :- Z = f(X, Y), b, c(Z).
```

Demoting the assignment to *Z* past the call to *b* will produce slower code if *b* fails less than 16% of the time because two variables, *X* and *Y* instead of one, *Z* must be stored to survive the call to *b*. Fortunately, this circumstance will be uncommon in real programs. Parma, which makes no assessment of the likelihood of a call failing, always demotes assignments past calls where possible.

Starting at the top of a conjunction Parma promotes tests upward. Tests are not promoted over calls, complex subgoals or cuts. They are also not promoted over any subgoal which may bind any of the variables in the test. A test is promoted over another test only if it contains a variable numbered less than any variable in that test. This last restriction is not to produce faster code but rather to yield a test ordering that produces good results from the algorithm that is later used to generate indexing code.

These restrictions also ensure that the execution of a subgoal which formerly failed can not be transformed to a subgoal which loops forever because of the presence of cyclic terms.
As a secondary consideration, the tests left in a conjunction after the indexing code is produced could again be reordered to improve efficiency. The efficiency of an ordering depends on the costs of executing each test and the frequency with which the test fails. Determining the optimal ordering is NP-hard [Smi85]. Simple heuristics such as ordering the tests on the cost divided by the failure frequency can produce close to optimal results. Parma, with no information on the frequency of failures, makes no attempts to do such ordering. It does attempt to return such tests to the original order in case this order was deliberately chosen by the user for efficiency.

Promoting a test over an assignment will never result in worse performance. Promoting tests over other subgoals may produce slower code in some cases. However in many important and common cases it produces significantly faster code.

4.2 INTERMEDIATE LANGUAGE

Employment of an intermediate language in a compiler serves two important purposes. First, it can form a more convenient representation for improving transformations than either the source language or the target language. Secondly, it allows much of the target machine-dependent part of the compiler to be isolated in a final code generation phase which makes changing the target of a compiler easier.

The first consideration was the primary factor in the choice of Parma's intermediate language. The intermediate language is basically 3-address code [Aho86] reduced to load-store form. Only two instructions, *load* and *store*, reference memory. The other instructions operate only on registers and immediate values. It could therefore be called one-address code! This representation is well suited to some of the improving transformations that must be applied to produce good MIPS code. At least one of these transformations which manipulates load and store instruction offsets cannot be applied to higher level representations such as 3-address code.

The simplified form of clauses, described in Section 3.2, although still valid Prolog, could also be considered an intermediate language. It is much more restricted than full Prolog and it has improving transformations applied to it. Its presence for these high-level transformations facilitates the choice of a very low-level load-store intermediate language well suited to subsequent transformations. Considering the large semantic gap between Prolog and conventional machine instructions, it is not surprising that if good code is to be produced the transition needs to be made in several steps.

It is simple to translate Parma's load/store intermediate language to MIPS instructions. Each intermediate instruction has an obvious translation to one or two MIPS instructions. Portability was not one of Parma's design goals. It is a very different task to translate intermediate language instructions to the instruction sets of other machines. A code generator for a CISC such as the VAX would need to translate several intermediate instructions into one VAX instruction to produce the most compact VAX code possible. It may not be necessary to exploit all of the (bloated) VAX instruction set to obtain fast code. One of the inspirations for a seminal RISC, the IBM 801, was that a compiler for the IBM 370, a CISC, could often obtain faster code by exploiting only the simpler instructions and addressing modes of the 370 instruction set.

The very regular nature of the intermediate language should mean construction of a code generator for any machine is feasible. It is not clear whether such a code generator can produce code of sufficient quality. If not, another higher-level intermediate language will need to be introduced into Parma to facilitate retargetting.

Parma's load/store intermediate language is described in Figure 4.2. It has a number of named registers: *zero, pc, successp, stackp, globalp, envp, trailp, choicep and call(1) ... call(8)* and an infinite number of general purpose registers. In the last phases of compilation the register allocator assigns all these to the 28 MIPS registers available to user programs.

Three of the named registers have special meanings within the intermediate language. The program counter - *pc* of course contains the address of the current instruction. The link register - *successp* is where the *call* instruction places the return address. The zero register - *zero* always contains the value zero. The MIPS architecture has a zero register but it is only necessary for the code generator to know this. The zero register was promoted to the intermediate language, not to ease translation, but to simplify the intermediate language. It removes the need for a load immediate

Instruction	Description
load R_1, R_2[O] store R_1, R_2[O]	R_1 := memory[R_2 + O] memory[R_2 + O] := R_1
Op R_1, R_2, R_3 Op R_1, R_2, I	R_1 := R_2 Op R_3 R_1 := R_2 Op I Op ∈ {+, -, *, /, mod, |, &, ^, <<, >>}
if R_1 Comp R_2 *Label* goto R_1 call *Label*	if condition PC := *Label* Comp ∈ {==, !=, >, >=, <, =<} PC := R_1 SUCCESSP := PC + 4, PC := *Label*

R_i is a register. I is an integer or a label. O is an integer.

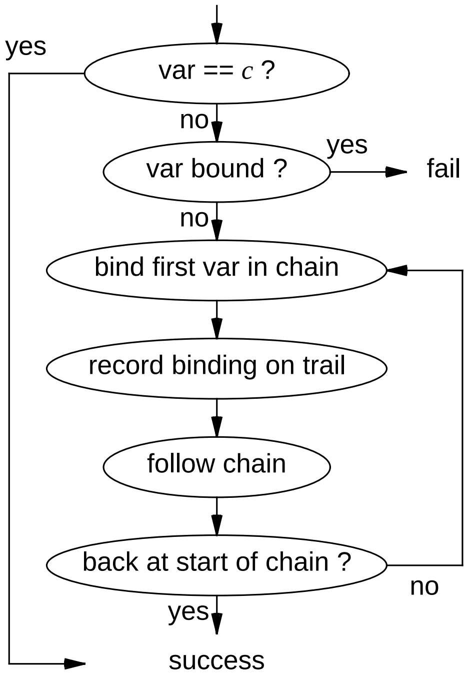
Figure 4.2. Intermediate Language Description

instruction as this can be performed by adding the immediate value to the zero register. This may sound clumsy but it actually simplifies some intermediate language transformations.

4.3 COMPILING UNIFICATIONS

Unification is the fundamental operation of Prolog execution. Its efficient implementation is vital so we will start our description of the main compilation phase at the bottom with details of the compilation of unifications. Recall that the transformation of clauses to the internal normal form has reduced the unifications to three simple types: the unification of a variable and a constant, the unification of a variable and a structure and the unification of a variable and a variable. The code produced for a unification depends heavily on the information global analysis yields on the variables involved. We will not enumerate all the cases that can be detected or the code that will be produced in each. Descriptions of several of the important cases should be sufficient to reveal the nature of the compilation of unifications. We will assume that the relevant variables are already in registers. The management of variables will be discussed latter.

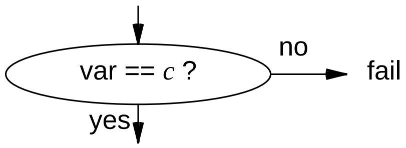
Figure 4.3 gives a code schema and the actual code, transliterated into pseudo-C, that Parma would produce to unify a variable in register *v* with a constant *c* if it has no information from global analysis regarding the variable.




```
     t1 = c
     if v == t1 goto L2
     t2 = v & tag_mask
     if t2 != ref_tag fail
     t2 = v
L1:  t3 = *t2
     *t2 = t1
     trailp[0] = t2
     trailp[4] = t3
     trailp = trailp + 8
     t2 = t3
     if t2 != v goto L1
L2:
```

Figure 4.3. Unifying a Variable with a Constant

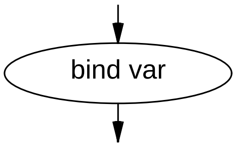
This code can be much improved by information from global analysis. As Figure 4.4 shows, the code required if the variable is known to be bound is reduced to a simple word comparison which is a great improvement.




```
     t1 = c
     if v == t1 fail
```

Figure 4.4. Unifying a Bound Variable with a Constant

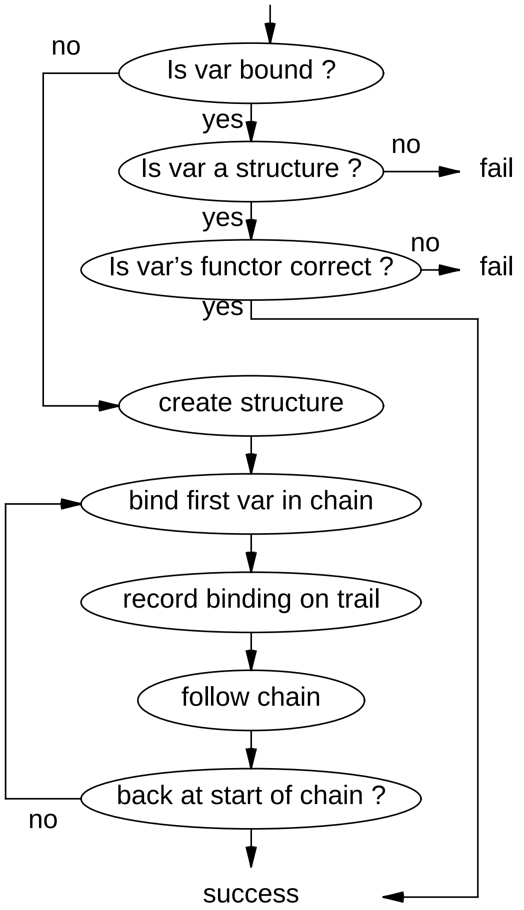
A similar large improvement can be obtained in another common case. The code can be reduced to a simple store if the variable is known to be unbound, unchained and not to need trailing as can be seen in Figure 4.5.



$$t_1 = c$$
$$*v = t_1$$

Figure 4.5. Unifying an Unbound Unchained Variable with a constant

Unification of a variable and a structure is similar but a little more complex to compile than unification of a variable and constant. Recall from Section 3.2 that a structure in this context has as sub-terms only variables which have not previously occurred. As a result the code for the unification is concerned only with the functor of the structure. The code to unify a structure of functor *a*/2, with a variable of unknown type (in register *v*) is shown in Figure 4.6.




```
    t1 = v & tag_mask
    if t1 == ref_tag goto L1
    if t1 == structure_tag fail
    t2 = v[functor_offset]
    if t2 == index(f, 2) goto L3
    goto fail
L1: t3 = globalp + tag_correct
    t4 = globalp + 4
    *t4 = t4
    t5 = globalp + 4
    *t5 = t5
    t6 = v
L2: t7 = *t6
    *t6 = t3
    trailp[0] = t6
    trailp[4] = t7
    trailp = trailp + 8
    t6 = t7
    if t6 != v goto L2
```

Figure 4.6. Unifying a Variable with a Structure

Again the information from global analysis may allow Parma to omit much of this code. Indeed, if the variable is known to be a structure with the appropriate functor, Parma will omit all the code. Parma will omit the initialisation of the structure arguments (to unbound) if subsequent code will store values in these variables without examining them. This will often be the case as such variables usually either occur in a subsequent unification or are passed in call as an uninitialised parameter (Section 4.5.1). The code to unify a variable with a list cell is slightly different; as there is no functor to check, checking the tag is sufficient.

The unification of a variable of unknown type with a structure gains information which may be usefully employed in subsequent code. If the variable is unbound then the argument variables of the structure will necessarily be unbound at their next occurrence. For example consider this clause:

a(X) :- X = a(Y), Y = b.

If the compiler cannot determine the type of *X* then it cannot determine the type of *Y* and must produce the general code for the second unification. However, if during execution of the first unification *X* is found to be unbound, then *Y* will be unbound in the second unification. This information could allow the second unification to be performed more efficiently. The WAM has a mode bit which is used to pass information between unifications. Parma could employ a similar approach, using a register to pass information to the subsequent unification. Setting and examining this register would take several instructions and it is hence not the most efficient approach.

As can be seen in Figure 4.6, the code for the first unification has a conditional branch choosing either the "bound" code or the "unbound" code. The code for the second unification can be duplicated and moved backwards into each arm of the branch. This makes the unbound case execute much faster. This allows the information from the first unification to be exploited by simplifying the code for the second unification in the unbound branch greatly.

The unification of two variables of unknown type is a complex operation requiring roughly 100 MIPS instructions to implement. Including this much code for each such unification would produce an unacceptable increase in code size so Parma implements the general unifier as a library routine and calls it where needed. The cost of this library routine dominates the execution of some programs so it was careful tuned to improve its execution time.

Knowledge of the type of the two variables can much simplify their unification. This can reduce the number of instructions needed to implement the unification sufficiently that the code can be included in-line. This yields further savings. For example, if one variable is known to be a constant and the other variable is known to be bound, then unification can be reduced to a comparison for equality which requires only one instruction if both variables are already in registers. Other important unification cases which allow Parma to produce in-line code are where a variable known to be free is unified with a variable known to be bound and when two variables, both known to be free, are unified. Unfortunately, knowing both variables are bound is not sufficient to allow Parma to produce in-line code but it does allow some savings as a different entry-point to the general unifier can be used, avoiding the execution of several instructions.

The handling of the case where both variables are known to be structures is less straightforward. In some cases, the code needed will be small enough that including the code in-line is desirable. In other cases, the code size will be too large and calling of an out-of-line routine will be desirable. Parma first constructs the in-line code for the unification. If this code is of acceptable size it is used otherwise a call to the general unifier is used. Some execution is saved by using a special entry-point in the general unifier. As the arity of the structures is known at compile-time it is loaded explicitly into a register rather than being determined at run-time. All the above assumes the variables are known to be structures with the same functor. If this is not the case the code needed is simple, a branch to the failure code. This is presumably rare in real programs.

A structure-sharing representation allows much of the work of term construction to be done at compile-time. Terms are represented by a tuple of two pointers usually referred to as a *molecule*. One pointer refers to the term's *skeleton*, which is the term but with variables replaced by indices into a vector containing the values of the variables in this instance of the term. The second part of the molecule is a pointer to the vector of variable values. The term skeleton can be constructed at compile-time as a static data structure but the variable vector must be allocated at run-time. Structure-sharing can allow a program containing large terms with few variables to obtain large benefits because much of the work of term construction is done once, at compile-time, rather than many times during execution. Unfortunately the molecule representation makes access slower and, for many programs, these costs outweigh the benefits.

Parma exploits the benefits of structure-sharing in a degenerate case. A molecule is not needed to represent a term containing no variables as obviously no variable vector is necessary. Parma exploits this by constructing ground terms occurring in the program at compile-time in a static memory area. Where the term is needed at run-time a pointer to the static version is used instead of constructing the term. This offers considerable time and space savings if large terms are involved. Further economy is possible if the same ground term occurs multiple times in the program as only one static copy is necessary. This approach entails none of the costs of structure-sharing. The question is how much of the benefits of structure-sharing does it obtain. It certainly produces useful improvements to the code for benchmarks. This is especially so for small benchmarks. This is because small benchmarks often have relatively large data sets explicit in the program as ground terms. Some user programs will have similar characteristics and obtain similar benefits. More user programs will contain fewer ground terms but should still obtain some benefit perhaps more in code size than in time.

More benefits will be obtained if the programmer ensures that as many terms as possible in the program are ground. In some cases, this can be done by changing the data representation to move all the ground parts of a term into a sub-term. Parma cannot construct at compile-time any of the terms in the predicate *fact*/1 on the left of Figure 4.7 because of the presence of a variable (*V*) in each.

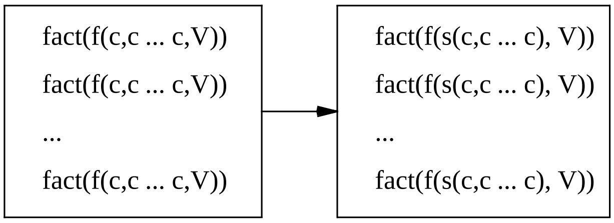


Figure 4.7. Transforming Terms to Create Ground Sub-Terms

It is possible that changes to other parts of the program would allow the the predicate *fact*/1 to be transformed to the form on the right. This predicate does contain ground terms and Parma can compile it much more efficiently, assuming *fact*/1 is going to be called with a free variable as its argument. The work involved and the possible reduction of program clarity will presumably mean programmers will be interested in this transformation only if large terms and hence large benefits are involved. Ideally, the transformation would be made automatically. This would be difficult to implement but the framework involved for such sophisticated source-to-source transformations should also allow a number of other useful improving transformations.

Parma's use of static copies of ground terms relies on it not being possible for any part of the program to destroy these terms. This is a safe assumption in pure Prolog but not in some Prolog implementations. For example, SICStus Prolog [Car88] has a built-in predicate named *setarg* which destructively assigns a value to the argument of a structure. Undesirable behaviour could result if a use of *setarg* changed the static copy of a term. The kindest thing that could be said of *setarg* and similar predicates is that they are foreign to the ideals of logic programming. Certainly support of *setarg* or an equivalent is not a necessity for a Prolog implementation.

Parma could still make static copies of terms and support a restricted version of *setarg*. The user could be informed (somewhat conservatively) that *setarg* can only be safely applied to terms created by the built-in predicate *functor*/3. This should still allow *setarg* to be used for many purposes. A prudent implementor would check in *setarg* that the address of the term being modified does not lie in the area of memory where static copies of terms are kept. Alternatively, it may be possible to place the static copies of terms in a read-only section of memory. Unfortunately, as discussed later, *setarg* causes other problems.

Bruynooghe et al. [Bru87a] introduced a technique which they term "compile-time garbage collection". This technique uses global analysis to detect the last use of a structure and hence allows the compiler to reuse the space the structure occupies. This saves not only space but can also save time as it can allow some manipulation of the global stack and possibly some copies to be avoided.

A compiler which uses compile-time garbage collection must be careful about using static copies of terms. It should not be too difficult to ensure that unfortunate interaction does not occur. It is not clear which should take priority. Should compile-time garbage collection be applied only where the structure can never be part of a static term or should terms be made static only where compile-time garbage collection will not occur? The work on compile-time garbage collection has so far been largely theoretical so it is unclear what its benefits will be.

4.4 COMPILING CALLS

Apart from unifications and cuts which will be discussed in Section 4.6, the other basic component of normalised clauses is calls. Recall that the arguments to these calls can only be variables. The only difficult part of compiling a call to a Prolog predicate is in deciding which variables need to be stored in the environment to survive the call. This is discussed in the following section. Otherwise all that is necessary is to load the call registers with the appropriate values. The only complication is that sometimes this requires the swapping of register values via a temporary register as in this clause:

a(X, Y)) :- b(Y, X).

The values in the registers $call(1)$ and $call(2)$ require swapping.

Parma implements many built-in predicates by library routines, written in assembly language or the language C. Calls to these are compiled almost exactly as calls to Prolog predicates. These are also called in a similar way to Prolog predicates. Some built-in predicates are sufficiently simple that calls to them can be compiled to an in-line sequence of instructions. These include some predicates which are frequently called and whose efficiency can be very important such as: *var*/1, *atomic*/1 and other type-checking predicates; *is*/2 and other arithmetic predicates; ‘<’/2, ‘>=’/2 and other comparison predicates; *functor*/3, *arg*/3, *fail*/0 and *true*/0.

The gains from compiling such predicates in-line can be considerable. It avoids branching to and returning from an out-of-line routine. It is not necessary to move the argument values to the call registers. It may remove the need to save some variables. The code for the built-in predicate call is exposed to subsequent improving transformations such as common sub-expression elimination.

The information from global analysis may allow the in-line code for a predicate call to be much simplified. It may allow type-checking to be avoided. For example, most of the code for an arithmetic expression may be spent checking that the variables involved are actually integers. As with unifications, the information from global analysis may much reduce the code needed to bind a variable. There can also be other benefits. The built-in *functor*/3 performs different functions on the instantiation of its first argument. If global analysis reveals the instantiation of this argument, half the code can then be omitted.

4.5 MANAGING VARIABLES

Parma makes no attempt at global register allocation. Registers are only allocated within a predicate. This means it must assume that any call to another predicate may destroy the value in any of the registers used to hold the value of variables. Parma must ensure, if a variable's value is needed subsequent to the call, that it has been stored in a suitable memory location. Calls then become key points to consider when managing variables. Before a clause is compiled an initial pass records for each use of a variable its next-use designation of that variable. Four categories of designations are used:

- *argument*(*i*) - The variable occurs as the *i*-th argument to the next call. If the variable occurs as several arguments to the next call it is designated for the first of these arguments. This imprecise handling will occasionally produce unnecessary moves but this should be corrected by the intermediate code transformations described in Section 4.8.5.
- *before* - The variable is used before the next call but not in that call.
- *after* - The next use of the variable is after the next call.
- *none* - The variable is not used again.

Management decisions become much easier given the next-use designation of a variable. For example, if the register containing a variable's value is about to be destroyed then the appropriate action can be determined from the variable's designation as follows:

argument(*i*) - The variable's value should be moved to the register *call*(*i*). Recursive action is needed here if *call*(*i*) contains the value of another variable.

before - A new register should be allocated and the variable's value moved there.

after - If the variable's value has not been appropriately saved it should be stored in the environment.

none - No action is needed.

It is useful when considering how Parma stores variables to separate variables into three categories. Variables whose first occurrence in the clause is in a structure we will refer to as global variables. Variables whose first occurrence is in the head of the clause we shall refer to as argument variables. Variables which do not fall into the first two categories we will refer to as local variables.

The type, lifetime and references to a variable determine the memory location used for a variable. Local variables are stored, if necessary, in the *environment* which is a vector of variables on the local stack.

By default, argument variables whose lifetime crosses a call are stored in the environment. This may not be necessary in nondeterminate predicates. The value of the argument variable may be found in the predicate's choice-point. Even if subsequent choice-points have been created this choice-point will be accessible at a negative offset from the base of the environment. Parma, where possible, uses the values of argument variables stored in the clause's choice-point and avoids storing argument variables in the environment.

The value of an argument variable may change after it is stored, either in the choice-point or in the environment. If the variable was unbound when it was stored it may since have been bound. If this is so, the value stored will be the address of a location now containing the variable's value. If it was known that the variable was unbound then, regardless of whether the variable is now bound or not, all that is needed is an extra load instruction which loads the stored address.

However, if the variable may have been unbound or bound when it was stored then more code is needed. The tag of the stored value of the variable must be examined. If it is a reference tag then the variable was unbound when stored and, as before, its value can be obtained by loading the store address. Otherwise the value stored is the current value of the variable. It takes four instructions to perform this sequence of operations but, fortunately, the information from global analysis makes this rarely necessary.

A cut in a clause does not destroy the choice-point for that clause. It alters the *choicep* so that the choice-point is no longer referenced. A garbage collector should delete such unreferenced choice-points. Thus the garbage collector must know which clauses use the values of argument variables stored in the choice-point. It can only partially remove such choice-points. The slight impact on the efficacy and complexity of the garbage collector should be well offset by the gains in time and space from not storing argument variables.

It is not necessary to store global variables in the environment if a reference to the structure in which they occur is still available. For example, consider the following clause:

```
a(A) :- A = f(G), b(A), c(A, G).
```

The argument variable *A* must be stored in the environment to ensure it survives the call to *b*. It is not necessary to store the global variable *G* as after *b* is called and *A* is reloaded from the environment it can be loaded at an offset to the value of *A*.

This approach is unsafe in the presence of built-in predicates which allow destructive assignment to the arguments of terms such as SICStus Prolog's *setarg*. Limited use of this approach could be made by determining that such predicates will not be called in the execution of a particular goal. However, this is just one of several problems caused by such predicates and their provision is not essential to a Prolog implementation. Parma has abandoned any possibility of supporting predicates which allow destructive assignment to the arguments of terms.

If the variable referring to the structure which contains the global variable is not otherwise needed then the best approach is less clear.

4.5

Consider these two clauses:

c(A) :- A = f(G), b, c(G).
c(A) :- A = f(G1, G2, G3), b, c(G1, G2, G3).

In the first clause, the best approach is to store the global variable *G* in the environment so it survives the call to *b*. However, in the second clause the best approach is to store the argument variable *A* in the environment and, after the call to *b*, use its re-loaded value to load the values of the global variables G1, G2 and G3.

The second case is more common and by default Parma uses the second approach to manage global variables. However it attempts, by examining use information, to store a copy of the global variable in the environment in the cases when this is worthwhile.

4.5.1 Initialising Variables

It is common for one or more call arguments to be variables which have not been used before and it is also common for the called predicate to bind these variables. In an implementation without global analysis the caller will initialise the variable to the unbound value, then the called predicate will examine the variable and bind the variable. Global analysis may reveal that this particular argument is always an unaliased unbound variable. This allows the code which examines the variable's value to be removed. The code that initialises the variable could also be omitted if, when compiling the call, it was known that the variable's value will not be examined.

Parma makes this possible by designating as uninitialised any predicate argument position which is known from global analysis to always be an unaliased unbound variable on entry and always be bound when the predicate exits. Parma has a convention that a reference to an uninitialised variable can be passed in an uninitialised argument position. This moves the onus of initialising the variable from the caller to the called predicate. This usually allows the initialisation code to be omitted. The only exception is where the called predicate passes the variable on in a call to another predicate but not in an uninitialised argument position.

There is an added benefit to this convention. As the value of an uninitialised variable obviously cannot be examined on backtracking there is no need to record the binding of such variables on the trail even if a choice-point has been created since creation of the uninitialised variables. This allows a significant number of trailing operations to be omitted.

The uninitialised argument position convention is mundane compared to the intricacies of global analysis but the gains produced are substantial. The only drawback is that the garbage collector must know which predicate arguments are uninitialised argument positions if it is to avoid following spurious references.

Van Roy [Van90] refines this handling of uninitialised variables further by a technique where an uninitialised argument position can be designated to return its value in the argument register. This is only useful if the caller does not need to store the returned value. Otherwise it may be disadvantageous because the code in the caller which stores the returned register may prevent tail recursion optimisation (Section 4.7). The ability to return values in registers should be most useful in programs involving a great deal of arithmetic.

4.6 COMPILING DISJUNCTIONS

We will first describe a simple scheme for compiling disjunctions and then refine it to yield the techniques Parma uses. A disjunction must be compiled to produce a backtracking mechanism which tries subsequent disjuncts of a disjunction on failure. This can be done simply by inserting code before the first disjunct which creates a choice-point and stores the execution state and the address of the second disjunct's code in the choice-point.

A failure is initiated by branching to code which loads the execution state from the choice-point. The trail is unwound, restoring variables to their recorded values, until the *trailp* returns to the value in the choice-point. Execution then branches to the address of the alternative disjunct stored in the choice-point.

The second and subsequent disjuncts do not need to completely re-create the choice-point as the failure code leaves the choice-point created by the first disjunct in place. They only need set the alternative field of the choice-point to the next disjunct. The last disjunct must delete the choice-point. This is described in Figure 4.8.

```
L_start_disjunct:
                create choice-point
                set alternative = L_disjunct2
                <code for disjunct 1>
L_disjunct2:
                set alternative = L_disjunct3
                <code for disjunct 2>
L_disjunct3:
                delete choice-point
                <code for last disjunct 3>
```

Figure 4.8. Compiling a Disjunction

This scheme is inefficient in several respects. Failure before the first call in a disjunct, often referred to as shallow backtracking [War77], is more expensive than necessary. Most of the registers will not have changed and so it is not necessary to load them from the values stored in the choice-point. The compiler can improve on this behaviour by noting which registers have changed and, where failure must be initiated, emit a branch to code which only loads the appropriate registers from the choice-point. Multiple entry-points in the failure code can be employed to reduce the amount of code needed to implement this. If it is not possible that any registers have changed then execution can branch directly to the next disjunct.

A disjunct that always executes a cut before the first call will use its choice-point only for shallow backtracking. We will refer to such disjuncts as shallow determinate. The only registers such a disjunct needs stored in its choice-point are those which may be changed before a cut occurs or, more precisely, before the last possible initiation of failure before a cut occurs. The choice-point is not needed at all if no registers are changed. We can modify our scheme to take advantage of this. If the first disjunct is shallow-determinate it creates a choice-point only if it needs one and then only large enough to hold the registers it may change before becoming determinate. Code is inserted before subsequent disjuncts to expand the choice-point as necessary for their needs.

Prolog programming texts discourage at length the use of cuts. There are sound reasons for avoiding cuts, but unfortunately it removes a useful source of information which the compiler could use to detect shallow determinism. Parma uses information from its global analysis phase to detect where a clause becomes shallow-determinate. Effectively it infers the "green" cuts of [Ste87].

A disjunct which creates a choice-point but only uses part of it because early failure causes shallow backtracking has done unnecessary work. This work is wasted if subsequent disjuncts do not use the full choice-point. The unnecessary work can be avoided if the storing of registers in the choice-point is delayed as long as possible i.e. it is moved down the disjunct to just before the first point the register may be changed. This however means subsequent disjuncts cannot rely on these registers having been stored in the choice-point and must also store the registers if they need them in the choice-point. This means that delaying parts of choice-point creation may avoid storing some registers entirely or it may result in the registers being stored in the choice-point many times. This depends on the program's patterns of backtracking, both shallow and deep. The gain from the best-case, when storing the registers is avoided, is much less than the cost of the worst-case, when the registers are written many times. Parma makes the conservative decision of only delaying storing registers in the choice-point when it is known that subsequent disjuncts cannot need them.

These modifications make our backtracking scheme more complex but much more efficient. In combination with global analysis, this scheme would be sufficient to yield good performance from many disjunctions but it has some weaknesses. Consider the following program and assume X is known to be bound:

```
a(X) :- X = a, ....
a(X) :- X = b, ....
 ....
a(X) :- X = z, ....
```

Our scheme will produce efficient backtracking, creating no choice-point, but by sequentially trying disjuncts it is effectively doing a linear search for the value of X in the set {a ... z}. The compiler has little choice but to use this process if it compiles each disjunct in isolation. However if it examines disjuncts as a group it can employ faster searching techniques such as binary trees and hash tables. The use of these faster techniques to narrow the set of disjuncts that must be tried is usually referred to as indexing [War77].

4.6.1 Indexing Disjunctions

Most existing implementations provide only a limited form of indexing based only on the first argument to the predicate and usually only on the top-level of the first argument, i.e. if that argument is a structure then indexing only takes places on its functor and not on its sub-terms. This limited form of indexing will often be sufficient if the programmer is aware of its nature and arranges their program appropriately. It often will not be sufficient for very large "database" predicates containing very large numbers (perhaps millions) of clauses . Discussion of the specialised techniques required to produce good results for such predicates can be found in [Ram86], [Wis84] and [Col86]. As mentioned in the introduction, efficient compilation of such predicates is beyond the scope of this work.

Parma concentrates on handling disjunctions with less than one hundred disjuncts efficiently. Parma can produce indexing on any part of any argument position as well as on certain built-in predicates. Basically, given a list of disjuncts to index, Parma proceeds to examine the first subgoal of each disjunct gathering as many as possible of the leading disjuncts into a group to be indexed. If there are subsequent disjuncts, a choice-point must be created. The indexing algorithm is applied to the subsequent disjuncts.

A set of disjuncts can be indexed if the subgoal of each disjunct is the unification of a particular bound variable with a constant or functor, if the first subgoal of each disjunct is an arithmetic comparison involving a particular variable or if the first subgoal of each disjunct is a type predicate, such as *nonvar*/1 or *atomic*/1 with a particular variable as its argument. For example, if the first subgoal of the first disjunct is the unification of a variable known to be bound with a constant then Parma proceeds down the list gathering disjuncts until it meets a disjunct whose first subgoal is not a unification of a constant or functor with that variable. If this indexing is not sufficient to determine a single disjunct the indexing algorithm is applied recursively to the subsequent subgoals of the lists of disjuncts that have been determined.

The reordering of the subgoals described in Section 4.1 tries to ensure that the subgoals of each disjunct are in the best order for this process. The variable renumbering which follows the reordering means corresponding variables in different disjuncts have the same number. Two variables correspond if the parts of the disjunct bodies preceding each variable are identical.

Producing indexing code for a variable whose instantiation is not known is problematic. Consider the following program again and assume X's instantiation is not known at compile-time:

```
a(X) :- X = a, ....
a(X) :- X = b, ....
 ....
a(X) :- X = z, ....
```

The lack of information about X could be the result of imprecise global analysis or the predicate could be called in two modes. If X is unbound, the disjuncts must be tried sequentially but, if X is bound, something faster than linear search is desirable. Parma does this by compiling the disjuncts as though X was unbound. On entry to the predicate, if X is unbound, execution branches to this code. However, if X is bound, execution branches to indexing code which, if a match is found, branches part way to the appropriate disjunct so both choice-point manipulation code and binding X are avoided. This is depicted in Figure 4.9.

The use of the technique in Figure 4.9 is possible because the indexing determines the disjunct. Parma only forms indexing groups for variables which are not known to be bound if the indexing must determine the disjunct. This technique does introduce a problem when compiling a cut in a disjunct body, as a choice-point may or may not have been created on entry to the disjunct. Code must be added to the disjunct preamble and to the indexing code to store the *choicep* in an environment variable if there is a cut in the disjunct body.

A more aggressive approach to indexing coding has been adopted by others [Van87], [Hic87]. Parma has not done this, not because there is great difficulty in implementing their approaches but rather because we believe the additional benefits are too slim to justify increasing the complexity of the compiler and the increased code size these schemes may entail. We believe that, in combination with global

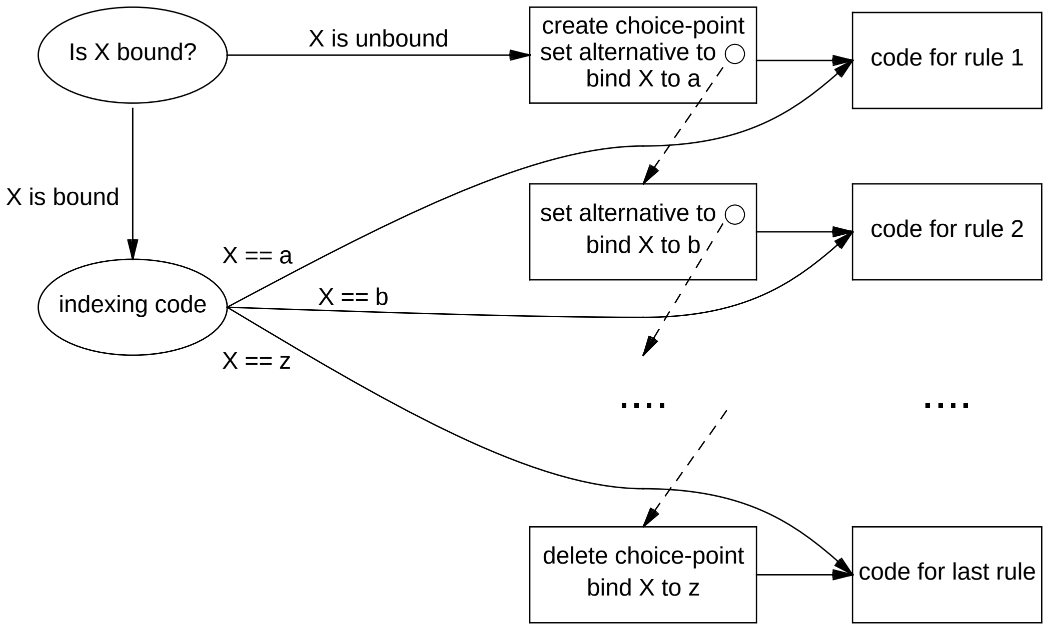


Figure 4.9. Indexing on a Variable of Unknown Mode

analysis and good shallow backtracking, the indexing scheme we have described will be sufficient. This is clearly a decision to be reviewed when we have experience with Parma's performance on a wider range of programs.

4.6.2 Index Searching

As the number of alternatives in an indexed disjunction grows large the technique used in the indexing code becomes important. The task of the indexing code can be presented more formally thus:

Given the key K and a set of n value-address pairs $\{(v_{1,}a_{1)},\ (v_{2,}a_{2)}\ \cdots (v_{n,}a_{n)}\ \}$
if there exists $v_i = K$ branch to a_i else fail.

The choices for implementing this search include linear search, binary tree, hash table and direct table lookup. The most important factor influencing this choice is n, the set size. Also important is the distribution of values the key K will take during program execution.

We have analysed the costs of various search techniques but we have assumed that K will always take each of the values with equal frequency. In other words we have assumed a uniform distribution. In practice this assumption will often be violated but we feel our analysis still yields a useful prediction of the performance of the search technique.

Information on the likely distribution of key values in a particular piece of indexing code could be obtained by dynamic profiling (see Chapter 3). This could allow the cost of indexing to be much reduced but, as this information is not available to Parma, we will not consider it further.

The most obvious search technique is linear search, sequentially comparing the key for equality with each of the values in the set. This is a special case of a comparison tree using comparison operations { =, !=, >, ≥, <, ≤ } to compare the key with members of the set. We have, by exhaustive search, found optimal (in time) comparison trees for sets up to one hundred in size, for the MIPS, under the above assumption. The exhaustive search is necessary because low-level considerations mean that there is no obvious way to construct an optimal comparison tree code for the MIPS instruction set. Obvious methods will produce close to optimal code but these fell sufficiently short of the optimal code for the use of exhaustive search to be worthwhile. The comparison tree optimal in space is always a linear sequence of comparisons for equality, i.e. linear search. As can been seen from Table 4.1 the extra space of a comparison tree is worthwhile even for small sets.

	Linear search			Comparison Tree		
Set size	Average hit (instructions)	Average miss (instructions)	Size (words)	Average hit (instructions)	Average miss (instructions)	Size (words)
5	7	13	13	6.6	8.6	17
10	12	23	23	8.6	10.6	31
20	22	43	43	11.1	13.1	71
30	32	63	63	12.5	14.5	96
40	42	83	83	13.5	15.5	128
50	52	103	103	14.3	16.3	165
75	77	153	153	15.8	17.8	240
100	102	203	203	16.7	18.7	318
200	202	403	403	22.7	24.7	685

TABLE 4.1. Linear Search vs. Comparison Tree

A great variety of work has been done on more complex searching techniques [Knu73]. Fortunately the good performance of the comparison tree over the range of set sizes that we are concerned with allows us to eliminate many searching techniques. A great variety of work has been done on hashing but all but the simplest hashing functions will be too expensive for our purposes. The most popular simple hashing function is the key modulo the table size. A general modulus operation is too expensive for our purposes so we must restrict the table sizes to be a power of two allowing a much cheaper bitwise *and* to be substituted for the modulus. This hashing function will produce poor results if the key is a tagged value and a particular tag predominates as is likely. We avoid this by (bitwise) shifting the key down to remove the tag.

There are also an assortment of techniques for dealing with collisions in the hash table. Again we can restrict our attention to the simplest techniques. Two such techniques are separate chaining and linear probing [Knu73]. Linear probing stores values which collide in subsequent table entries. Separate chaining stores values which hash to the same table entry in the list pointed to by that entry. We have analysed the performance of both on the MIPS. For separate chaining we have used a variant where the table entry contains the address of code which does a linear search through the values which hash to that address. This is faster but uses slightly more space than a linked list of data. We have assumed the hashing function distributes values uniformly over the hash table. This is an optimistic assumption given the hash function's simplicity. It can been seen from table 4.2 that both hashing techniques compete with comparison trees only at the top end of the range we are considering and at the cost of considerable extra space.

	Hashing - Linear Probing			Hashing - Separate Chaining		
Set size	Average hit (instructions)	Average miss (instructions)	Size (words)	Average hit (instructions)	Average miss (instructions)	Size (words)
5	17.8	33.1	31	11.3	11.7	38.7
10	17.8	33.1	47	11.3	11.7	67.8
20	17.8	33.1	79	11.3	11.7	126.0
30	14.1	21.9	143	11.5	12.7	160.0
40	16.8	33.1	143	11.3	11.6	243
50	13.2	16.9	271	11.4	12.2	278
75	16.0	27.9	271	11.3	11.5	458
100	13.2	16.9	527	11.4	12.2	546
200	13.2	16.9	1039	11.4	12.2	1082

TABLE 4.2. Hashing - Linear Probing vs. Separate Chaining

Atoms and functors are represented by their indices in the atom and functor tables. The compiler, by rearranging these tables can change the representation of a particular atom or functor. This makes searching a set of functors or atoms a somewhat unusual problem in that, not only is the set of values that must be searched known to the compiler, but it can manipulate these values to obtain better performance. It cannot, of course, manipulate the representations of integers and floats which may also occur in tables of constants.

It may be possible for the compiler to arrange the values of atoms or functors so that the hashing function yields no collisions. This yields slightly better results than the previous hashing methods as can be seen in Table 4.3

If the set of values in the table lie within a small range then the key's value can be used to directly index the table. Firstly the key is shifted right to remove the tag, then the minimum table value is subtracted. This value is checked to ensure it lies in the range zero to the table size, inclusive, otherwise execution branches to failure code. Finally the value is used to index the table to yield the address to branch to. Gaps in the range are handled by placing the address of failure code in the table. As can be seen in Table 4.3 direct indexing is reasonably fast for large sets and economic in its use of space. However in Table 4.3 it is assumed that the values in the set are contiguous - no gaps. While a few sets will have few gaps and approach these results many will not. Again, if it is a set of functors or a set of atoms, the compiler may be able to re-arrange the atom or functor tables to ensure this occurs. It should be noted that although the values of a set of integers are fixed these will often lie within a small range and so direct indexing will be feasible.

	Perfect Hashing			Direct Indexing		
Set size	Average hit (instructions)	Average miss (instructions)	Size (words)	Average hit (instructions)	Average miss (instructions)	Size (words)
5	11	9	27	13	5	18
10	11	9	43	13	5	23
20	11	9	75	13	5	33
30	11	9	75	13	5	43
40	11	9	139	13	5	53
50	11	9	139	13	5	63
75	11	9	267	13	5	88
100	11	9	267	13	5	113
200	11	9	523	13	5	213

TABLE 4.3. Perfect Hashing vs. Direct Indexing

Both perfect hashing and direct indexing have drawbacks. Given a number of hash tables it may not be possible to arrange the values of atoms or functors so that no collisions occur in any of the hash tables. It is shown in Appendix A that determining if such an arrangement exists is an NP-hard problem. Even if such an arrangement exists it may be necessary to make the atom or functor table unacceptably large to achieve it. Similarly, given a number of direct indexing tables, finding an arrangement of the atom or functor table to minimise the the total size of the direct indexing tables, i.e. minimise the number of gaps, is NP-hard. Neither of these complexity results is an insurmountable hurdle. Straightforward algorithms which take an incremental approach to the re-arrangement of the functor or atom table should provide useful results.

Ideally a compiler should have a suite of indexing strategies, choosing the best strategy for each piece of indexing code. In practice, we were discouraged by the small size of the indexing sets in the programs we were interested in. The only indexing technique Parma employs is the comparison tree. A pre-computed table containing the optimal comparison tree code for sets of up to size 50 is employed.

4.7 COMPILING PREDICATES

Parma compiles a predicate by treating it as a disjunction composed of the bodies of the clauses in the predicate. The only extra code which must be produced is for management of the stacks.

Any clause which contains a call must have an environment allocated. Parma has a convention that any predicate which is always determinate must never leave anything on the local stack. This ensures that calls to such predicates leave the *stackp* unchanged. This is also true of most built-in predicates.

Clauses which call predicates which may be non-determinate or which contain disjunctions which create choice-points must allocate an environment on the local stack, save the old values of the *envp* and the *successp* in this environment and set the *envp* to point to it, so it can be later accessed. Clauses which contain only determinate calls do not need to use the *envp* and hence do not need to save it in the environment.

Parma delays the allocation and deallocation of the environment for as long as possible. The purpose is to reduce the work done if an early failure occurs and also possibly to reduce the information that must be stored in shallow choice-points. Delaying choice-point allocation is aided by Parma, when necessary, storing variables where the environment will be, above the *stackp*, without actually allocating the environment.

A Prolog implementation must make some provision for overflow of the various stacks it employs. Parma does not perform explicit bounds checking when extending a stack. Rather it traps the fault produced by an out of bounds reference. Currently the trap routine just prints a suitable error message and aborts execution. A full implementation would either extend the appropriate stack, call the garbage collector or both. This facility cannot be supported on all operating systems. Indeed the support from the MIPS's operating system is not ideal. This approach causes problems for the garbage collector as it may be invoked at a time when the stacks are in an inconsistent state. It would be feasible for the garbage collector to emulate execution until it reaches a point, such as a call, where the stacks are known to be in a consistent state. Alternatively, a robust garbage collector may be able to operate with the stacks in an inconsistent state.

There is a well known improvement that can be made if a call occurs as the last operation in a clause. If the clause is determinate and the arguments of the call cannot refer to the environment then the call can be replaced by code which first restores the *envp* and the *successp*, deallocates the environment and then branches to the routine. This results in execution not returning to the clause. This is usually called tail recursion optimisation (TRO). There are two other advantages to TRO besides avoiding the return branch. If TRO can be applied to a clause containing only one call then the creation and deallocation of an environment is no longer necessary. If the last call is a recursive call then execution will now branch to the start of the predicate. This transforms the predicate into a loop which may allow further improvement from intermediate code transformations.

4.8 INTERMEDIATE CODE TO ASSEMBLY LANGUAGE

The translation of intermediate code to MIPS assembly language is broken into eight phases. Fortunately these phases are much simpler than Parma's earlier phases. Most of the techniques involved are applicable to many languages and thus well studied. As a result there is little innovation in this part of Parma but the operations performed are important. Figure 4.10 depicts the order of these phases.

4.8

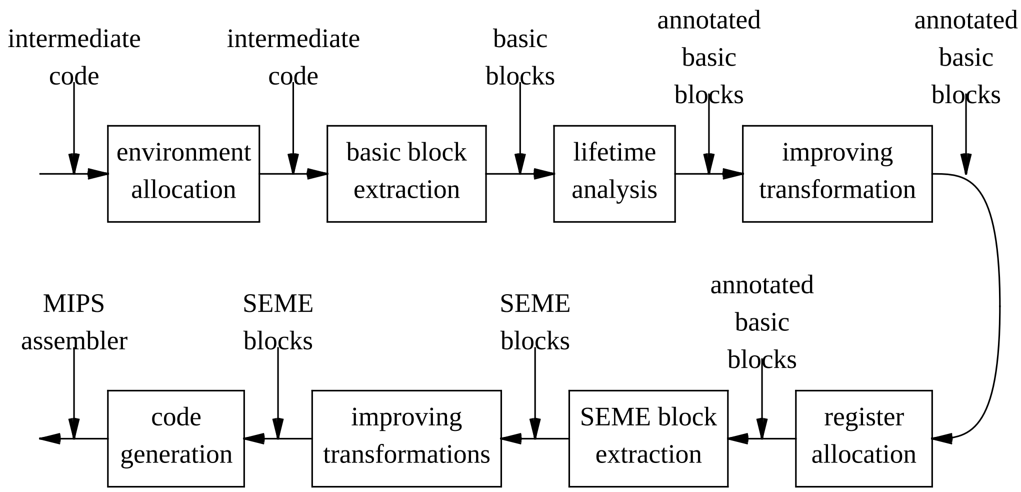


Figure 4.10. Parma's Phases: Intermediate Code to Assembler

A Catch-22 problem occurs with the ordering of the phases. It is desirable that allocation of variables to slots in the environment precede the improving transformations applied to intermediate code so the stack frame offsets used to access these variables are visible to these transformations. It is also desirable that register allocation be done after these improving transformations as the transformations remove some uses of registers. Unfortunately, if insufficient registers are available, the register allocation code will need to spill some registers to slots in the environment. This, of course, requires changes to the environment allocation. The abundance of registers on the MIPS makes register spills a rare event so we have adopted an ad hoc solution. If a register spill is needed, the number of environment slots needed are determined and compilation is begun again at the stack frame allocation phase. This repetition of operations is inefficient but occurs very rarely. Overall, it is faster and produces slightly better code than the obvious alternative of performing improving transformations, register allocation, stack frame allocation and then repeating the improving transformations.

4.8.1 Environment

The main compilation phase produces the instructions which manipulate the environment on the local stack but it leaves the size of the environment and the positions of the variables in this environment unspecified. Parma does not prune the size of an environment during the execution of a clause. This makes allocation of environment slots a task easily accomplished by a backward pass through the code for a predicate.

4.8.2 Basic and SEME Blocks

A basic block is a maximal sequence of consecutive instructions which can only be entered at the top and only left at the bottom. Hence, only the first instruction in a block can be the target of a call or goto and only the last instruction can be a call or goto. The intermediate code is formed into basic blocks because they are a convenient scope for some improving transformations.

Other improving transformations operate within the scope of single entry multiple exit (SEME) blocks. These are maximal sequences of consecutive instructions which can only be entered at the top. They are obtained by merging basic blocks.

4.8.3 Lifetime Analysis

We will refer to a register as *live* at a particular program point if it is possible a subsequent executed instruction may use the value in that register. Otherwise the register is termed *dead*. A register is *killed* if the value currently in the register is changed. Dataflow analysis is used to discover which registers are live at the bottom of each basic block[Aho86]. Each basic block is annotated with this information.

4.8.4 Register Allocation

The main compilation phase assumes an unlimited number of registers are available. This phase allocates these symbolic registers to the actual 28 MIPS registers available to user programs. When these 28 registers are insufficient, values must be spilled to memory. Quite sophisticated algorithms exist for register allocation. A very simple register allocation algorithm was used in the construction of Parma because we planned to latter employ an algorithm which allocated registers globally (between predicates). This simple algorithm proved more than adequate due to the abundance of MIPS registers so it was retained when plans to construct a global register allocator were abandoned.

Parma’s register allocator reserves 18 of the 28 registers available for fixed purposes. This included the named registers used to represent the execution state such as the *choicep* and the *envp*. A more ambitious register allocator could make use of at least some of these 18 registers for general purposes at certain times. We discovered that it was advantageous for the register allocator to keep free registers in a queue so that a freed register was not reused. This resulted in greater freedom for the assembler in instruction scheduling.

4.8.5 Improving Intermediate Code

The techniques which could be used to improve Parma's intermediate code are applicable to many languages. Some of these, for example Fortran, have been the focus of much research. As a result, there exists in the literature an extensive array of techniques which could be employed to improve Parma's intermediate code. However, the benefits of many of these transformations may be much less than if they were employed, for example, in a Fortran compiler. This is both because of the nature of the intermediate code produced by Parma and because the typical nature of Prolog programs is somewhat different to Fortran. Parma employs only a few of the simpler techniques. We feel these produce a large fraction of the benefit that would result if all the best and most complex techniques published were applied. Currently we believe work on improving Parma would be more fruitfully applied in areas other than intermediate code transformation.

Several important transformations are made within the scope of basic blocks. Many compilers remove unnecessary sequences of register-to-register moves. This is important because such sequences arise not only as an artifact of the user's program but also as a result of the earlier phases of compilation. It is easier to remove these sequences at this stage rather than change the earlier compilation phases so that they do not produce such sequences. Here is the template for a transformation which removes some unnecessary moves:

y = x <code which does not kill x or y> z = y # y is dead here	→	<code with uses of y replaced by x> z = x

The code Parma produces involves much manipulation of stacks with consequent sequences of additions to the stack pointer and use of the stack pointer. This makes generalising the previous transformation to sequences of additions useful. This can be expressed with the follow template:

y = x + c_1 <code which does not kill x or y> z = y + c_2 # y is dead here	→	<code with uses of y replaced by x + c_1> z = x + c_1 + c_2

If c_1 is non-zero this transformation is restricted to code where all uses of *y* can be replaced by x + c_1. This is possible if *y* is used as the address register in a load or store instruction as *x* can be used as the address register and c_1 added to the constant offset. Similarly it is possible if *y* is the operand in a constant addition because *y* can be replaced with *x* and c_1 added to the constant. It is not possible for other instructions. Here is an example of this transformation:

y = x + 4 store a, y[0] b = y + 1 z = y + 4 # y is dead here	→	store a, x[4] b = x + 5 z = x + 8

The second improvement which is made within the scope of the basic blocks is that repeated re-calculation of expressions are removed. This again is widely used and important to many compilers. It is usually termed common sub-expression elimination. Here is an example of this transformation:

```
b = a * 2
c = b + 1
d = a * 2
e = d + 1
# d is dead here
```

→

```
b = a * 2
c = b + 1
e = c
```

The algorithms for this can be found in standard texts such as [Aho86] and will not be described here. Also at this stage instructions whose target is a dead register are removed.

The improving transformations Parma uses within the scope of SEME blocks apply only to loops. A loop, in this context, is a section of code starting at the top of the SEME block and finishing with a branch, conditional or unconditional, whose target is the top of the SEME block. These result from tail-recursive predicates. In languages such as Fortran producing efficient code for loops is of great importance so it has been well researched. Parma employs two simple well-known transformations.

The first transformation applies to loops which test the loop condition at the top i.e. their first instruction is a conditional branch to the instruction following the loop and the last instruction is an unconditional branch to the top of the loop. These loops can be improved by testing the loop-condition at the bottom as well. This saves one or two instructions per loop iteration. Here is a template for this transformation:

```
L1: if <condition> goto L2
<code>
goto L1
L2: ...
```

→

```
L1: if <condition> goto L2
L3: <code>
if ! <condition> goto L1
L2 ....
```

Often loops will contain instructions which calculate the same value on every loop iteration. It is desirable, assuming that on average the loop will be executed at least once, to move such instructions out of the loop. It is easy to detect such instructions by examining which registers are changed inside the loop. We will not describe the technique in detail here as it can be found in standard texts such as [Aho86]. Here is an example of this transformation:

L1: <*code*$_1$>		
# *code*$_1$ doesn't kill x, y or z		y = x + 2
y = x + 2		z = y * 3
z = y * 3	→	L1: <*code*$_1$>
<*code*$_2$>		<*code*$_2$>
# *code*$_2$ doesn't kill x, y or z		goto L1
goto L1		

4.8.6 Code Generation

The small gap between Parma's intermediate language and the MIPS instruction set makes the task of code generation almost syntactic. The only awkward aspect of the MIPS architecture, delay slots, can be ignored by the code generators. The assembler re-schedules instructions to fill the delay-slots. Perhaps the most important task of the code generator is to ensure that the resulting code is readable by humans, or at least compiler-writers.

4.9 A COMPILATION EXAMPLE

It is probably useful to review Parma's operation by following a small example through the phases of compilation. The example we will use is the well-known predicate *append*/3. Here is the program we will assume it is contained in:

```
main :–
    append([a,b,c], [d,e], X),
    write(X),
    nl.

append([], L, L).
append([X|L1], L2, [X|L3]) :–
    append(L1, L2, L3).
```

Parma's first phase converts the clauses of *append* into a normalised form:

```
append(V1, V2, V3)  :–
        V1 = [],
        V3 = V2.

append(V1, V2, V3)  :–
        V1 = [V4|V5],
        V3 = [V6|V7],
        V6 = V4,
        append(V5, V2, V7).
```

The global analysis phase annotates these normalised clauses with the information discovered during the abstract execution of the program.

```
append(V1, V2, V3)  :–
        V1{list(atom, nil)}  = [],
        V3{free(no_trail, no_chain)} = V2{list(atom, nil)}.

append(V1, V2, V3)  :–
        V1{list(atom, nil)} = [V4|V5],
        V3{free(no_trail, no_chain)} = [V6|V7],
        V6{free(no_trail, no_chain)} = V4{atom},
        append(V5, V2, V7).
```

The main compilation phase translates these annotated clauses to Parma's intermediate code. We have rendered the intermediate code in pseudo-C:

```
p_append_3()
{
        g1 = CONST([])
        if (call_1 == g1) goto L2
        call_3[0] = globalp
        g4 = globalp
        globalp = globalp + 8
        g5 = call_1[O_CAR]
        g4[O_CAR] = g5
        call_3 = g4 + O_CDR
        g6 = call_1
        call_1 = g6[O_CAR]
        goto p_append_3
L2:
        *call_3 = call_2
        goto successp
}
```

Subsequent phases apply improving transformations to this intermediate code and perform register allocation. Here is the result:

```
p_append_3()
{
        r2 = CONST([])
        if (call_1 == r2) goto L2
L4:
        r3 = call_1[-2]
        call_1 = call_1[2]
        call_3[0] = globalp
        call_3 = globalp + 2
        globalp[-2] = r3
        globalp = globalp + 8
        if (call_1 != r2) goto L4
L2:
        *call_3 = call_2
        goto successp
}
```

The code generator translates intermediate code to the MIPS assembly language:

```
p_append_3:
        li      2, 5
        beq     arg1, 2, L2
L4:     lw      3, -2(arg1)
        lw      arg1, 2(arg1)
        sw      globalp, 0(arg3)
        add     arg3, globalp, 2
        sw      3, -2(globalp)
        add     globalp, globalp, 8
        bne     arg1, 2, L4
L2:     sw      arg2, 0(arg3)
        j       successp
```

The MIPS assembler (not part of Parma) schedules instructions in delay slots and produces the final code for *append*:

```
0x4001d0:   24020005   li     v0,5
0x4001d4:   10820008   beq    a0,v0,0x4001f8
0x4001d8:   00000000   nop
0x4001dc:   8c83fffe   lw     v1,-2(a0)
0x4001e0:   8c840002   lw     a0,2(a0)
0x4001e4:   acd40000   sw     s4,0(a2)
0x4001e8:   22860002   addi   a2,s4,2
0x4001ec:   22940008   addi   s4,s4,8
0x4001f0:   1482fffa   bne    a0,v0,0x4001dc
0x4001f4:   ae83fff6   sw     v1,-10(s4)
0x4001f8:   03e00008   jr     ra
0x4001fc:   acc50000   sw     a1,0(a2)
```

5 PERFORMANCE

This Chapter examines the performance of Parma and compares its performance to other Prolog implementations. We also attempt to evaluate the utility of individual features of Parma to ascertain whether their inclusion in Parma was worthwhile.

There is an apparently common belief in the computer industry that the performance characteristics of complex software and hardware systems can be reduced to a single number. This would be considered absurd in other fields. No serious presentation would reduce the performance of a car to a single number. Instead a number of performance metrics quantifying such attributes as acceleration, fuel economy and braking would be presented. It is no more sensible to attempt to reduce the performance of a Prolog compiler to single number. Aspects such as backtracking, indexing and recursion will be of differing importance to applications and will be handled with differing degrees of efficiency by Prolog compilers.

Although these multiple aspects of Prolog compiler performance clearly exist they are difficult to isolate and measure. We considered constructing programs designed to measure only a single aspect of compiler performance. We decided such programs would almost certainly have other characteristics not found in real programs which would interfere with the measurement. For example, a program constructed to spend a large fraction of its execution making recursive predicate calls so that call overhead can be measured may force a Prolog compiler for the MIPS to leave most delay slots unfilled. The results would be misleading as performance would be reduced by a feature of the benchmark which may never occur in real programs. Instead we have chosen to use a set of varied real programs as benchmarks. Although it is difficult to isolate individual performance aspects from the results, we feel the variation in the set of programs is sufficient that all important performance aspects will be reflected in the results of at least some of the programs in the set.

Over half the programs in our benchmark set are small. Small benchmark programs present at least two perils. Program size in itself can be an important characteristic.

For example, a small program may obtain atypically good caching performance on a machine with a very small cache. A particular concern for Parma is that the information global analysis yields for small programs may be atypically precise.

When a benchmark program becomes widely used to compare competing products its performance becomes an important concern to implementors. They may invest considerable effort in improving the performance of the benchmark program. This is a particular problem for small benchmarks as their performance can often be significantly improved by introducing very specialised improving transformations. These transformations will be applicable to, at best, a few other programs. This, of course, makes the benchmark much less useful for predicting the performance of other programs. This is not done in Parma but there are rumours in the Prolog community that some commercial Prolog implementations apply such very specialised transformations to the tiny, but widely used, naive reverse benchmark. Even when a suite of benchmarks is used a very specialised transformation applied to a single small benchmark may significantly effect the combined result.

We have retained the small programs in our benchmarks, despite the perils, for a number of reasons. We were restricted in our choice of benchmark programs. We could not use benchmark programs which relied on built-in predicates which we have not implemented. Other programs were not of interest to us as benchmarks because their performance was dominated by the cost of calls to certain expensive built-in predicates such as *read*/1, *write*/1, *assert*/1 and *retract*/1. The cost of these built-in predicates has not been changed significantly by Parma and so the performance of such programs will be little improved. Many will see this as poetic justice for many of the programs which make heavy use of *assert*/1 and *retract*/1. Much of the use of *assert*/1 and *retract*/1 has long been discouraged within the Prolog community as poor programming style.

Small programs are very useful to a compiler writer as the factors behind their performance are easier to analyse than in larger programs. Large size does not inherently make a program more useful as a benchmark. Many large programs spend a great fraction of their time executing in a small portion of their code. A 1000 line program that spends 80% of its time in determinate execution of *append*/3 is little more useful than naive reverse as a benchmark and considerably less convenient.

5.

	predicates	clauses	lines	analysis time (seconds)	description
nreverse	3	5	10	0.42	Naively reverses a 30-element list.
tak	2	3	15	0.44	A heavily recursive program. The recursion is controlled by simple integer arithmetic.
qsort	3	6	19	0.84	A 50-element list is sorted using quicksort.
pri2	5	9	28	0.21	A list of the first 100 integers is created and the composite numbers are removed by testing for factors.
serialise	7	13	29	1.12	The serial numbers for a list of 25 integers is calculated.
queens_8	7	11	31	0.35	Eight queens are placed by exhaustive search on an empty chessboard so none are under attack.
mu	8	16	33	0.80	A theorem is proved in Hofstadter's mu-math system.
zebra	6	11	36	1.9	A logic puzzle of constraints is solved by exhaustive search.
deriv	10	19	44	2.0	Simple exercise in symbolic differentiation
crypt	9	27	64	1.7	A crypto-arithmetic problem is solved.
query	5	54	68	1.1	A simple database is searched to answer a query involving multiplication and division.
prover	11	20	81	5.3	A small theorem prover.
poly_10	11	28	86	3.1	Symbolic arithmetic is used to raise a polynomial to the 10th power.
browse	14	29	92	1.3	An exercise in pattern matching.
press1	46	21	273	15.0	Several examples are run using part of the Press symbolic arithmetic system.
reducer	41	119	301	9.3	Graph reduction of T-combinator expressions.
boyer	25	135	377	15.0	A theorem is proved using part of a Boyer-Moore theorem prover.
nand	43	139	493	20.0	A logic circuit synthesis program based on heuristic search.
chat_parser	156	514	1138	37.0	The sentence parser from Chat, a program which answers geographical queries.

TABLE 5.1. Benchmark Descriptions

Somewhat disheartened by the difficulties of benchmark selection, we decided to adopt 17 of the benchmarks used by [Van90]. This had the advantage that many of the benchmarks are well known in the Prolog community. We have included two programs from other sources (*pri* 2 and *press* 1). We would like to think our benchmark programs are typical but it is difficult to know what is typical in Prolog programs. Prolog programming style is still evolving and the use of Prolog is still spreading into new application areas. We believe this benchmark set is adequate to achieve our aims. Improving the state of Prolog benchmarking will have to be left to others.

Table 5.1 contains a short description of each benchmark. The number of predicates, clauses and lines in each is given. The time taken, in seconds, for global analysis is also listed. These times are more than acceptable, most falling in the 20-30 program lines/seconds range. This is, of course, with the analysis phase implemented in SICStus Prolog. A great performance improvement should result if the analysis phase could be compiled by Parma. This is currently not possible because Parma does not implement some of the built-in predicates the analysis phase uses. It is also pleasing that there is no hint of anything worse than linear increase in analysis as program size increases. The 19 benchmark programs are listed in full in Appendix B. This is essential as most of the programs exist in other, slightly different, incarnations.

5.1 PARMA VERSUS SICSTUS

The hardware used for our measurements was a MIPS RC3230 desktop workstation configured with 72 megabytes of memory. It uses a 25MHz R3000 microprocessor. The R3000 has a 32-bit architecture which was described in Section 1.3. It has separate instruction and data caches each of 32 kilobytes. The caches are direct-mapped and write-through. The operating system used was RISC/OS 4.51, a version of Unix.

We have used the MIPS RC3230 to measure the execution times of the benchmarks compiled with Parma. We have also measured the performance of the benchmarks using another software implementation on the MIPS RC3230, SICStus Prolog (Version 0.6 #14) [Car88]. SICStus belongs to the Edinburgh family of Prologs. It is reasonably compatible with a number of other widely used Prolog implementations. We believe it is typical of Prolog implementations currently in use. SICStus has two modes of program execution. It can either interpret a program or compile the program to an abstract instruction set which is executed by a software emulator. The abstract instruction set is similar to the WAM. SICStus is written in C and runs on a variety of Unix machines. It should not be assumed that SICStus's performance is close to maximal for an abstract machine based implementation. It has other priorities beside high performance such as portability. However, we believe it would be difficult to obtain more than a factor of two improvement in its performance on the MIPS.

The measurements on the MIPS RC3230 were done with the system in multi-user mode but with it only lightly loaded. The Unix command *time* was used to measure the execution time of binaries produced by Parma. Only the *user* time reported by *time* was used. As it is possible for other system activity to slightly inflate the times produced, each program was run three times and the fastest time used. No variation of more than 3% was observed between runs. The predicate *statistics*/2 was used for the timing of SICStus measurements.

The execution time of many of our benchmarks is very small. This caused measurement problems when using both Parma and SICStus. This was remedied by executing the benchmark multiple times for each of the three measurements. The number of executions used ranged between 10 and 100,000 times. This effectively results in a warm start for caching and paging. This has only a minor effect on the results and probably makes the benchmarks with short execution times more realistic. Our estimate that out timings are accurate to slightly less than two digits of precision which is sufficient for our purposes.

The first column of Table 5.2 contains the benchmark execution times measured for SICStus compiled mode in seconds. The remaining columns contain execution times for SICStus and Parma scaled inversely relative to this time. In other words, the number given is the speed-up factor over SICStus(compiled). We have resisted the temptation to provide any form of average of the times because we feel such a number would be misleading.

As expected SICStus performs much better in compiled mode than in interpreted mode. The performance improvement exceeds a factor of 8 for many of the benchmarks. This is dwarfed by the performance improvement of up to a factor of 66 that Parma offers over SICStus(compiled). Proponents of the naive reverse benchmark would rate Parma at just below 3 megaLIPS. However, the results for other benchmarks suggest that naive reverse overrates Parma's performance relative to SICStus(compiled). The great variation in Parma's performance relative to SICStus(compiled) clearly indicates that any single performance number will be of

	SICStus compiled		SICStus interpreted	Parma no analysis	Parma
program	seconds	scaled	scaled	scaled	scaled
nreverse	0.0056	1	0.095	4.1	34
tak	2.0	1	0.064	23.0	66
qsort	0.0083	1	0.11	7.4	39
pri2	0.019	1	0.12	7.9	19
serialise	0.0069	1	0.17	15.0	30
queens_8	0.041	1	0.90	8.7	29
mu	0.012	1	0.17	7.0	16
zebra	1.3	1	0.27	6.6	10
deriv	0.0019	1	0.12	4.4	24
crypt	0.18	1	0.12	17.0	53
query	0.047	1	0.11	8.0	15
prover	0.015	1	0.15	7.5	15
poly_10	0.68	1	0.18	6.2	17
browse	12.0	1	0.11	8.1	18
press1	0.20	1	0.11	7.9	17
reducer	0.54	1	0.14	5.5	7.9
boyer	8.3	1	0.77	3.7	4.1
nand	0.36	1	0.16	7.9	23
chat_parser	2.4	1	0.21	11.0	19

TABLE 5.2. Parma - Execution Times

little predictive use. Although the information from global analysis clearly has great benefits, it can be seen from Parma's performance without global analysis that Parma's compilation to MIPS native-code was responsible for much of the improvement.

From informal analysis it seems that Parma's exceptional performance on benchmarks such as *tak*, *qsort* and *crypt* results when global analysis allows important predicates to be greatly simplified. This allows important improving transformations to be applied to the intermediate code. The final MIPS assembler that results in these cases is delightful to the compiler writer's eye. It is little different to the MIPS assembly language that would result from an equivalent C program.

There are several factors involved in the cases where Parma produces the smallest performance gains. The information produced by global analysis for the *boyer* benchmark is poor. This is difficult to remedy. The heart of the problem is the predicate *rewrite_args*/3 which incrementally instantiates a compound term using the built-in predicate *arg*/3. The resulting term will always be ground but this can only be inferred by induction. The loss of precision for this predicate propagates through the entire program. A single exit mode declaration for the predicate *rewrite_args*/3, indicating that its third argument is always ground on exit, produces a performance improvement of more than a factor of two. Little performance improvement is possible for the *zebra* benchmark because its performance is dominated by calls to the general unify routine. This could perhaps be remedied by sophisticated transformations. The *reducer* benchmark appears, as far as we can determine, less tractable.

These measurements give us confidence that Parma will offer many programs a factor of 15 or better performance improvement over SICStus(compiled). In some cases the performance gain will be less. Methods for improving performance in these cases is obviously an important area for further work.

	SICStus compiled		SICStus interpreted	Parma no analysis	Parma
program	bytes	scaled	scaled	scaled	scaled
nreverse	1254	1	1.03	0.84	0.33
tak	684	1	1.33	0.68	0.28
qsort	1596	1	1.17	1.03	0.42
pri2	1658	1	1.04	0.91	0.25
serialise	3026	1	0.93	1.12	0.41
queens_8	2284	1	0.91	0.73	0.19
mu	3432	1	1.12	1.04	0.57
zebra	2502	1	1.31	1.70	1.27
deriv	5255	1	0.84	0.91	0.31
crypt	5238	1	0.96	1.47	0.64
query	5920	1	0.85	1.27	0.16
prover	7002	1	0.91	1.39	0.77
poly_10	7824	1	0.77	1.16	0.32
browse	7774	1	0.87	1.25	0.65
press1	26759	1	0.90	1.08	0.69
reducer	24864	1	0.90	1.48	1.10
boyer	25702	1	0.89	2.47	1.57
nand	44644	1	0.81	1.59	0.56
chat_parser	95051	1	0.86	1.17	0.71

TABLE 5.3. Parma - Static Code Size

The first column of Table 5.3 contains the code size in bytes produced for each of the benchmarks with SICStus(compiled). The remaining columns contain the code sizes for SICStus and Parma scaled relative to the size for SICStus(compiled). Again we have decided not to provide any form of average.

The measurements only include space necessary for program predicates. They do not include the size of routines from the run-time library such as the routine used to implement *write*/1. They also do not include the space consumed by the atom table and other such data. The code sizes for SICStus were measured indirectly and so may be slightly inaccurate.

The data space consumed by terms constructed at compile-time is included in the measurements for Parma. However this space is considerably smaller than would be required for the code to construct such terms at run-time. This has a large effect on *nreverse*, *qsort* and *deriv* because they are small programs containing large terms which can be constructed at compile-time.

Our primary interest in code size is to ensure that Parma does not produce significantly larger code than SICStus(compiled). This could result in poor performance in large programs because of its impact on caching and paging. The results in Table 5.3 make it clear that this is not a problem. In most cases the code produced by Parma is smaller than that produced by SICStus in either mode. There is some variation but even in the worst case the code produced by Parma is less than twice as large. Even without global analysis the code Parma produces would probably not provoke significantly worse caching or paging than SICStus(compiled).

5.2 PARMA VERSUS THE BAM

For comparison with Parma, we have used measurements for a Prolog implementation on another hardware platform, the Berkeley Abstract Machine (BAM) [Hol90]. The BAM is a general-purpose microprocessor whose instruction set has been extended to provide extra support to Prolog. This is a particularly interesting comparison because much of the general-purpose core of the BAM instruction set is similar to the MIPS architecture. There are still important differences in the general-purpose core. For example, the BAM has a more complex implementation of branch delay slots. Different versions of the branch instruction allow the instruction in the delay slot to be annulled [Hen90] if the branch is taken or not taken. The BAM results [Van90] were measured by simulating execution of a BAM with 128 kilobyte instruction and data caches. The BAM's caches are direct-mapped but unlike the MIPS RC3230 they are write-back which prevents the BAM making memory access in consecutive cycles [Van90a]. For practical reasons the BAM omitted some features, such as byte and half-word memory operations and support for floating-point operations, which are found in the MIPS and similar general-purpose machines.

Moreover the results for the BAM are also interesting because they are produced using Peter Van Roy's global analysing compiler [Van90]. The abstract domain used by Van Roy's compiler is simpler and is effectively a subset of Parma's abstract domain. Although some of the techniques Van Roy employs are not found in Parma and vice-versa the great similarities in approach allow a very useful comparison to be made.

	BAM / Parma no analysis		BAM / Parma	
program	speed	size	speed	size
nreverse	3.62	1.09	0.99	1.34
tak	1.00	0.69	1.01	0.70
qsort	2.46	1.18	1.01	1.28
serialise	0.76	1.02	0.50	1.67
queens_8	2.32	1.13	1.09	1.35
mu	1.44	1.31	0.80	1.49
zebra	1.93	1.19	1.28	1.03
deriv	2.52	4.53	0.73	2.78
crypt	2.17	0.64	0.71	1.22
query	0.96	0.76	0.74	1.68
prover	1.54	1.80	1.00	0.98
poly_10	1.51	1.43	0.94	1.44
browse	1.44	0.77	0.72	0.91
reducer	1.67	1.27	1.26	1.12
boyer	1.33	1.56	1.24	0.90
nand	2.03	1.32	0.97	1.07
chat_parser	1.21	1.21	0.78	1.21

TABLE 5.4. BAM versus Parma

Table 5.4 contains the execution times and code sizes for each of the benchmarks on the BAM compiled by Van Roy's compiler both with and without global analysis. The results have been scaled relative to Parma with and without global analysis respectively. The measurements we have are based on simulations of a 30 MHz BAM chip. As the execution times for Parma were measured for a 25MHz MIPS RC3230 we have scaled them downward by a factor of 1.2 to compensate for the difference. This multiplication of the CPU speed ignores the effect of the performance of the memory sub-system but the factor is small so we believe the result is still more reasonable than a raw comparison.

It could be argued that an even faster clock speed should be assumed for the RC3230 as its instruction set is effectively simpler than that of the BAM and hence likely to be produced at a higher clock speed than the BAM for various pragmatic reasons. However, the designers of the BAM have invested considerable effort in ensuring that the extra Prolog instructions have minimal impact on the speed of the general purpose instructions so the difference may be quite small.

The BAM clearly outperforms Parma when global analysis is not used for either, producing faster execution time and smaller code size. This is almost certainly a consequence of the special-purpose instructions of the BAM. These allow operations such as dereferencing and trailing, which without global analysis are pervasive, to be more cheaply and concisely implemented.

The performance of Parma and the BAM become similar when global analysis is employed for both. The BAM still produces slightly smaller code. However overall execution times seem similar although with significant variation. This variation results from the conflicting effects in each benchmark of several important factors:

- The additional information that Parma's more complex abstract domain yields may allow more efficient code to be produced. For example, this is the main reason that Parma executes the *serialise* benchmark twice as fast.

- The BAM's special-purpose instructions can provide a significant advantage for some programs. For example, this is the main reason for the BAM's better performance on the *zebra* benchmark.

- Van Roy's compiler and Parma employ some different techniques. The BAM's slightly better performance on the *tak* benchmark is because of a technique Van Roy's compiler uses to return values in registers.

5.2

In the larger benchmarks all three of the above factors are acting to some degree and it is difficult to apportion their effects. It is likely that the merging of Parma and Van Roy's compiler could produce significantly better performance from the BAM.

In [Hol90] the benefits of each of the BAM special-purpose instructions is assessed with respect to Van Roy's compiler. The benefits of these instructions would be significantly different if Parma was used. However, it is not clear that the overall performance benefit of the BAM's architectural support for Prolog would be any less.

5.3 PARMA VERSUS THE KCM

The extended general-purpose machine approach of the BAM is not typical of Prolog hardware. Most of the work on Prolog hardware has been devoted to complex instruction sets designed solely for Prolog, such as the WAM. We have used measurements from one such piece of hardware, the Knowledge Crunching Machine (KCM) [Ben89]. The KCM is a piece of special-purpose Prolog hardware designed to be used as a back-end processor for a Unix workstation. It is a 64-bit microcoded architecture based on a version of the WAM. It is implemented in TTL/CMOS with an 80 nanosecond (12.5 MHz) clock cycle. The KCM has separate instruction and data caches each of 64 kilobytes(8k words). The data cache is write-back and direct-mapped but into zones. The instruction is write-through and direct-mapped. The KCM was chosen because it was the highest performance WAM-based hardware for which we had useful measurements.

We have measurements of the KCM's performance for 10 small programs. Fortunately, eight of the programs are versions of programs in the benchmark set we have used in this thesis. The versions are sufficiently different to affect measurements slightly but in character they can be considered identical for our purposes. The two other programs, *con* 6 and *hanoi*, are very small determinate programs. The ten benchmark programs are listed in full in [Tay90]. Table 5.5 contains the execution times and code sizes for each of the benchmarks on the KCM scaled relative to Parma. As the 12.5 MHz clock speed of the KCM is considerably slower than the 25MHz clock speed of the MIPS RC3230 a raw comparison is probably not reasonable. It is hard to formulate a fair comparison because of the great difference between the KCM and the MIPS RC3230 in both architecture and

	KCM / Parma	
program	speed	size
con6	0.62	1.53
hanoi	0.43	1.26
nrev1 (nreverse)	0.49	1.63
qs4(qsort)	0.32	3.00
pri2	0.37	2.61
diff (deriv)	0.82	?
query	0.59	1.33
palin25 (serialise)	1.01	1.36
mutest (mu)	0.91	0.84
queens (queens_8)	0.52	1.52

TABLE 5.5. KCM versus Parma

implementation technology. We have reduced the KCM times by a factor of two to compensate for the difference. In terms of processor technology this probably is generous to the KCM, especially as it ignores the memory sub-system. However the KCM's 64-bit architecture allows it to provide a little more functionality than Parma. Our scaling is an attempt to make the best of what is inherently an inexact comparison.

Although the results in Table 5.5 make the KCM scaled execution times seem slower and its code size larger than Parma we feel both are only consequences of the small size of the benchmarks. We believe that the KCM's performance on larger programs would be better just as, relative to Parma, the performance of SICStus(compiled) in both execution and code size is better on the larger benchmarks. The conclusion we draw is that, after scaling to the same clock speed, the KCM offers similar performance in both execution time and code size to Parma on the MIPS RC3230.

5.4 EVALUATION OF FEATURES

In the last part of our evaluation of Parma's performance we wish to ascertain that the inclusion of several techniques employed by Parma was worthwhile. This is made much easier because these features are unlikely to have negative effects on the code produced. Their costs are only in slowing compilation and increasing Parma's complexity. This can be assessed informally. We have evaluated the benefits of three features of Parma's abstract domain. Two are the sub-features of the variable symbol which represents reference chain and trailing information. The third feature

	reference chain analysis	trailing analysis	structure/list analysis	uninitialised variables
nreverse	2.31	2.16	1.34	1.35
tak	1.00	1.00	1.00	1.24
qsort	1.52	1.00	1.38	1.29
pri2	1.17	1.10	1.05	1.07
serialise	1.08	1.11	1.63	1.08
queens_8	1.07	1.03	1.29	1.10
mu	1.05	1.16	1.51	1.16
zebra	1.05	1.00	1.34	1.00
deriv	1.25	1.44	1.00	1.15
crypt	1.12	1.09	1.17	1.07
query	1.00	1.00	1.00	1.18
prover	1.06	1.13	1.26	1.03
poly_10	1.48	1.07	1.09	1.23
browse	1.14	1.07	1.48	1.12
press1	1.12	1.11	1.58	1.19
reducer	1.05	1.03	1.00	1.13
boyer	1.06	1.07	1.02	1.07
nand	1.17	1.05	1.29	1.15
chat_parser	1.11	1.09	1.19	1.02

TABLE 5.6. Parma - Evaluation of Individual Features

is the inclusion of compound *list* and *term* symbols to allow exact representation of compound terms. We would have preferred to separately evaluate the benefits of the *list* and *term* symbols but this was not possible as their implementation was too closely intertwined. We have also evaluated the benefits gained by the compilation phase designating some predicate argument positions uninitialised (Section 4.5.1).

We measured the benefit from each feature by compiling each benchmark with the feature disabled. Table 5.6 contains this time scaled with respect to the normal execution time. In effect this is the speed-up factor offered by this feature for that benchmark. As this is only the marginal benefit, comparisons must be made carefully. The combined marginal benefit of two features need not be, and probably will not be, their product. It could be less because the features co-operate or it could be more because the features compete. For example, uninitialised argument variables do not need trailing when bound but this would also often be determined by trailing analysis so these two features compete. Removing either of these features may have little effect because the other takes up the slack but removing both features would have a significant effect. Therefore their combined marginal benefit is larger than suggested by the figures in Table 5.6.

The measurements demonstrate that all four features produce useful improvements in over half the benchmark programs. The improvement factors considered alone may not seem large but they are each significant components of Parma's high performance. As can by comparing Table 5.6 to Table 5.2 these four features by no means accounts for all of Parma's high performance. Much comes from the type information global analysis yields. This gain is not surprising as it is analagous to the benefit some existing implementations obtain with the information from user's mode declarations.

Our benchmark set hasa glaringly artificial characteristic: the input to each program is contained in the program. So for example, in the benchmark *nreverse* the predicate *nreverse*/2 is called thus:

```
main :- nreverse([1,2,3 ... 29,30], _).
```

Some of the benefits of analysis will be lost if it is called thus:

```
main :- read(X), nreverse(X, _).
```

However the user can rectify this loss with a single declaration indicating that the first argument to *nreverse*/2 will always be a list of integer. This will result in exactly the same code being produced for the predicate *nreverse*/2 as in the original benchmark. It may however be necessary (see Section 3.4.2) to add code after the call to *read*/1 to ensure that the user declaration is correct. This, if necessary, will add less than 5% to the execution time of the benchmark.

6 CONCLUSION

The importance of the work presented in this thesis lies in its span. We began by dirtying our hands on delay slots, bit fields and such-like at the face of the raw machine and continued upward until we reached the purity of the lattices and fixpoints of abstract analysis. The bottom-up approach ensured that our research was demand-driven. We knew what information was most desirable before we constructed the analysis phase. The full span of the work allowed concrete assessment to be made of the ideas it incorporated.

The greatest weakness of this work lies also in its extent. Parma is not a complete Prolog implementation or even a complete Prolog compiler. This is a disappointment. We were brought to computer science and this work by the joy in constructing useful programs. We have only produced a vehicle to test ideas. It was not for lack of desire. The claims of this thesis will not be truly tested until many users have had the opportunity to run their programs on an implementation employing them. Too many false paths, too many re-designs and too many questions to be answered left us with insufficient resources to complete the journey. We must be content with some interesting discoveries.

6.1 CONTRIBUTIONS

In Chapter 2 we attempted a systematic review of the choices in data representation for a Prolog implementation. Although there has been little published on this subject, the ideas discussed are unlikely to be original. The exception perhaps is the technique of performing arithmetic directly on the tagged representation of integers (Section 2.7). Although this technique does not appear to have been previously employed for Prolog prior to this work it is so useful that it may have been employed in other languages.

In Chapter 3 we described an abstract interpretation-based global analysis phase which yields information useful for compilation. In this lies the most important single contribution of this thesis: the discovery that the representation of reference chains and free variable trailing status in an abstract domain allows information very useful to a Prolog compiler to be determined by abstract execution. It is interesting to note that these are not natural elements of a Prolog program but rather implementation artifacts. The abstract representation of reference chains was suggested independently in [Mar89].

In the *list* and *term* compound abstract domain symbols we have demonstrated a feasible if ad hoc technique of obtaining more precise information regarding compound terms. Perhaps more importantly, we have discovered that this information can have significant benefits and hence encourage more refined approaches to the representation of compound terms.

We have shown that abstract execution over a complex abstract domain is not only feasible even for very large programs but can be done in a time which should be acceptable to most users.

Chapter 4 may describe Parma's heart but there is little that is exciting among the very necessary description of the pragmatics of compilation. One exception is the convention by which Parma can designate some predicate argument positions uninitialised. The concept itself is not surprising for it is analogous to constructs in imperative languages such as Pascal's *var* parameters. It is the size of the benefits it yields which is surprising.

The measurements of Chapter 5 are the fruition of this work. They reveal that, in combination, the techniques we have developed produce formidable performance, far exceeding previous software implementations of Prolog. This performance rivals that of the best special-purpose Prolog hardware which has been built.

6.2 FUTURE WORK

There are many avenues for further work in this area.

Although we were unable to improve on the data representation used by the WAM, we feel that all the possibilities have not been exhausted. It may be possible for a compiler employing global analysis to obtain fundamental gains by using another data representation. In particular, it may be possible to improve on the WAM's representation of aliased variables (Section 2.6).

During the implementation of Parma's analysis phase several compromises were made to ensure the analysis of large programs could be done in reasonable time. These compromises were probably unduly conservative. Close examination of these compromises should be fruitful. There surely must be abstract control algorithms (Section 3.5) which yield more precision but still execute large programs in reasonable time over Parma's abstract domain. It also seems likely that the ad hoc solution to the representation of compound terms in Parma's abstract domain (Section 3.3) can be improved upon.

More fundamentally, characterisation of the program features which cause Parma's global analysis to perform poorly, such as the *boyer* benchmark, followed by examination of techniques to remedy the imprecision would be valuable.

Parma's source-to-source transformations are not ambitious. There is great scope for improving Prolog performance through more ambitious source-level transformations. However, ensuring the safety and desirability of such transformations will be non-trivial.

It is difficult to anticipate the directions in which useful improvements could be made to Parma's compilation phase. We hope that we have been sufficiently thorough that further additions to Parma's compilation phase will either involve only small gains or great effort. There are two promising possibilities. The re-use of obsolete compound terms should produce significant gains for some programs (Section 2.3). The main obstacle is adding suitable information to the abstract domain to discover the redundant terms without making analysis time impractical. Global (inter-predicate) register allocation also looks promising on register-rich architectures such as the MIPS.

Expanding the set of benchmark programs measured in Chapter 5 could increase our confidence in Parma's results. It would be much more valuable to incorporate Parma's techniques in a full Prolog implementation and discover the results real users obtain.

The inclusion of Parma's techniques in a compiler designed to make re-targeting more feasible would yield very useful information on the scope of these techniques.

Until this is done it is difficult to assess the utility of these techniques and the importance of global analysis in general on architectures which differ greatly from the MIPS such as, for example, the widely used Motorola 68000 architecture.

6.3 IMPLICATIONS

Prolog implementations for the MIPS and hopefully many other architectures can greatly boost their performance by including a compiler employing Parma's techniques. Such a compiler could be tacked on to an implementation such as SICStus Prolog. This would give three performance tiers: interpretation, compilation to an emulated abstract machine and compilation to native-code. The improving performance of workstations is reducing the need for the middle tier: compilation to abstract machine code. Interpretation is becoming sufficiently fast for a great deal of debugging and development. It is possible that a compiler such as Parma combined with an interpreter would form a sufficient basis for a Prolog environment. However compilation to an emulated abstract machine retains the great advantage of easy portability. Inherent lack of portability is surely the largest obstacle to the spread of Prolog compilers such as Parma.

This work holds serious implications for the future of special-purpose Prolog hardware. The high performance of Parma and the MIPS must bring a re-assessment of the value of special-purpose Prolog hardware. The thesis of Tep Dobry's work [Dob87] in constructing the Berkeley PLM was:

"An additional ten fold improvement in performance can be realised for sequential execution of Prolog programs over implementations on general-purpose processors by a processor specifically designed and tuned to the Prolog task."

Not only does Parma out-perform the PLM by a considerable margin but it also matches the performance of faster special purpose hardware, the KCM and the BAM, built since the PLM. The immediate prospects for Prolog hardware with high-level instruction sets, such as the KCM, seem dim. Of course, this may change with the development of new hardware technology. Some special-purpose hardware, such as the BAM, can benefit from the techniques employed in Parma and can certainly still offer a performance advantage but for many programs the advantage will be slight. It seems likely that only a few programs will gain more than a factor of two performance improvement. It is difficult to see this performance advantage justifying the production of special-purpose Prolog hardware in the current technological and commercial environment. The aggressive pursuit of the edge of technology by RISC manufacturers such as MIPS Computer Systems, Inc. will force Prolog hardware to chase a fast-moving target. Although the context may change, research on Prolog hardware certainly remains an important field. As the fraction of their user base formed by Prolog users increases, the designers of general-purpose machines wishing to improve the performance of Prolog on their machines will look to such research.

Although this work has focused completely on Prolog it does have implications for other languages. Bottom-up approaches are anathema to many in computer science. However it is important that there should be a bottom-up element in the construction of a high-performance compiler. The development of improving transformations should be demand-driven.

Parma clearly demonstrates that it is possible for sufficiently sophisticated compilation techniques to bridge a large gap between language and machine. This should relax the shackles of efficiency and allow the programmer to strive more for elegance.

7 REFERENCES

[Aho79] A. Aho, B. W. Kernighan and P. J. Weinberger, ‘‘Awk - a Pattern Scanning and Processing Language’’, *Software - Practice and Experience*, 9:4, 267-280.

[Aho86] A. Aho, R. Sethi and J. Ullman, *Compilers, Principles, Techniques and Tools*, Addison-Wesley(1986).

[App87] K. Appleby, S. Haridi and Dan Sahlin, ‘‘Garbage Collection for Prolog Based on WAM’’, IBM Research Technical Report RC 12941 (#57635), March 87.

[Bat73] G. Battani and H Meloni, ‘‘Interpreteur du langage de programmation Prolog’’, Groupe d’Intelligence Artificielle, Marseille-Luminy, 1973.

[Ben89] H. Benker et al., ‘‘KCM: A Knowledge Crunching Machine’’, *16th Annual International Symposium on Computer Architecture* Jerusalem, May 1989.

[Bow86] K. Bowen, K Buettner, I Cicekli and A. Turk, ‘‘The design and implementation of a high-speed incremental portable Prolog compiler’’, *3rd International Conference on Logic Programming*, London, July 1986.

[Bru76] M. Bruynooghe, ‘‘An interpereter for predicate logic programs’’, Report CW 10, Applied Maths and Programming Division, Katholieke Universiteit, Belguim, October 1976.

[Bru87] M. Bruynooghe, ‘‘A Framework for the Abstract Interpretation of Logic Programs’’, Research report 62, Katholieke Universiteit, Leuven, 1987.

[Bru87a] M Bruynooghe et al., ‘‘Abstract Interpretation: Towards the Global Optimization of Prolog Programs’’, *4th IEEE Symposium on Logic Programming*, San Francisco, September 1987.

[Car88] M. Carlsson and J. Widen, ‘‘SICStus Prolog Users Manual’’, SICS Research Report R88007B, October 1988.

[Car89] M. Carlsson, ‘‘On the Efficiency of Optimising Shallow Backtracking in Compiled Prolog’’, *Sixth International Conference on Logic Programming*, Lisbon, July 1989.

[Cit88] W. Citrin and A. Despain, ‘‘Scheduling of Parallel Unification Operations through Static Data-Dependency Analysis’’. *5th International Conference* *on Logic Programming*, Seattle, August 1988.

[Col86] R. Colomb and Jayasooriah, ‘‘A Clause Indexing System for Prolog Based on Superimposed Coding’’, *Australian Computer Journal*, vol. 18, no. 1, pp. 18-25.

[Cou77] P. Cousot and R. Cousot, ‘‘Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction of Fixed Points’’, *4th ACM Symposium on Principles of Programming Languages*, pp. 78-88, 1977.

[Deb85] S. K. Debray, ‘‘Efficient Register Allocation for Temporary Variables in the Warren Prolog Engine’’, Research Paper 85/10, Department of Computer Science, SUNY at Stony Brook, April 1985.

[Deb86] S. K. Debray and D. S. Warren, ‘‘Automatic Mode Inference for Prolog Programs’’, *1986 International Symposium on Logic Programming*, Salt Lake City, pp. 78-88, September 1986.

[Deb86a] S. K. Debray and D. S. Warren, ‘‘Detection and Optimization of Functional Computations’’, *3rd International Conference on Logic Programming*, London, July 1986.

[Deb87] S. K. Debray, ‘‘Static Inference of Modes and Data Dependencies in Logic Programs’’, Technical Report 87-15, Dept of Computer Science, University of Arizona, Tucson, March 1988.

[Deb87a] S. K. Debray, ‘‘Flow Analysis of a Simple Class of Dynamic Logic Programs’’, *4th IEEE Symposium on Logic Programming*, San Francisco, September 1987.

[Deb89] S. K. Debray, ‘‘A Simple Code Improvement Scheme for Prolog’’, *Sixth International Conference on Logic Programming*, Lisbon, July 1989.

[Dem89] B. Demoen, A. Marien and A Callebaut, ‘‘Indexing Prolog Clauses’’, *North American Conference on Logic Programming*, October 1989.

[Dob87] T. P. Dobry, ‘‘A High Performance Architecture for Prolog’’, Ph.D. Thesis, Computer Science Division, TR UCB/CS 87/352, University of California, Berkeley, April 1987.

[Hen90] D. Patterson and J. Hennessy, *Computer Architecture A Quantitative Approach*, Morgan Kaufman(1990).

[Hic87] T. Hickey and S. Mudambi, ‘‘Global Compilation of Prolog’’, *Journal of Logic Programming*, Vol. 9, No. 3, November 1989.

[Hil86] M. Hill et al., ‘‘Design decisions in SPUR’’, *Computer 19*, pp. 8-22, November 1986.

[Hol90] B. K. Holmer et al., ‘‘Fast Prolog with and Extended General purpose Architecture’’, *17th International Symposium on Computer Architecture*, May 1990.

[IBM86] The RT-PC Processor Technical Reference, IBM 1986.

[Kan88] G. Kane, *MIPS RISC Architecture*, Prentice-Hall, Englewood Cliffs NJ, 1988.

[Knu73] D. Knuth, *Sorting and Searching*, Addison-Wesley(1973).

[Li86] K. Li and P. Hudak, ‘‘A new List Compaction Method’’, *Software - Practice and Experience*, V16(2), pp. 145-163, February 1986.

[Klu87] F. Kluzniak, ‘‘Type Synthesis for Ground Prolog’’, *4th International Conference on Logic Programming*, Melbourne, May 1987.

[Llo84] J. Lloyd, *Foundations of Logic Programming*, Springer-Verlag(1984).

[Mad89] V.S. Madan, C.-J. Peng and G. S. Sohi, ‘‘On the Adequacy of Direct Mapped Caches for Lisp and Prolog Data References Patterns’’, Proceedings of the North American Conference on Logic Programming, October 1989.

[Man87] H. Mannila and E. Ukkonen, ‘‘Flow Analysis of Prolog Programs’’, *4th IEEE Symposium on Logic Programming*, San Francisco, September 1987.

[Mar89] A Marien et al., ‘‘The impact of abstract interpretation: an experiment in code generation’’, *Sixth International Conference on Logic Programming*, Lisbon, July 1989.

[Mar89a] A. Marien and B. Demoen, ‘‘On the Management of Choicepoint and Environment frames in the WAM’’, *North American Conference on Logic Programming*, October 1989.

[Mel85] C. S. Mellish, ‘‘Some Global Optimizations For a Prolog Compiler’’, *Journal of Logic Programming*, Vol. 2(1), pp. 43-66, 1987.

[Mil89] J. W. Mills, ‘‘A High-Performance LOW RISC Machine for Logic Programming’’, *Journal of Logic Programming*, Vol. 6(1&2), pp. 186-194, 1989.

[MIP91] Advance Technical Information, R4000 Architecture Overview, February 1991.

[Mor86] K. Morris, J. Ullman and A. Van Gelder, ‘‘Design Overview of the NAIL! System’’, *Third International Conference on Logic Programming*, London, July 1986.

[Muk90] K. Muthukumar and M. Hermengildo, ‘‘The DCG, UDG and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-Parallelism’’, *Seventh International Conference on Logic Programming*, Jerusalem July 1990.

[Mul87] J. Mulder and E. Tick, ‘‘A Performance Comparison Between PLM and an MC68020 Prolog Processor’’, *4th International Conference on Logic Programming*, Melbourne, May 1987.

[Nak87] H. Nakashime and K. Nakajima, ‘‘Hardware Architecture of Sequential Machine: PSI-II’’, *4th IEEE Symposium on Logic Programming*, San Francisco, September 1987.

[Pla84], D. Plaisted, ‘‘The Occur-Check Problem in Prolog’’, *International Symposium on Logic Programming*, Atlantic City, 1984.

[Ram86] K. Ramamohanarao and J. Shepherd, ‘‘A Superimposed Codeword Scheme for Very Large Prolog Databases’’ *Third International Conference on Logic Programming*, London, July 1986.

[Rou72] P Roussel, ‘‘Definition et traitement de l’egalite formelle en demonstration automatique’’, These 3me. cycle, UER de Luminy, Marseille, 1972.

[Sic87] ‘‘SICStus Prolog Users Manual’’, Swedish Institute of Computer Science, Sweden, September 1987.

[Smi85] D. Smith and M. Genesereth, ‘‘Ordering Conjunctive Queries’’, *Artificial Intelligence*, pp. 171-215, 26(1985).

[Ste87] L. Sterling and U. Shapiro, *The Art of Prolog*, MIT Press(1986).

[Ste87a] P. Steenkiste and J. Hennessy, ‘‘Tags and Type Checking in LISP: Hardware and Software Approaches’’, *2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 50-59, October 1987.

[Sze77] P. Szeredi, ‘‘Prolog - a very high level language based on predicate logic’’, *2nd Hungarian Conference on Computer Science*, Budapest, January 1977.

[Tay89] A. Taylor, ‘‘Removal of Dereferencing and Trailing in Prolog Compilation’’, *Sixth International Conference on Logic Programming*, Lisbon, July 1989.

[Tay90] A. Taylor, ‘‘LIPS on a MIPS: Results from a Prolog Compiler for a RISC’’, *Seventh International Conference on Logic Programming*, Jerusalem, July 1990.

[Tay91] A. Taylor, ‘‘A High Performance Prolog on a RISC’’, *New Generation Computing*, (to appear late 1991).

[Tay91a] A. Taylor, ‘‘High Performance Prolog Implementation Through Global Analysis’’, *International Workshop on Processing of Declarative Knowledge*, Kaiserslautern, July 1991.

[Tic84] E. Tick and D.H.D. Warren, ‘‘Towards a Pipelined Prolog Processor’’, *International Symposium of Logic Programming*, pp. 29-40, 1984.

[Tic87] ‘‘Memory Performance of Prolog Architectures’’, Technical Report CSL TR-87-329, Computer System Laboratory, Stanford University, June 1987

[Tou87] H. Touati and A Despain, ‘‘An Empirical Study of The Warren Abstract Machine’’, *4th IEEE Symposium on Logic Programming*, San Francisco, September 1987.

[Van84], P. Van Roy, ‘‘A Prolog Compiler for the PLM’’, Technical Report UCB/CSD 84/203, University of California at Berkeley, August 1984.

[Van87] P. Van Roy and B. Demoen, ‘‘Improving the Execution Speed of Compiled Prolog with Modes, Clause Selection and determinism’’, TAPSOFT ’87, *Lecture Notes in Computer Science*, pp. 111-125, March 1987.

[Van89] P. Van Roy, ‘‘An Intermediate Language to Support Prolog Unification’’, *North American Conference on Logic Programming*, October 1989.

[Van90] P. Van Roy and A. Despain, ‘‘The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler’’, *North American Conference on Logic Programming*, October 1990.

[Van90a] Peter van Roy, Personal communication.

[War77] D.H.D. Warren ‘‘Applied Logic - Its Use and Implementation as a Programming Tool’’, Ph.D. Thesis, Univ. Edinburgh, Scotland, 1977.

[War83] D.H.D. Warren ‘‘An Abstract Prolog Instruction Set’’, Technical Note 309, SRI International, Menlo Park, California, 1983.

[War88] R. Warren, M. Hermenegildo and S. K. Debray, ‘‘On the Practicality of Global Flow Analysis of Logic Programs’’, *5th International Conference on Logic Programming*, Seattle, pp. 684-699, August 1988.

[Wis84] M. Wise and D. Powers, ‘‘Indexing Prolog Clauses via Superimposed Code Words and Field Encoded Words’’, *Proceedings of International Symposium on Logic Programming*, IEEE Computer Society, pp. 203-210, 1984.

[Zob87] J. Zobel, ’’Derivation of Polymorphic Types for Prolog Programs’’, *4th International Conference on Logic Programming*, Melbourne, May 1987.

7.

APPENDICES

A ORDERING HASH TABLES

We have a number of hash tables containing atoms. The size of each table is a power of two. The hashing function if the atom's index value modulo the hash table size. We can choose the index value used to represent each atom. We would like assign index values to atoms in such a way that no hashing collisions occur in any of the tables. It can be shown than determining if such an assignment exists is NP-hard by transforming the problem of determining if a graph has a 4-colouring to it. In the style of [Gar79].

Instance: a finite set *S*, a collection of subset $C_1 \cdots C_j$.

Problem: Is there a function $f : S \rightarrow Z^+$ such that

$\forall\, i,\ x,\ y : x \in C_i,\ y \in C_i,\ X \ mod\ 2^{\lceil \log_2(|C_i|) \rceil} \neq Y\ mod\ 2^{\lceil \log_2(|C_i|) \rceil}$

Transformation: given a graph $G=(V,E)$

> let $S = \{ 1 \cdots |V|+|E| \}$
> $i = 1$
> for $(X,Y) \in E$ do
>   let $C_i = \{ X,\ Y,\ |V| + i \}$

There is a 4-colouring of *G* iff there exists a function *f*.

REFERENCES

[Gar79] M. Garey and D. Johnson, *Computers and Intractability:a Guide to the Theory of NP-Completeness*, Freeman(1979).

B BENCHMARKS

```
%
%   nreverse
%
%   David H. D. Warren
%
%   "naive"-reverse a list of 30 integers

main :- nreverse([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30],_).

nreverse([X|L0],L) :- nreverse(L0,L1), concatenate(L1,[X],L).
nreverse([],[]).

concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
concatenate([],L,L).

%
%   tak
%
%   Evan Tick (from Lisp version by R. P. Gabriel)
%
%   (almost) Takeuchi function (recursive arithmetic)

main :- tak(18,12,6,_).

tak(X,Y,Z,A) :-
        X =< Y,
        Z = A.
tak(X,Y,Z,A) :-
        X > Y,
        X1 is X - 1,
        tak(X1,Y,Z,A1),
        Y1 is Y - 1,
        tak(Y1,Z,X,A2),
        Z1 is Z - 1,
        tak(Z1,X,Y,A3),
        tak(A1,A2,A3,A).

%
%   qsort
%
%   David H. D. Warren
%
%   quicksort a list of 50 integers

main :- qsort([27,74,17,33,94,18,46,83,65, 2,32,53,28,85,99,47,28,82, 6,11,55,29,39,81,90,37,10, 0,66,
                51,7,21,85,27,31,63,75, 4,95,99,11,28,61,74,18,92,40,53,59, 8],_,[]).

qsort([X|L],R,R0) :-
        partition(L,X,L1,L2),
        qsort(L2,R1,R0),
        qsort(L1,R,[X|R1]).
qsort([],R,R).

partition([X|L],Y,[X|L1],L2) :-
        X =< Y, !,
        partition(L,Y,L1,L2).
partition([X|L],Y,L1,[X|L2]) :-
        partition(L,Y,L1,L2).
partition([],_,[],[]).
```

```
%
%       pri2
%
main :-
        primes(98, X).

primes(Limit, Ps) :-
        integers(2, Limit, Is),
        sift(Is, Ps).

integers(Low, High, [Low | Rest]) :-
        Low =< High,
        !,
        M is Low+1,
        integers(M, High, Rest).
integers(_,_,[]).

sift([],[]).
sift([I | Is], [I | Ps]) :-
        remove(I,Is,New),
        sift(New, Ps).

remove(P,[],[]).
remove(P,[I | Is], [I | Nis]) :-
        0 == I mod P,
        !,
        remove(P,Is,Nis).
remove(P,[I | Is], Nis) :-
        0 =:= I mod P,
        !,
        remove(P,Is,Nis).

%
%   serialise
%
%   David H. D. Warren
%
%   itemize (pick a "serial number" for each
%   unique integer in) a list of 25 integers

main :- serialise("ABLE WAS I ERE I SAW ELBA",_).

serialise(L,R) :-
        pairlists(L,R,A),
        arrange(A,T),
        numbered(T,1,_).

pairlists([X|L],[Y|R],[pair(X,Y)|A]) :- pairlists(L,R,A).
pairlists([],[],[]).

arrange([X|L],tree(T1,X,T2)) :-
        split(L,X,L1,L2),
        arrange(L1,T1),
        arrange(L2,T2).
arrange([],void).

split([X|L],X,L1,L2) :- !, split(L,X,L1,L2).
split([X|L],Y,[X|L1],L2) :- before(X,Y), !, split(L,Y,L1,L2).
split([X|L],Y,L1,[X|L2]) :- before(Y,X), !, split(L,Y,L1,L2).
split([],_,[],[]).

before(pair(X1,_),pair(X2,_)) :- X1 < X2.

numbered(tree(T1,pair(_,N1),T2),N0,N) :-
        numbered(T1,N0,N1),
        N2 is N1+1,
        numbered(T2,N2,N).
numbered(void,N,N).
```

```
%
%   queens_8
%
%   from Sterling and Shapiro, "The Art of Prolog," page 211.
%
%   This program solves the N queens problem:  place N pieces on an N
%   by N rectangular board so that no two pieces are on the same line
%   - horizontal, vertical, or diagonal.  (N queens so placed on an N
%   by N chessboard are unable to attack each other in a single move
%   under the rules of chess.)  The strategy is incremental generate-
%   and-test.
%
%   A solution is specified by a permutation of the list of numbers 1 to
%   N.  The first element of the list is the row number for the queen in
%   the first column, the second element is the row number for the queen
%   in the second column, et cetera.  This scheme implicitly incorporates
%   the observation that any solution of the problem has exactly one
%   queen in each column.
%
%   The program distinguishes symmetric solutions.  For example,
%
%   ?- queens(4, Qs). produces Qs = [3,1,4,2] ; Qs = [2,4,1,3]

main :- queens(8,_), !.

queens(N,Qs) :-
        range(1,N,Ns),
        queens(Ns,[],Qs).

queens([],Qs,Qs).
queens(UnplacedQs,SafeQs,Qs) :-
        select(UnplacedQs,UnplacedQs1,Q),
        not_attack(SafeQs,Q),
        queens(UnplacedQs1,[Q|SafeQs],Qs).

not_attack(Xs,X) :-
        not_attack(Xs,X,1).

not_attack([],_,_) :- !.
not_attack([Y|Ys],X,N) :-
        X == Y+N, X == Y-N,
        N1 is N+1,
        not_attack(Ys,X,N1).

select([X|Xs],Xs,X).
select([Y|Ys],[Y|Zs],X) :- select(Ys,Zs,X).

range(N,N,[N]) :- !.
range(M,N,[M|Ns]) :-
        M < N,
        M1 is M+1,
        range(M1,N,Ns).
```

```
%
% mu
%
% derived from Douglas R. Hofstadter,
% "Godel, Escher, Bach," pages 33-35.
%
% prove "mu-math" theorem muiiu

main :- theorem([m,u,i,i,u], 5, _), !.

theorem([m,i], _, [[a|[m,i]]]).
theorem(R, Depth, [[N|R]|P]) :-
        Depth > 0,
        D is Depth-1,
        theorem(S, D, P),
        rule(N, S, R).

rule(1, S, R) :- rule1(S, R).
rule(2, S, R) :- rule2(S, R).
rule(3, S, R) :- rule3(S, R).
rule(4, S, R) :- rule4(S, R).

rule1([i], [i,u]).
rule1([H|X], [H|Y]) :-
        rule1(X, Y).

rule2([m|X], [m|Y]) :-
        append(X, X, Y).

rule3([i,i,i|X], [u|X]).
rule3([H|X], [H|Y]) :-
        rule3(X, Y).

rule4([u,u|X], X).
rule4([H|X], [H|Y]) :-
        rule4(X, Y).

append([], X, X).
append([A|B], X, [A|B1]) :-
        append(B, X, B1).

% deriv
%
% David H. D. Warren
%

main :- ops8, divide10, log10, times10.

d(U+V,X,DU+DV) :- !,
        d(U,X,DU),
        d(V,X,DV).
d(U-V,X,DU-DV) :- !,
        d(U,X,DU),
        d(V,X,DV).
d(U*V,X,DU*V+U*DV) :- !,
        d(U,X,DU),
        d(V,X,DV).
d(U/V,X,(DU*V-U*DV)/(^(V,2))) :- !,
        d(U,X,DU),
        d(V,X,DV).
d(^(U,N),X,DU*N*(^(U,N1))) :- !,
        integer(N),
        N1 is N-1,
        d(U,X,DU).
d(-U,X,-DU) :- !,
        d(U,X,DU).
d(exp(U),X,exp(U)*DU) :- !,
        d(U,X,DU).
d(log(U),X,DU/U) :- !,
        d(U,X,DU).
d(X,X,1) :- !.
d(_,_,0).

ops8 :- d((x+1)*((^(x,2)+2)*(^(x,3)+3)),x,_).
divide10 :- d(((((((((x/x)/x)/x)/x)/x)/x)/x)/x)/x,x,_).
log10 :- d(log(log(log(log(log(log(log(log(log(log(x)))))))))),x,_).
times10 :- d(((((((((x*x)*x)*x)*x)*x)*x)*x)*x)*x,x,_).

%
% zebra
%
% Where does the zebra live?
% Puzzle solution written by Claude Sammut.
main :-
        houses(Houses),
        member(house(red, english, _, _, _), Houses),
        member(house(_, spanish, dog, _, _), Houses),
        member(house(green, _, _, coffee, _), Houses),
        member(house(_, ukrainian, _, tea, _), Houses),
        right_of(house(green,_,_,_,_), house(ivory,_,_,_,_), Houses),
        member(house(_, _, snails, _, winstons), Houses),
        member(house(yellow, _, _, _, kools), Houses),
        Houses = [_, _, house(_, _, _, milk, _), _,_],
        Houses = [house(_, norwegian, _, _, _)|_],
        next_to(house(_,_,_,_,chesterfields), house(_,_,fox,_,_), Houses),
        next_to(house(_,_,_,_,kools), house(_,_,horse,_,_), Houses),
        member(house(_, _, _, orange_juice, lucky_strikes), Houses),
        member(house(_, japanese, _, _, parliaments), Houses),
        next_to(house(_,norwegian,_,_,_), house(blue,_,_,_,_), Houses),
        member(house(_, _, zebra, _, _), Houses),
        member(house(_, _, _, water, _), Houses),
        print_houses(Houses).

houses([
        house(_, _, _, _, _),
        house(_, _, _, _, _),
        house(_, _, _, _, _),
        house(_, _, _, _, _),
        house(_, _, _, _, _)
]).

right_of(A, B, [B, A | _]).
right_of(A, B, [_ | Y]) :- right_of(A, B, Y).

next_to(A, B, [A, B | _]).
next_to(A, B, [B, A | _]).
next_to(A, B, [_ | Y]) :- next_to(A, B, Y).

member(X, [X|_]).
member(X, [_|Y]) :- member(X, Y).

print_houses([A|B]) :- !,
        write(A), nl,
        print_houses(B).
print_houses([]).
```

B.

```
%
% crypt
%
% Cryptomultiplication:
% Find the unique answer to:
%     OEE
%      EE
%     ---
%     EOEE
%     EOE
%     ----
%     OOEE
%
% where E=even, O=odd.
% This program generalizes easily
% to any such problem.
% Written by Peter Van Roy

main :-
        odd(A), even(B), even(C),
        even(E),
        mult([C,B,A], E, [I,H,G,F|X]),
        lefteven(F), odd(G), even(H),
        even(I), zero(X),
        lefteven(D),
        mult([C,B,A], D, [L,K,J|Y]),
        lefteven(J), odd(K), even(L), zero(Y),
        sum([I,H,G,F], [0,L,K,J], [P,O,N,M|Z]),
        odd(M), odd(N), even(O), even(P), zero(Z).
        % write(' '), write(A), write(B), write(C), nl,
        % write('  '), write(D), write(E), nl,
        % write(F), write(G), write(H), write(I), nl,
        % write(J), write(K), write(L), nl,
        % write(M), write(N), write(O), write(P), nl.

% Addition of two numbers
sum(AL, BL, CL) :- sum(AL, BL, 0, CL).

sum([A|AL], [B|BL], Carry, [C|CL]) :- !,
        X is (A+B+Carry),
        C is X mod 10,
        NewCarry is X // 10,
        sum(AL, BL, NewCarry, CL).
sum([], BL, 0, BL) :- !.
sum(AL, [], 0, AL) :- !.
sum([], [B|BL], Carry, [C|CL]) :- !,
        X is B+Carry,
        NewCarry is X // 10,
        C is X mod 10,
        sum([], BL, NewCarry, CL).
sum([A|AL], [], Carry, [C|CL]) :- !,
        X is A+Carry,
        NewCarry is X // 10,
        C is X mod 10,
        sum([], AL, NewCarry, CL).
sum([], [], Carry, [Carry]).

% Multiplication
mult(AL, D, BL) :- mult(AL, D, 0, BL).

mult([A|AL], D, Carry, [B|BL] ) :-
        X is A*D+Carry,
        B is X mod 10,
        NewCarry is X // 10,
        mult(AL, D, NewCarry, BL).
mult([], _, Carry, [C,Cend]) :-
        C is Carry mod 10,
        Cend is Carry // 10.

zero([]).
zero([0|L]) :- zero(L).

odd(1).
odd(3).
odd(5).
odd(7).
odd(9).

even(0).
even(2).
even(4).
even(6).
even(8).

lefteven(2).
lefteven(4).
lefteven(6).
lefteven(8).
```

```
%
%   query
%
%   David H. D. Warren
%
%   query population and area database to find coun-
%   tries of approximately equal population density

query :- query(_), fail.
query.

query([C1,D1,C2,D2]) :-
        density(C1,D1),
        density(C2,D2),
        D1 > D2,
        T1 is 20*D1,
        T2 is 21*D2,
        T1 < T2.

density(C,D) :-
        pop(C,P),
        area(C,A),
        D is (P*100)//A.

% populations in 100000's % areas in 1000's of square miles
pop(china,        8250).    area(china,3380).
pop(india,        5863).    area(india,1139).
pop(ussr,         2521).    area(ussr,8708).
pop(usa,          2119).    area(usa,3609).
pop(indonesia,    1276).    area(indonesia,  570).
pop(japan,        1097).    area(japan,  148).
pop(brazil,       1042).    area(brazil,3288).
pop(bangladesh,    750).    area(bangladesh,   55).
pop(pakistan,      682).    area(pakistan,  311).
pop(w_germany,     620).    area(w_germany,   96).
pop(nigeria,       613).    area(nigeria,  373).
pop(mexico,        581).    area(mexico,  764).
pop(uk,            559).    area(uk,    86).
pop(italy,         554).    area(italy,  116).
pop(france,        525).    area(france,  213).
pop(philippines,   415).    area(philippines,   90).
pop(thailand,      410).    area(thailand,  200).
pop(turkey,        383).    area(turkey,  296).
pop(egypt,         364).    area(egypt,  386).
pop(spain,         352).    area(spain,  190).
pop(poland,        337).    area(poland,  121).
pop(s_korea,       335).    area(s_korea,   37).
pop(iran,          320).    area(iran,  628).
pop(ethiopia,      272).    area(ethiopia,  350).
pop(argentina,     251).    area(argentina,1080).
```

```
%
%   prover
%
%   Richard A. O'Keefe
%
%   Prolog theorem prover
%
%   from "Prolog Compared with Lisp?,"
%   SIGPLAN Notices, v. 18 #5, May 1983

% op/3 directives

:- op(950, xfy, #).     % disjunction
:- op(850, xfy, &).     % conjunction
:- op(500, fx, +).      % assertion
:- op(500, fx, -).      % denial

prover :- problem(_, P, C),
        implies(P, C),
        fail.
prover.

% problem set

problem( 1, -a, +a).

problem( 2, +a, -a & -a).

problem( 3, -a, +to_be # -to_be).

problem( 4, -a & -a, -a).

problem( 5, -a, +b # -a).

problem( 6, -a & -b, -b & -a).

problem( 7, -a, -b # (+b & -a)).

problem( 8, -a # (-b # +c), -b # (-a # +c)).

problem( 9, -a # +b, (+b & -c) # (-a # +c)).

problem( 10, (-a # +c) & (-b # +c), (-a & -b) # +c).

% Prolog theorem prover

implies(Premise, Conclusion) :-
        opposite(Conclusion, Denial),
        add_conjunction(Premise, Denial, fs([],[],[],[])).

opposite(F0 & G0, F1 # G1) :- !,
        opposite(F0, F1),
        opposite(G0, G1).
opposite(F1 # G1, F0 & G0) :- !,
        opposite(F1, F0),
        opposite(G1, G0).
opposite(+Atom, -Atom) :- !.
opposite(-Atom, +Atom).

add_conjunction(F, G, Set) :-
        expand(F, Set, Mid),
        expand(G, Mid, New),
        refute(New).

expand(_, refuted, refuted) :- !.
expand(F & G, fs(D,_,_,_), refuted) :-
        includes(D, F & G), !.
expand(F & G, fs(D,C,P,N), fs(D,C,P,N)) :-
        includes(C, F & G), !.
expand(F & G, fs(D,C,P,N), New) :- !,
        expand(F, fs(D,[F & G|C],P,N), Mid),
        expand(G, Mid, New).
expand(F # G, fs(D,C,P,N), Set) :- !,
        opposite(F # G, Conj),
        extend(Conj, D, C, D1, fs(D1,C,P,N), Set).
expand(+Atom, fs(D,C,P,N), Set) :- !,
        extend(Atom, P, N, P1, fs(D,C,P1,N), Set).
expand(-Atom, fs(D,C,P,N), Set) :-
        extend(Atom, N, P, N1, fs(D,C,P,N1), Set).

includes([Head|_], Head) :- !.
includes([_|Tail], This) :- includes(Tail, This).

extend(Exp, _, Neg, _, _, refuted) :-
        includes(Neg, Exp), !.
extend(Exp, Pos, _, Pos, Set, Set) :-
        includes(Pos, Exp), !.
extend(Exp, Pos, _, [Exp|Pos], Set, Set).

refute(refuted) :- !.
refute(fs([F1 & G1|D], C, P, N)) :-
        opposite(F1, F0),
        opposite(G1, G0),
        Set = fs(D, C, P, N),
        add_conjunction(F0, G1, Set),
        add_conjunction(F0, G0, Set),
        add_conjunction(F1, G0, Set).
```

B.

```
%
% poly_10
%
% Ralph Haygood (based on Prolog version by Rick McGeer
%              based on Lisp version by R. P. Gabriel)
%
% raise a polynomial (1+x+y+z) to the 10th power (symbolically)

:-op(700,xfx,less_than).

poly_10 :- test_poly(P), poly_exp(10, P, _).

% test polynomial definition

test_poly(P) :-
        poly_add(poly(x,[term(0,1),term(1,1)]),poly(y,[term(1,1)]),Q),
        poly_add(poly(z,[term(1,1)]),Q,P).

% 'less_than'/2 for x, y, z

x less_than y.
y less_than z.
x less_than z.

% polynomial addition

poly_add(poly(Var,Terms1), poly(Var,Terms2), poly(Var,Terms)) :- !,
        term_add(Terms1, Terms2, Terms).
poly_add(poly(Var1,Terms1), poly(Var2,Terms2), poly(Var1,Terms)) :-
        Var1 less_than Var2, !,
        add_to_order_zero_term(Terms1, poly(Var2,Terms2), Terms).
poly_add(Poly, poly(Var,Terms2), poly(Var,Terms)) :- !,
        add_to_order_zero_term(Terms2, Poly, Terms).
poly_add(poly(Var,Terms1), C, poly(Var,Terms)) :- !,
        add_to_order_zero_term(Terms1, C, Terms).
poly_add(C1, C2, C) :-
        C is C1+C2.

% term addition

term_add([], X, X) :- !.
term_add(X, [], X) :- !.
term_add([term(E,C1)|Terms1], [term(E,C2)|Terms2], [term(E,C)|Terms]) :- !,
        poly_add(C1, C2, C),
        term_add(Terms1, Terms2, Terms).
term_add([term(E1,C1)|Terms1], [term(E2,C2)|Terms2], [term(E1,C1)|Terms]) :-
        E1 < E2, !,
        term_add(Terms1, [term(E2,C2)|Terms2], Terms).
term_add(Terms1, [term(E2,C2)|Terms2], [term(E2,C2)|Terms]) :-
        term_add(Terms1, Terms2, Terms).

add_to_order_zero_term([term(0,C1)|Terms], C2, [term(0,C)|Terms]) :- !,
        poly_add(C1, C2, C).
add_to_order_zero_term(Terms, C, [term(0,C)|Terms]).

% polynomial exponentiation

poly_exp(0, _, 1) :- !.
poly_exp(N, Poly, Result) :-
        M is N>>1,
        N is M<<1, !,
        poly_exp(M, Poly, Part),
        poly_mul(Part, Part, Result).
poly_exp(N, Poly, Result) :-
        M is N-1,
        poly_exp(M, Poly, Part),
        poly_mul(Poly, Part, Result).

% polynomial multiplication

poly_mul(poly(Var,Terms1), poly(Var,Terms2), poly(Var,Terms)) :- !,
        term_mul(Terms1, Terms2, Terms).
poly_mul(poly(Var1,Terms1), poly(Var2,Terms2), poly(Var1,Terms)) :-
        Var1 less_than Var2, !,
        mul_through(Terms1, poly(Var2,Terms2), Terms).
poly_mul(P, poly(Var,Terms2), poly(Var,Terms)) :- !,
        mul_through(Terms2, P, Terms).
poly_mul(poly(Var,Terms1), C, poly(Var,Terms)) :- !,
        mul_through(Terms1, C, Terms).
poly_mul(C1, C2, C) :-
        C is C1*C2.

term_mul([], _, []) :- !.
term_mul(_, [], []) :- !.
term_mul([Term|Terms1], Terms2, Terms) :-
        single_term_mul(Terms2, Term, PartA),
        term_mul(Terms1, Terms2, PartB),
        term_add(PartA, PartB, Terms).

single_term_mul([], _, []) :- !.
single_term_mul([term(E1,C1)|Terms1], term(E2,C2),
                [term(E,C)|Terms]) :-
        E is E1+E2,
        poly_mul(C1, C2, C),
        single_term_mul(Terms1, term(E2,C2), Terms).

mul_through([], _, []) :- !.
mul_through([term(E,Term)|Terms], Poly,
                [term(E,NewTerm)|NewTerms]) :-
        poly_mul(Term, Poly, NewTerm),
        mul_through(Terms, Poly, NewTerms).
```

```
%
%   browse
%
%   Tep Dobry
%   (from Lisp version by R. P. Gabriel)
%
%   (modified January 1987 by Herve' Touati)

main :-
        init(100,10,4,
         [[a,a,a,b,b,b,b,a,a,a,a,a,b,b,a,a,a],
          [a,a,b,b,b,b,a,a,[a,a],[b,b]],
          [a,a,a,b,[b,a],b,a,b,a]
         ],
         Symbols),
        randomize(Symbols,RSymbols,21),!,
        investigate(RSymbols,
                [[star(SA),B,star(SB),B,a,star(SA),
                        a,star(SB),star(SA)],
                 [star(SA),star(SB),star(SB),star(SA),
                        [star(SA)],[star(SB)]],
                 [_,_,star(_),[b,a],star(_),_,_]
                ]).


init(N,M,Npats,Ipats,Result) :-
        init(N,M,M,Npats,Ipats,Result).

init(0,_,_,_,_,_) :- !.
init(N,I,M,Npats,Ipats,[Symb|Rest]) :-
        fill(I,[],L),
        get_pats(Npats,Ipats,Ppats),
        J is M - I,
        fill(J,[pattern(Ppats)|L],Symb),
        N1 is N - 1,
        (I =:= 0 -> I1 is M; I1 is I - 1),
        init(N1,I1,M,Npats,Ipats,Rest).

fill(0,L,L) :- !.
fill(N,L,[dummy([])|Rest]) :-
        N1 is N - 1,
        fill(N1,L,Rest).

randomize([],[],_) :- !.
randomize(In,[X|Out],Rand) :-
        length(In,Lin),
        Rand1 is (Rand * 17) mod 251,
        N is Rand1 mod Lin,
        split(N,In,X,In1),
        randomize(In1,Out,Rand1).

split(0,[X|Xs],X,Xs) :- !.
split(N,[X|Xs],RemovedElt,[X|Ys]) :-
        N1 is N - 1,
        split(N1,Xs,RemovedElt,Ys).

investigate([],_) :- !.
investigate([U|Units],Patterns) :-
        property(U,pattern,Data),
        p_investigate(Data,Patterns),
        investigate(Units,Patterns).

get_pats(Npats,Ipats,Result) :-
        get_pats(Npats,Ipats,Result,Ipats).

get_pats(0,_,[],_) :- !.
get_pats(N,[X|Xs],[X|Ys],Ipats) :-
        N1 is N - 1,
        get_pats(N1,Xs,Ys,Ipats).
get_pats(N,[],Ys,Ipats) :-
        get_pats(N,Ipats,Ys,Ipats).

property([],_,_) :- fail.  /* don't really need this */
property([Prop|_],P,Val) :-
        functor(Prop,P,_),!,
        arg(1,Prop,Val).
property([_|RProps],P,Val) :-
        property(RProps,P,Val).

p_investigate([],_).
p_investigate([D|Data],Patterns) :-
        p_match(Patterns,D),
        p_investigate(Data,Patterns).

p_match([],_).
p_match([P|Patterns],D) :-
        (match(D,P),fail; true),
        p_match(Patterns,D).

match([],[]) :- !.
match([X|PRest],[Y|SRest]) :-
        var(Y),!,X = Y,
        match(PRest,SRest).
match(List,[Y|Rest]) :-
        nonvar(Y),Y = star(X),!,
        concat(X,SRest,List),
        match(SRest,Rest).
match([X|PRest],[Y|SRest]) :-
        (atom(X) -> X = Y; match(X,Y)),
        match(PRest,SRest).

concat([],L,L).
concat([X|L1],L2,[X|L3]) :- concat(L1,L2,L3).
```

B.

```
%
% press1
%
% An adaption of part of the PRESS symbolic algebra system
% from The Art of Prolog by Sterling and Shapiro
%
main :- equation(_, E, U), solve_equation(E, U, Y),
      write(E), write(' -> '), write(Y), nl, fail.

equation(1, 3*x+1=0, x).
equation(2, x+2=0, x).
equation(3, cos(x)*(1-2*sin(x))=0, x).
equation(4, x^2-3*x+2=0, x).
equation(5, 2^(2*x)-5*2^(x+1)+16=0, x).

solve_equation(A*B=0,X,Solution) :-
      !,
      factorize(A*B, X, Factors/[]),
      remove_duplicates(Factors, Factors1),
      solve_factors(Factors1, X, Solution).
solve_equation(Equation, X, Solution) :-
      single_occurrence(X, Equation),
      !,
      position(X, Equation, [Side|Position]),
      maneuver_sides(Side, Equation, Equation1),
      isolate(Position, Equation1, Solution).
solve_equation(Lhs=Rhs, X, Solution) :-
      polynomial(Lhs, X),
      polynomial(Rhs, X),
      !,
      polynomial_normal_form(Lhs-Rhs, X, PolyForm),
      solve_polynomial_equation(PolyForm, X, Solution).
solve_equation(Equation, X, Solution) :-
      homogenize(Equation, X, Equation1, X1),
      !,
      solve_equation(Equation1, X1, Solution1),
      solve_equation(Solution1, X, Solution).

factorize(A*B, X, Factors/Rest) :-
      !, factorize(A, X, Factors/Factors1),
      factorize(B, X, Factors1/Rest).
factorize(C, X, [C|Factors]/Factors) :-
      subterm(X, C), !.
factorize(_, _, Factors/Factors).

solve_factors([Factor|_], X, Solution) :-
      solve_equation(Factor=0, X, Solution).
solve_factors([_|Factors], X, Solution) :-
      solve_factors(Factors, X, Solution).

maneuver_sides(1, Lhs = Rhs, Lhs = Rhs) :- !.
maneuver_sides(2, Lhs = Rhs, Rhs = Lhs) :- !.

isolate([N|Position], Equation, IsolatedEquation) :-
      isolax(N, Equation, Equation1),
      isolate(Position, Equation1, IsolatedEquation).
isolate([], Equation, Equation).

isolax(1, -Lhs = Rhs, Lhs = -Rhs).
isolax(1, Term1+Term2 = Rhs, Term1 = Rhs-Term2).
isolax(2, Term1+Term2 = Rhs, Term2 = Rhs-Term1).
isolax(1, Term1-Term2 = Rhs, Term1 = Rhs+Term2).
isolax(2, Term1-Term2 = Rhs, Term2 = Term1-Rhs).
isolax(1, Term1*Term2 = Rhs, Term1 = Rhs/Term2) :-
      Term2 == 0.
isolax(2, Term1*Term2 = Rhs, Term2 = Rhs/Term1) :-
      Term1 == 0.
isolax(1, Term1^Term2 = Rhs, Term1 = Rhs^(-Term2)).
isolax(2, Term1^Term2 = Rhs, Term2 = log(base(Term1), Rhs)).
isolax(1, sin(U) = V, U = arcsin(V)).
isolax(1, sin(U) = V, U = pi-arcsin(V)).
isolax(1, cos(U) = V, U = arccos(V)).
isolax(1, cos(U) = V, U = -arccos(V)).

polynomial(X, X) :- !.
polynomial(Term, _) :- atomic(Term), !.
polynomial(Term1+Term2, X) :-
      !, polynomial(Term1, X), polynomial(Term2, X).
polynomial(Term1-Term2, X) :-
      !, polynomial(Term1, X), polynomial(Term2, X).
polynomial(Term1*Term2, X) :-
      !, polynomial(Term1, X), polynomial(Term2, X).
polynomial(Term1/Term2, X) :-
      !, polynomial(Term1, X), polynomial(Term2, X).
polynomial(Term^N, X) :-
      !, integer(N), polynomial(Term, X).

polynomial_normal_form(Polynomial, X, NormalForm) :-
      polynomial_form(Polynomial, X, PolyForm),
      remove_zero_terms(PolyForm, NormalForm), !.

polynomial_form(X, X, [(1,1)]).
polynomial_form(X^N, X, [(1,N)]).
polynomial_form(Term1+Term2, X, PolyForm) :-
      polynomial_form(Term1, X, PolyForm1),
      polynomial_form(Term2, X, PolyForm2),
      add_polynomials(PolyForm1, PolyForm2, PolyForm).
polynomial_form(Term1-Term2, X, PolyForm) :-
      polynomial_form(Term1, X, PolyForm1),
      polynomial_form(Term2, X, PolyForm2),
      subtract_polynomials(PolyForm1, PolyForm2, PolyForm).
polynomial_form(Term1*Term2, X, PolyForm) :-
      polynomial_form(Term1, X, PolyForm1),
      polynomial_form(Term2, X, PolyForm2),
      multiply_polynomials(PolyForm1, PolyForm2, PolyForm).
polynomial_form(Term^N, X, PolyForm) :-
      polynomial_form(Term, X, PolyForm1),
      binomial(PolyForm1, N, PolyForm).
polynomial_form(Term, X, [(Term,0)]) :-
      free_of(X, Term).

remove_zero_terms([(0,_)|Poly], Poly1) :-
      !, remove_zero_terms(Poly, Poly1).
remove_zero_terms([(C,N)|Poly], [(C,N)|Poly1]) :-
      C == 0, !, remove_zero_terms(Poly, Poly1).
remove_zero_terms([], []).

add_polynomials([], Poly, Poly) :- !.
add_polynomials(Poly, [], Poly) :- !.
add_polynomials([(Ai,Ni)|Poly1],[(Aj,Nj)|Poly2],[(Ai,Ni)|Poly]) :-
      Ni > Nj, !, add_polynomials(Poly1, [(Aj, Nj)|Poly2], Poly).
add_polynomials([(Ai,Ni)|Poly1],[(Aj,Nj)|Poly2],[(A,Ni)|Poly]) :-
      Ni =:= Nj, !, A is Ai + Aj, add_polynomials(Poly1, Poly2, Poly).
add_polynomials([(Ai,Ni)|Poly1],[(Aj,Nj)|Poly2],[(Aj,Nj)|Poly]) :-
      Ni < Nj, !, add_polynomials([(Ai, Ni)|Poly1], Poly2, Poly).

subtract_polynomials(Poly1, Poly2, Poly) :-
      multiply_single(Poly2, (-1, 0), Poly3),
      add_polynomials(Poly1, Poly3, Poly), !.

multiply_single([(C1,N1)|Poly1], (C,N), [(C2,N2)|Poly]) :-
      C2 is C1*C, N2 is N1 + N, multiply_single(Poly1, (C, N), Poly).
multiply_single([], _, []).

multiply_polynomials([(C,N)|Poly1], Poly2, Poly) :-
      multiply_single(Poly2, (C,N), Poly3),
      multiply_polynomials(Poly1, Poly2, Poly4),
      add_polynomials(Poly3, Poly4, Poly).
multiply_polynomials([], _, []).

binomial(Poly, 1, Poly). /* ? */

solve_polynomial_equation(PolyEquation, X, X = - B/A) :-
      linear(PolyEquation), !,
      pad(PolyEquation, [(A,1),(B,0)]).
solve_polynomial_equation(PolyEquation, X, Solution) :-
      quadratic(PolyEquation), !,
      pad(PolyEquation, [(A,2),(B,1),(C,0)]),
      discriminant(A, B, C, Discriminant),
      root(X, A, B, C, Discriminant, Solution).
```

```
discriminant(A, B, C, D) :- D is B*B - 4*A*C.

root(X, A, B, _, 0, X = -B/(2*A)).
root(X, A, B, _, D, X = (-B+sqrt(D))/(2*A)).
root(X, A, B, _, D, X = (-B-sqrt(D))/(2*A)).

pad([(C,N)|Poly], [(C,N)|Poly1]) :-
    !, pad(Poly, Poly1).
pad(Poly, [(0,_)|Poly1]) :-
    !, pad(Poly, Poly1).
pad([], []).

linear([(_,1)|_]).

quadratic([(_, 2)|_]).

homogenize(Equation, X, Equation1, X1) :-
    offenders(Equation, X, Offenders),
    reduced_term(X, Offenders, Type, X1),
    rewrite(Offenders, Type, X1, Substitutions),
    substitute(Equation, Substitutions, Equation1).

offenders(Equation, X, Offenders) :-
    parse(Equation, X, Offenders1/[]),
    remove_duplicates(Offenders1, Offenders),
    multiple(Offenders).

reduced_term(X, Offenders, Type, X1) :-
    classify(Offenders, X, Type),
    candidate(Type, Offenders, X, X1).

classify(Offenders, X, exponential) :-
    exponential_offenders(Offenders, X).

exponential_offenders([A^B|Offs], X) :-
    free_of(X, A),
    subterm(X, B),
    exponential_offenders(Offs, X).
exponential_offenders([], _).

candidate(exponential, Offenders, X, A^X) :-
    base(Offenders, A), polynomial_exponents(Offenders, X).

base([A^_|Offs], A) :- base(Offs, A).
base([], _).

polynomial_exponents([_^B|Offs], X) :-
    polynomial(B, X), polynomial_exponents(Offs, X).
polynomial_exponents([], _).


parse(A+B, X, L1/L2) :-
    !, parse(A, X, L1/L3), parse(B, X, L3/L2).
parse(A*B, X, L1/L2) :-
    !, parse(A, X, L1/L3), parse(B, X, L3/L2).
parse(A-B, X, L1/L2) :-
    !, parse(A, X, L1/L3), parse(B, X, L3/L2).
parse(A=B, X, L1/L2) :-
    !, parse(A, X, L1/L3), parse(B, X, L3/L2). /* ? */
parse(A^B, X, L) :-
    integer(B), !, parse(A, X, L). /* ? */
parse(A, X, L/L) :- free_of(X, A), !.
parse(A, X, [A|L]/L) :- subterm(X, A), !.

substitute(A+B, Subs, NewA+NewB) :-
    !, substitute(A, Subs, NewA), substitute(B, Subs, NewB).
substitute(A*B, Subs, NewA*NewB) :-
    !, substitute(A, Subs, NewA), substitute(B, Subs, NewB).
substitute(A-B, Subs, NewA-NewB) :-
    !, substitute(A, Subs, NewA), substitute(B, Subs, NewB).
substitute(A=B, Subs, NewA=NewB) :-
    !, substitute(A, Subs, NewA), substitute(B, Subs, NewB).
substitute(A^B, Subs, NewA^B) :-
    integer(B), !, substitute(A, Subs, NewA).
substitute(A, Subs, B) :-
    member(A = B, Subs), !.
substitute(A, _, A).

rewrite([Off|Offs], Type, X1, [Off=Term|Rewrites]) :-
    homog_axiom(Type, Off, X1, Term),
    rewrite(Offs, Type, X1, Rewrites).
rewrite([], _, _, []).

homog_axiom(exponential, A^(N*X), A^X, (A^X)^N).
homog_axiom(exponential, A^(-X), A^X, 1/(A^X)).
homog_axiom(exponential, A^(X+B), A^X, A^B*A^X).

subterm(Term, Term).
subterm(Sub, Term) :-
    functor(Term, _, N), N > 0, subterm(N, Sub, Term).

subterm(N, Sub, Term) :-
    N > 1, N1 is N - 1, subterm(N1, Sub, Term).
subterm(N, Sub, Term) :-
    arg(N, Term, Arg), subterm(Sub, Arg).

position(Term, Term, []) :- !.
position(Sub, Term, Path) :-
    functor(Term, _, N), N > 0,
    position(N, Sub, Term, Path), !.

position(N, Sub, Term, [N|Path]) :-
    arg(N, Term, Arg), position(Sub, Arg, Path).
position(N, Sub, Term, Path) :-
    N > 1, N1 is N - 1, position(N1, Sub, Term, Path).

free_of(Subterm, Term) :-
    occurrence(Subterm, Term, N), !, N = 0.

single_occurrence(Subterm, Term) :-
    occurrence(Subterm, Term, N), !, N = 1.

occurrence(Term, Term, 1) :- !.
occurrence(Sub, Term, N) :-
    functor(Term, _, M), M > 0, !,
    occurrence(M, Sub, Term, 0, N).
occurrence(_, _, 0).

occurrence(M, Sub, Term, N1, N2) :-
    M > 0, !, arg(M, Term, Arg), occurrence(Sub, Arg, N),
    N3 is N + N1, M1 is M - 1,
    occurrence(M1, Sub, Term, N3, N2).
occurrence(0, _, _, N, N).

multiple([_,_|_]).

remove_duplicates(X, X).

member(H, [H|_]).
member(H, [_|T]) :- member(H, T).
```

B.

```
%
%  reducer
%
% A Graph Reducer for T-Combinators:
% Reduces a T-combinator expression to a final answer.  Recognizes
% the combinators I,K,S,B,C,S',B',C', cond, apply, arithmetic, tests,
% basic list operations, and function definitions in the data base stored
% as facts of the form t_def(_func, _args, _expr).
% Written by Peter Van Roy

% Uses write/1, compare/3, functor/3, arg/3.
main :-
        try(fac(3), _ans1),
        write(_ans1), nl,
        try(quick([3,1,2]), _ans2),
        write(_ans2), nl.

try(_inpexpr, _anslist) :-
        listify(_inpexpr, _list),
        curry(_list, _curry),
        t_reduce(_curry, _ans), nl,
        make_list(_ans, _anslist).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Examples of applicative functions which can be compiled & executed.
% This test version compiles them just before each execution.

% Factorial function:
t_def(fac, [N], cond(N=0, 1, N*fac(N-1))).

% Quicksort:
t_def(quick, [_l], cond(_l=[], [],
              cond(tl(_l)=[], _l,
              quick2(split(hd(_l),tl(_l)))))).
t_def(quick2, [_l], append(quick(hd(_l)), quick(tl(_l)))).

t_def(split, [_e,_l], cond(_l=[], [[_e]|[]],
                cond(hd(_l)=<_e, inserthead(hd(_l),split(_e,tl(_l))),
                inserttail(hd(_l),split(_e,tl(_l)))))).
t_def(inserthead, [_e,_l], [[_e|hd(_l)]|tl(_l)]).
t_def(inserttail, [_e,_l], [hd(_l)|[_e|tl(_l)]]).

t_def(append, [_a,_b], cond(_a=[], _b, [hd(_a)|append(tl(_a),_b)])).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Full reduction:
% A dot '.' is printed for each reduction step.

t_reduce(_expr, _ans) :-
        atomic(_expr), !,
         _ans=_expr.
% The reduction of '.' must be here to avoid an infinite loop
t_reduce([_y,_x|'.'], [_yr,_xr|'.']) :-
        t_reduce(_x, _xr),
        !,
        t_reduce(_y, _yr),
        !.
t_reduce(_expr, _ans) :-
        t_append(_next, _red, _form, _expr),
        write('.'),
        t_redex(_form, _red),
        !,
        t_reduce(_next, _ans),
        !.

t_append(_link, _link, _l, _l).
t_append([_a|_l1], _link, _l2, [_a|_l3]) :- t_append(_l1, _link, _l2, _l3).

% One step of the reduction:

% Combinators:
t_redex([_x,_g,_f,_k|sp], [[_xr|_g],[_xr|_f]|_k]) :- t_reduce(_x, _xr).
t_redex([_x,_g,_f,_k|bp], [[_x|_g],_f|_k]).
t_redex([_x,_g,_f,_k|cp], [_g,[_x|_f]|_k]).
t_redex([_x,_g,_f|s], [[_xr|_g]|[_xr|_f]]) :- t_reduce(_x, _xr).
t_redex([_x,_g,_f|b], [[_x|_g]|_f]).
t_redex([_x,_g,_f|c], [_g,_x|_f]).
t_redex([_y,_x|k], _x).
t_redex([_x|i], _x).

% Conditional:
t_redex([_elsepart,_ifpart,_cond|cond], _ifpart) :-
        t_reduce(_cond, _bool), _bool=true, !.
        % Does NOT work if _bool is substituted in the call!
t_redex([_elsepart,_ifpart,_cond|cond], _elsepart).

% Apply:
t_redex([_f|apply], _fr) :-
        t_reduce(_f, _fr).

% List operations:
t_redex([_arg|hd], _x) :-
        t_reduce(_arg, [_y,_x|'.']).
t_redex([_arg|tl], _y) :-
        t_reduce(_arg, [_y,_x|'.']).
```

```
% Arithmetic:
t_redex([_y,_x|_op], _res) :-
        atom(_op),
        member(_op, ['+', '-', '*', '//', 'mod']),
        t_reduce(_x, _xres),
        t_reduce(_y, _yres),
        number(_xres), number(_yres),
        eval(_op, _res, _xres, _yres).

% Tests:
t_redex([_y,_x|_test], _res) :-
        atom(_test),
        member(_test, ['<', '>', '=<', '>=', '==', '=:=']),
        t_reduce(_x, _xres),
        t_reduce(_y, _yres),
        number(_xres), number(_yres),
        (relop(_test, _xres, _yres)
        -> _res=true
        ;  _res=false
        ), !.

% Equality:
t_redex([_y,_x|=], _res) :-
        t_reduce(_x, _xres),
        t_reduce(_y, _yres),
        (_xres=_yres -> _res=true; _res=false), !.

% Arithmetic functions:
t_redex([_x|_op], _res) :-
        atom(_op),
        member(_op, ['-']),
        t_reduce(_x, _xres),
        number(_xres),
        eval1(_op, _t, _xres).

% Definitions:
% Assumes a fact t_def(_func,_def) in the database for every
% defined function.
t_redex(_in, _out) :-
        append(_par,_func,_in),
        atom(_func),
        t_def(_func, _args, _expr),
        t(_args, _expr, _def),
        append(_par,_def,_out).

% Basic arithmetic and relational operators:

eval(  '+', C, A, B) :- C is A + B.
eval(  '-', C, A, B) :- C is A - B.
eval(  '*', C, A, B) :- C is A * B.
eval( '//', C, A, B) :- C is A // B.
eval('mod', C, A, B) :- C is A mod B.

eval1('-', C, A) :- C is -A.

relop(  '<', A, B) :- A<B.
relop(  '>', A, B) :- A>B.
relop( '=<', A, B) :- A=<B.
relop( '>=', A, B) :- A>=B.
relop('==', A, B) :- A==B.
relop('=:=', A, B) :- A=:=B.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Scheme T:
% A Translation Scheme for T-Combinators

% Translate an expression to combinator form
% by abstracting out all variables in _argvars:
t(_argvars, _expr, _trans) :-
        listify(_expr, _list),
        curry(_list, _curry),
        t_argvars(_argvars, _curry, _trans), !.

t_argvars([], _trans, _trans).
t_argvars([_x|_argvars], _in, _trans) :-
        t_argvars(_argvars, _in, _mid),
        t_vars(_mid, _vars), % calculate variables in each subexpression
        t_trans(_x, _mid, _vars, _trans). % main translation routine

% Curry the original expression:
% This converts an applicative expression of any number
% of arguments and any depth of nesting into an expression
% where all functions are curried, i.e. all function
% applications are to one argument and have the form
% [_arg|_func] where _func & _arg are also of that form.
% Input is a nested function application in list form.
% Currying makes t_trans faster.
curry(_a, _a) :- (var(_a); atomic(_a)), !.
curry([_func|_args], _cargs) :-
        currylist(_args, _cargs, _func).

% Transform [_a1, ..., _aN] to [_cN, ..., _c1|_link]-_link
currylist([], _link, _link) :- !.
currylist([_a|_args], _cargs, _link) :-
        curry(_a, _c),
        currylist(_args, _cargs, [_c|_link]).
```

```
% Calculate variables in each subexpression:
% To any expression a list of the form
% [_vexpr, _astr, _fstr] is matched.
% If the expression is a variable or an atom
% then this list only has the first element.
% _vexpr = List of all variables in the expression.
% _astr, _fstr = Similar structures for argument & function.
t_vars(_v, [[_v]]) :- var(_v), !.
t_vars(_a, [[]]) :- atomic(_a), !.
t_vars([_func], [[]]) :- atomic(_func), !.
t_vars([_arg|_func], [_g,[_g1|_af1],[_g2|_af2]]) :-
        t_vars(_arg, [_g1|_af1]),
        t_vars(_func, [_g2|_af2]),
        unionv(_g1, _g2, _g).

% The main translation routine:
% trans(_var, _curriedexpr, _varexpr, _result)
% The translation scheme T in the article is followed literally.
% A good example of Prolog as a specification language.
t_trans(_x, _a, _, [_a|k]) :- (atomic(_a); var(_a), _a==_x), !.
t_trans(_x, _y, _, i) :- _x==_y, !.
t_trans(_x, _e, [_ve|_], [_e|k]) :- notinv(_x, _ve).
t_trans(_x, [_f|_e], [_vef,_sf,_se], _res) :-
        _sf=[_vf|_],
        _se=[_ve|_other],
        (atom(_e); _other=[_,[_ve1|_]], _ve1==[]),
        t_rule1(_x, _e, _ve, _se, _f, _vf, _sf, _res).
t_trans(_x, [_g|[_f|_e]], [_vefg,_sg,_sef], _res) :-
        _sg=[_vg|_],
        _sef=[_vef,_sf,_se],
        _se=[_ve|_],
        _sf=[_vf|_],
        t_rule2(_x, _e, _f, _vf, _sf, _g, _vg, _sg, _res).

% First complex rule of translation scheme T:
t_rule1(_x, _e, _ve, _se, _f, _vf, _sf, _e) :-
        notinv(_x, _ve), _x==_f, !.
t_rule1(_x, _e, _ve, _se, _f, _vf, _sf, [_resf,_e|b]) :-
        notinv(_x, _ve), inv(_x, _vf), _x==_f, !,
        t_trans(_x, _f, _sf, _resf).
t_rule1(_x, _e, _ve, _se, _f, _vf, _sf, [_f,_rese|c]) :-
        /* inv(_x, _ve), */
        notinv(_x, _vf), !,
        t_trans(_x, _e, _se, _rese).
t_rule1(_x, _e, _ve, _se, _f, _vf, _sf, [_resf,_rese|s]) :-
        /* inv(_x, _ve), inv(_x, _vf), */
        t_trans(_x, _e, _se, _rese),
        t_trans(_x, _f, _sf, _resf).

% Second complex rule of translation scheme T:
t_rule2(_x, _e, _f, _vf, _sf, _g, _vg, _sg, [_g,_e|c]) :-
        _x==_f, notinv(_x, _vg), !.
t_rule2(_x, _e, _f, _vf, _sf, _g, _vg, _sg, [_resg,_e|s]) :-
        _x==_f, /* inv(_x, _vg), */ !,
        t_trans(_x, _g, _sg, _resg).
t_rule2(_x, _e, _f, _vf, _sf, _g, _vg, _sg, [_g,_resf,_e|cp]) :-
        /* _x==_f, */ inv(_x, _vf), notinv(_x, _vg), !,
        t_trans(_x, _f, _sf, _resf).
t_rule2(_x, _e, _f, _vf, _sf, _g, _vg, _sg, [_resg,_resf,_e|sp]) :-
        /* _x==_f, */ inv(_x, _vf), /* inv(_x, _vg), */ !,
        t_trans(_x, _f, _sf, _resf),
        t_trans(_x, _g, _sg, _resg).
t_rule2(_x, _e, _f, _vf, _sf, _g, _vg, _sg, [_f|_e]) :-
        /* notinv(_x, _vf), */ _x==_g, !.
t_rule2(_x, _e, _f, _vf, _sf, _g, _vg, _sg, [_resg,_f,_e|bp]) :-
        /* notinv(_x, _vf), inv(_x, _vg), _x==_g, */
        t_trans(_x, _g, _sg, _resg).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% List utilities:

% Convert curried list into a regular list:
make_list(_a, _a) :- atomic(_a).
make_list([_b,_a|'.'], [_a|_rb]) :- make_list(_b, _rb).

listify(_X, _X) :-
        (var(_X); atomic(_X)), !.
listify(_Expr, [_Op|_LArgs]) :-
        functor(_Expr, _Op, N),
        listify_list(1, N, _Expr, _LArgs).

listify_list(I, N, _, []) :- I>N, !.
listify_list(I, N, _Expr, [_LA|_LArgs]) :- I=<N, !,
        arg(I, _Expr, _A),
        listify(_A, _LA),
        I1 is I+1,
        listify_list(I1, N, _Expr, _LArgs).

member(X, [X|_]).
member(X, [_|L]) :- member(X, L).

append([], L, L).
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Set utilities:
% Implementation inspired by R. O'Keefe, Practical Prolog.
% Sets are represented as sorted lists without duplicates.
% Predicates with 'v' suffix work with sets containing
% uninstantiated vars.

% *** Intersection
intersectv([], _, []).
intersectv([A|S1], S2, S) :- intersectv_2(S2, A, S1, S).

intersectv_2([], _, _, []).
intersectv_2([B|S2], A, S1, S) :-
        compare(Order, A, B),
        intersectv_3(Order, A, S1, B, S2, S).

intersectv_3(<, _, S1, B, S2,     S) :- intersectv_2(S1, B, S2, S).
intersectv_3(=, A, S1, _, S2, [A|S]) :- intersectv(S1, S2, S).
intersectv_3(>, A, S1, _, S2,     S) :- intersectv_2(S2, A, S1, S).

intersectv_list([], []).
intersectv_list([InS|Sets], OutS) :- intersectv_list(Sets, InS, OutS).

intersectv_list([]) --> [].
intersectv_list([S|Sets]) --> intersectv(S), intersectv_list(Sets).

% *** Difference
diffv([], _, []).
diffv([A|S1], S2, S) :- diffv_2(S2, A, S1, S).

diffv_2([], A, S1, [A|S1]).
diffv_2([B|S2], A, S1, S) :-
        compare(Order, A, B),
        diffv_3(Order, A, S1, B, S2, S).

diffv_3(<, A, S1, B, S2, [A|S]) :- diffv(S1, [B|S2], S).
diffv_3(=, A, S1, _, S2,     S) :- diffv(S1, S2, S).
diffv_3(>, A, S1, _, S2,     S) :- diffv_2(S2, A, S1, S).

% *** Union
unionv([], S2, S2).
unionv([A|S1], S2, S) :- unionv_2(S2, A, S1, S).

unionv_2([], A, S1, [A|S1]).
unionv_2([B|S2], A, S1, S) :-
        compare(Order, A, B),
        unionv_3(Order, A, S1, B, S2, S).

unionv_3(<, A, S1, B, S2, [A|S]) :- unionv_2(S1, B, S2, S).
unionv_3(=, A, S1, _, S2, [A|S]) :- unionv(S1, S2, S).
unionv_3(>, A, S1, B, S2, [B|S]) :- unionv_2(S2, A, S1, S).

% *** Subset
subsetv([], _).
subsetv([A|S1], [B|S2]) :-
        compare(Order, A, B),
        subsetv_2(Order, A, S1, S2).

subsetv_2(=, _, S1, S2) :- subsetv(S1, S2).
subsetv_2(>, A, S1, S2) :- subsetv([A|S1], S2).

% For unordered lists S1:
small_subsetv([], _).
small_subsetv([A|S1], S2) :- inv(A, S2), small_subsetv(S1, S2).

% *** Membership
inv(A, [B|S]) :-
        compare(Order, A, B),
        inv_2(Order, A, S).

inv_2(=, _, _).
inv_2(>, A, S) :- inv(A, S).

% *** Non-membership
notinv(A, S) :- notinv_2(S, A).

notinv_2([], _).
notinv_2([B|S], A) :-
        compare(Order, A, B),
        notinv_3(Order, A, S).

notinv_3(<, _, _).
notinv_3(>, A, S) :- notinv_2(S, A).
```

B.

```
%
% boyer
%
% Evan Tick (from Lisp version by R. P. Gabriel)
%
% November 1985
%
% prove arithmetic theorem

main :- wff(Wff),
	rewrite(Wff,NewWff),
	tautology(NewWff,[],[]).

wff(implies(and(implies(X,Y),
		and(implies(Y,Z),
		    and(implies(Z,U),
			implies(U,W)))),
	    implies(X,W))) :-
	X = f(plus(plus(a,b),plus(c,zero))),
	Y = f(times(times(a,b),plus(c,d))),
	Z = f(reverse(append(append(a,b),[]))),
	U = equal(plus(a,b),difference(x,y)),
	W = lessp(remainder(a,b),member(a,length(b))).

tautology(Wff) :-
	write('rewriting...'),nl,
	rewrite(Wff,NewWff),
	write('proving...'),nl,
	tautology(NewWff,[],[]).

tautology(Wff,Tlist,Flist) :-
	(truep(Wff,Tlist) -> true
	;falsep(Wff,Flist) -> fail
	;Wff = if(If,Then,Else) ->
		(truep(If,Tlist) -> tautology(Then,Tlist,Flist)
		;falsep(If,Flist) -> tautology(Else,Tlist,Flist)
		;tautology(Then,[If|Tlist],Flist),% both must hold
		 tautology(Else,Tlist,[If|Flist])
		)
	),!.

rewrite(Atom,Atom) :-
	atomic(Atom),!.
rewrite(Old,New) :-
	functor(Old,F,N),
	functor(Mid,F,N),
	rewrite_args(N,Old,Mid),
	( equal(Mid,Next),	% should be ->, but is compiler smart
	  rewrite(Next,New)	% enough to generate cut for -> ?
	; New=Mid
	),!.

rewrite_args(0,_,_) :- !.
rewrite_args(N,Old,Mid) :-
	arg(N,Old,OldArg),
	arg(N,Mid,MidArg),
	rewrite(OldArg,MidArg),
	N1 is N-1,
	rewrite_args(N1,Old,Mid).

truep(t,_) :- !.
truep(Wff,Tlist) :- member(Wff,Tlist).

falsep(f,_) :- !.
falsep(Wff,Flist) :- member(Wff,Flist).

member(X,[X|_]) :- !.
member(X,[_|T]) :- member(X,T).


equal(  and(P,Q),
	if(P,if(Q,t,f),f)
	).
equal(  append(append(X,Y),Z),
	append(X,append(Y,Z))
	).
equal(  assignment(X,append(A,B)),
	if(assignedp(X,A),
	   assignment(X,A),
	   assignment(X,B))
	).
equal(  assume_false(Var,Alist),
	cons(cons(Var,f),Alist)
	).
equal(  assume_true(Var,Alist),
	cons(cons(Var,t),Alist)
	).
equal(  boolean(X),
	or(equal(X,t),equal(X,f))
	).
equal(  car(gopher(X)),
	if(listp(X),
	car(flatten(X)),
	zero)
	).
equal(  compile(Form),
	reverse(codegen(optimize(Form),[]))
	).
equal(  count_list(Z,sort_lp(X,Y)),
	plus(count_list(Z,X),
	     count_list(Z,Y))
	).
equal(  countps_(L,Pred),
	countps_loop(L,Pred,zero)
	).
equal(  difference(A,B),
	C
	) :- difference(A,B,C).
equal(  divides(X,Y),
	zerop(remainder(Y,X))
	).
equal(  dsort(X),
	sort2(X)
	).
equal(  eqp(X,Y),
	equal(fix(X),fix(Y))
	).
equal(  equal(A,B),
	C
	) :- eq(A,B,C).
equal(  even1(X),
	if(zerop(X),t,odd(decr(X)))
	).
equal(  exec(append(X,Y),Pds,Envrn),
	exec(Y,exec(X,Pds,Envrn),Envrn)
	).
equal(  exp(A,B),
	C
	) :- exp(A,B,C).
equal(  fact_(I),
	fact_loop(I,1)
	).
equal(  falsify(X),
	falsify1(normalize(X),[])
	).
equal(  fix(X),
	if(numberp(X),X,zero)
	).
equal(  flatten(cdr(gopher(X))),
	if(listp(X),
	   cdr(flatten(X)),
	   cons(zero,[]))
	).
equal(  gcd(A,B),
	C
	) :- gcd(A,B,C).
equal(  get(J,set(I,Val,Mem)),
	if(eqp(J,I),Val,get(J,Mem))
	).
equal(  greatereqp(X,Y),
	not(lessp(X,Y))
	).
equal(  greatereqpr(X,Y),
	not(lessp(X,Y))
	).
equal(  greaterp(X,Y),
	lessp(Y,X)
	).
equal(  if(if(A,B,C),D,E),
	if(A,if(B,D,E),if(C,D,E))
	).
equal(  iff(X,Y),
	and(implies(X,Y),implies(Y,X))
	).
equal(  implies(P,Q),
	if(P,if(Q,t,f),t)
	).
equal(  last(append(A,B)),
	if(listp(B),
	   last(B),
	   if(listp(A),
	      cons(car(last(A))),
	      B))
	).
equal(  length(A),
	B
	) :- mylength(A,B).
equal(	lesseqp(X,Y),
	not(lessp(Y,X))
	).
equal(  lessp(A,B),
	C
	) :- lessp(A,B,C).
equal(  listp(gopher(X)),
	listp(X)
	).
equal(  mc_flatten(X,Y),
	append(flatten(X),Y)
	).
equal(  meaning(A,B),
	C
	) :- meaning(A,B,C).
equal(  member(A,B),
	C
```

```
        ) :- mymember(A,B,C).
equal(  not(P),
        if(P,f,t)
        ).
equal(  nth(A,B),
        C
        ) :- nth(A,B,C).
equal(  numberp(greatest_factor(X,Y)),
        not(and(or(zerop(Y),equal(Y,1)),
                not(numberp(X))))
        ).
equal(  or(P,Q),
        if(P,t,if(Q,t,f),f)
        ).
equal(  plus(A,B),
        C
        ) :- plus(A,B,C).
equal(  power_eval(A,B),
        C
        ) :- power_eval(A,B,C).
equal(  prime(X),
        and(not(zerop(X)),
            and(not(equal(X,add1(zero))),
                prime1(X,decr(X))))
        ).
equal(  prime_list(append(X,Y)),
        and(prime_list(X),prime_list(Y))
        ).
equal(  quotient(A,B),
        C
        ) :- quotient(A,B,C).
equal(  remainder(A,B),
        C
        ) :- remainder(A,B,C).
equal(  reverse_(X),
        reverse_loop(X,[])
        ).
equal(  reverse(append(A,B)),
        append(reverse(B),reverse(A))
        ).
equal(  reverse_loop(A,B),
        C
        ) :- reverse_loop(A,B,C).
equal(  samefringe(X,Y),
        equal(flatten(X),flatten(Y))
        ).
equal(  sigma(zero,I),
        quotient(times(I,add1(I)),2)
        ).
equal(  sort2(delete(X,L)),
        delete(X,sort2(L))
        ).
equal(  tautology_checker(X),
        tautologyp(normalize(X),[])
        ).
equal(  times(A,B),
        C
        ) :- times(A,B,C).
equal(  times_list(append(X,Y)),
        times(times_list(X),times_list(Y))
        ).
equal(  value(normalize(X),A),
        value(X,A)
        ).
equal(  zerop(X),
        or(equal(X,zero),not(numberp(X)))
        ).

difference(X, X, zero) :- !.
difference(plus(X,Y), X, fix(Y)) :- !.
difference(plus(Y,X), X, fix(Y)) :- !.
difference(plus(X,Y), plus(X,Z), difference(Y,Z)) :- !.
difference(plus(B,plus(A,C)), A, plus(B,C)) :- !.
difference(add1(plus(Y,Z)), Z, add1(Y)) :- !.
difference(add1(add1(X)), 2, fix(X)).

eq(plus(A,B), zero, and(zerop(A),zerop(B))) :- !.
eq(plus(A,B), plus(A,C), equal(fix(B),fix(C))) :- !.
eq(zero, difference(X,Y),not(lessp(Y,X))) :- !.
eq(X, difference(X,Y),and(numberp(X),
                          and(or(equal(X,zero),
                                 zerop(Y))))) :- !.
eq(times(X,Y), zero, or(zerop(X),zerop(Y))) :- !.
eq(append(A,B), append(A,C), equal(B,C)) :- !.
eq(flatten(X), cons(Y,[]), and(nlistp(X),equal(X,Y))) :- !.
eq(greatest_factor(X,Y),zero, and(or(zerop(Y),equal(Y,1)),
                                  equal(X,zero))) :- !.
eq(greatest_factor(X,_),1, equal(X,1)) :- !.
eq(Z, times(W,Z), and(numberp(Z),
                  or(equal(Z,zero),
                     equal(W,1)))) :- !.
eq(X, times(X,Y), or(equal(X,zero),
                 and(numberp(X),equal(Y,1)))) :- !.
eq(times(A,B), 1, and(not(equal(A,zero)),
                  and(not(equal(B,zero)),
                      and(numberp(A),
                          and(numberp(B),
                              and(equal(decr(A),zero),
                                  equal(decr(B),zero))))))) :- !.
eq(difference(X,Y), difference(Z,Y),if(lessp(X,Y),
                                       not(lessp(Y,Z)),
                                       if(lessp(Z,Y),
                                          not(lessp(Y,X)),
                                          equal(fix(X),fix(Z))))) :- !.
eq(lessp(X,Y), Z, if(lessp(X,Y),
                     equal(t,Z),
                     equal(f,Z))).

exp(I, plus(J,K), times(exp(I,J),exp(I,K))) :- !.
exp(I, times(J,K), exp(exp(I,J),K)).

gcd(X, Y, gcd(Y,X)) :- !.
gcd(times(X,Z), times(Y,Z), times(Z,gcd(X,Y))).

mylength(reverse(X),length(X)).
mylength(cons(_,cons(_,cons(_,cons(_,cons(_,cons(_,X7)))))),
         plus(6,length(X7))).

lessp(remainder(_,Y), Y, not(zerop(Y))) :- !.
lessp(quotient(I,J), I, and(not(zerop(I)),
                            or(zerop(J),
                               not(equal(J,1))))) :- !.
lessp(remainder(X,Y), X, and(not(zerop(Y)),
                             and(not(zerop(X)),
                                 not(lessp(X,Y))))) :- !.
lessp(plus(X,Y), plus(X,Z), lessp(Y,Z)) :- !.
lessp(times(X,Z), times(Y,Z), and(not(zerop(Z)),
                                  lessp(X,Y))) :- !.
lessp(Y, plus(X,Y), not(zerop(X))) :- !.
lessp(length(delete(X,L)), length(L), member(X,L)).

meaning(plus_tree(append(X,Y)),A,
        plus(meaning(plus_tree(X),A),
             meaning(plus_tree(Y),A))) :- !.
meaning(plus_tree(plus_fringe(X)),A,
        fix(meaning(X,A))) :- !.
meaning(plus_tree(delete(X,Y)),A,
        if(member(X,Y),
           difference(meaning(plus_tree(Y),A),
                      meaning(X,A)),
           meaning(plus_tree(Y),A))).

mymember(X,append(A,B),or(member(X,A),member(X,B))) :- !.
mymember(X,reverse(Y),member(X,Y)) :- !.
mymember(A,intersect(B,C),and(member(A,B),member(A,C))).

nth(zero,_,zero).
nth([],I,if(zerop(I),[],zero)).
nth(append(A,B),I,append(nth(A,I),nth(B,difference(I,length(A))))).

plus(plus(X,Y),Z,
         plus(X,plus(Y,Z))) :- !.
plus(remainder(X,Y),
         times(Y,quotient(X,Y)),
         fix(X)) :- !.
plus(X,add1(Y),
         if(numberp(Y),
        add1(plus(X,Y)),
        add1(X))).

power_eval(big_plus1(L,I,Base),Base,
          plus(power_eval(L,Base),I)) :- !.
power_eval(power_rep(I,Base),Base,
          fix(I)) :- !.
power_eval(big_plus(X,Y,I,Base),Base,
          plus(I,plus(power_eval(X,Base),
                      power_eval(Y,Base)))) :- !.
power_eval(big_plus(power_rep(I,Base),
                   power_rep(J,Base),
                   zero,
                   Base),
          Base,
          plus(I,J)).

quotient(plus(X,plus(X,Y)),2,plus(X,quotient(Y,2))).
quotient(times(Y,X),Y,if(zerop(Y),zero,fix(X))).

remainder(_,        1,zero) :- !.
remainder(X,        X,zero) :- !.
remainder(times(_,Z),Z,zero) :- !.
remainder(times(Y,_),Y,zero).

reverse_loop(X,Y,  append(reverse(X),Y)) :- !.
reverse_loop(X,[], reverse(X)          ).

times(X, plus(Y,Z), plus(times(X,Y),times(X,Z)) ) :- !.
times(times(X,Y),Z, times(X,times(Y,Z)) ) :- !.
times(X,difference(C,W),difference(times(C,X),times(W,X))) :- !.
times(X,add1(Y),if(numberp(Y), plus(X,times(X,Y)), fix(X))).
```

B.

```
%
% nand
%
% This is a rough approximation to the algorithm presented in:
%
%     "An Algorithm for NAND Decomposition Under Network Constraints,"
%     IEEE Trans. Comp., vol C-18, no. 12, Dec. 1969, p. 1098
%     by E. S. Davidson.
%
% Written by Bruce Holmer
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% I have used the paper's terminology for names used in the program.
%
% The data structure for representing functions and variables is
%           function(FunctionNumber, TrueSet, FalseSet,
%                   ConceivableInputs,
%                   ImmediatePredecessors, ImmediateSuccessors,
%                   Predecessors, Successors)
%
%
% Common names used in the program:
%
%     NumVars          number of variables (signal inputs)
%     NumGs            current number of variables and functions
%     Gs          list of variable and function data
%     Gi,Gj,Gk,Gl      individual variable or function--letter corresponds to
%                 the subscript in the paper (most of the time)
%     Vector,V    vector from a function's true set
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

main :- main(0).

main(N) :-
        init_state(N, NumVars, NumGs, Gs),
        add_necessary_functions(NumVars, NumGs, Gs, NumGs2, Gs2),
        test_bounds(NumVars, NumGs2, Gs2),
        search(NumVars, NumGs2, Gs2).
main(_) :-
        write('Search completed'), nl.

% Test input
% init_state(circuit(NumInputs, NumOutputs, FunctionList))
init_state(0, 2, 3, [              % 2 input xor
                function(2, [1,2], [0,3], [], [], [], [], []),
                function(1, [2,3], [0,1], [], [], [], [], []),
                function(0, [1,3], [0,2], [], [], [], [], [])
                ]) :-
        update_bounds(_, 100, _).
init_state(1, 3, 4, [              % carry circuit
                function(3, [3,5,6,7], [0,1,2,4], [], [], [], [], []),
                function(2, [4,5,6,7], [0,1,2,3], [], [], [], [], []),
                function(1, [2,3,6,7], [0,1,4,5], [], [], [], [], []),
                function(0, [1,3,5,7], [0,2,4,6], [], [], [], [], [])
                ]) :-
        update_bounds(_, 100, _).
init_state(2, 3, 4, [              % example in paper
                function(3, [1,2,4,6,7], [0,3,5], [], [], [], [], []),
                function(2, [4,5,6,7], [0,1,2,3], [], [], [], [], []),
                function(1, [2,3,6,7], [0,1,4,5], [], [], [], [], []),
                function(0, [1,3,5,7], [0,2,4,6], [], [], [], [], [])
                ]) :-
        update_bounds(_, 100, _).
init_state(3, 3, 4, [              % sum (3 input xor)
                function(3, [1,2,4,7], [0,3,5,6], [], [], [], [], []),
                function(2, [4,5,6,7], [0,1,2,3], [], [], [], [], []),
                function(1, [2,3,6,7], [0,1,4,5], [], [], [], [], []),
                function(0, [1,3,5,7], [0,2,4,6], [], [], [], [], [])
                ]) :-
        update_bounds(_, 100, _).
init_state(4, 3, 5, [              % do sum and carry together
                function(4, [3,5,6,7], [0,1,2,4], [], [], [], [], []),
                function(3, [1,2,4,7], [0,3,5,6], [], [], [], [], []),
                function(2, [4,5,6,7], [0,1,2,3], [], [], [], [], []),
                function(1, [2,3,6,7], [0,1,4,5], [], [], [], [], []),
                function(0, [1,3,5,7], [0,2,4,6], [], [], [], [], [])
                ]) :-
        update_bounds(_, 100, _).
init_state(5, 5, 8, [              % 2 bit full adder
                function(7,        % A2 (output)
                        [1,3,4,6,9,11,12,14,16,18,21,23,24,26,29,31],
                        [0,2,5,7,8,10,13,15,17,19,20,22,25,27,28,30],
                        [], [], [], [], []),
                function(6,        % B2 (output)
                        [2,3,5,6,8,9,12,15,17,18,20,21,24,27,30,31],
                        [0,1,4,7,10,11,13,14,16,19,22,23,25,26,28,29],
                        [], [], [], [], []),
                function(5,        % carry-out (output)
                        [7,10,11,13,14,15,19,22,23,25,26,27,28,29,30,31],
                        [0,1,2,3,4,5,6,8,9,12,16,17,18,20,21,24],
                        [], [], [], [], []),
                function(4,        % carry-in
                        [16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31],
                        [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15],
                        [], [], [], [], []),
                function(3,        % B1 input
                        [8,9,10,11,12,13,14,15,24,25,26,27,28,29,30,31],
                        [0,1,2,3,4,5,6,7,16,17,18,19,20,21,22,23],
                        [], [], [], [], []),
                function(2,        % B0 input
                        [4,5,6,7,12,13,14,15,20,21,22,23,28,29,30,31],
                        [0,1,2,3,8,9,10,11,16,17,18,19,24,25,26,27],
                        [], [], [], [], []),
                function(1,        % A1 input
                        [2,3,6,7,10,11,14,15,18,19,22,23,26,27,30,31],
                        [0,1,4,5,8,9,12,13,16,17,20,21,24,25,28,29],
                        [], [], [], [], []),
                function(0,        % A0 input
                        [1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31],
                        [0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30],
                        [], [], [], [], [])
                ]) :-
        update_bounds(_, 21, _).

% Iterate over all the TRUE vectors that need to be covered.
% If no vectors remain to be covered (select_vector fails), then
% the circuit is complete (printout results, update bounds, and
% continue search for a lower cost circuit).
search(NumVars, NumGsIn, GsIn) :-
        select_vector(NumVars, NumGsIn, GsIn, Gj, Vector), !,
        cover_vector(NumVars, NumGsIn, GsIn, Gj, Vector, NumGs, Gs),
        add_necessary_functions(NumVars, NumGs, Gs, NumGsOut, GsOut),
        test_bounds(NumVars, NumGsOut, GsOut),
        search(NumVars, NumGsOut, GsOut).
search(NumVars, NumGs, Gs) :-
        output_results(NumVars, NumGs, Gs),
        update_bounds(NumVars, NumGs, Gs),
        fail.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Given the current solution, pick the best uncovered TRUE vector
% for covering next.
% The selected vector is specified by its vector number and function.
% Select_vector fails if all TRUE vectors are covered.
% Select_vector is determinant (gives only one solution).
select_vector(NumVars, NumGs, Gs, Gj, Vector) :-
        select_vector(Gs, NumVars, NumGs, Gs,
                dummy, 0, nf, 999, Gj, Vector, Type, _), !,
        + Type = cov,
        + Type = nf.

% loop over functions
select_vector([Gk|_], NumVars, _, _, Gj, V, Type, N, Gj, V, Type, N) :-
        function_number(Gk, K),
        K < NumVars.
select_vector([Gk|Gks], NumVars, NumGs, Gs,
                GjIn, Vin, TypeIn, Nin, GjOut, Vout, TypeOut, Nout) :-
        function_number(Gk, K),
        K >= NumVars,
        true_set(Gk, Tk),
        select_vector(Tk, Gk, NumVars, NumGs, Gs,
                GjIn, Vin, TypeIn, Nin, Gj, V, Type, N),
        select_vector(Gks, NumVars, NumGs, Gs,
                Gj, V, Type, N, GjOut, Vout, TypeOut, Nout).

% loop over vectors
select_vector([], _, _, _, _, Gj, V, Type, N, Gj, V, Type, N).
select_vector([V|Vs], Gk, NumVars, NumGs, Gs,
                GjIn, Vin, TypeIn, Nin, GjOut, Vout, TypeOut, Nout) :-
        vector_cover_type(NumVars, Gs, Gk, V, Type, N),
        best_vector(GjIn, Vin, TypeIn, Nin,
                        Gk, V, Type, N,
                        Gj2, V2, Type2, N2),
        select_vector(Vs, Gk, NumVars, NumGs, Gs,
                Gj2, V2, Type2, N2, GjOut, Vout, TypeOut, Nout).

vector_cover_type(NumVars, Gs, Gj, Vector, Type, NumCovers) :-
        immediate_predecessors(Gj, IPs),
        conceivable_inputs(Gj, CIs),
        false_set(Gj, Fj),
        cover_type1(IPs, Gs, Vector, nf, 0, T, N),
        cover_type2(CIs, Gs, NumVars, Fj, Vector, T, N, Type, NumCovers).

cover_type1([], _, _, T, N, T, N).
cover_type1([I|IPs], Gs, V, TypeIn, Nin, TypeOut, Nout) :-
        function(I, Gs, Gi),
        true_set(Gi, Ti),
        + set_member(V, Ti), !,
        false_set(Gi, Fi),
        (set_member(V, Fi) ->
                max_type(TypeIn, cov, Type);
                max_type(TypeIn, exp, Type)),
        N is Nin + 1,
        cover_type1(IPs, Gs, V, Type, N, TypeOut, Nout).
cover_type1([_|IPs], Gs, V, TypeIn, Nin, TypeOut, Nout) :-
        cover_type1(IPs, Gs, V, TypeIn, Nin, TypeOut, Nout).

cover_type2([], _, _, _, _, T, N, T, N).
cover_type2([I|CIs], Gs, NumVars, Fj, V, TypeIn, Nin, TypeOut, Nout) :-
        I < NumVars,
        function(I, Gs, Gi),
        false_set(Gi, Fi),
        set_member(V, Fi), !,
        max_type(TypeIn, var, Type),
        N is Nin + 1,
        cover_type2(CIs, Gs, NumVars, Fj, V, Type, N, TypeOut, Nout).
cover_type2([I|CIs], Gs, NumVars, Fj, V, TypeIn, Nin, TypeOut, Nout) :-
        I >= NumVars,
        function(I, Gs, Gi),
        true_set(Gi, Ti),
        + set_member(V, Ti), !,
        false_set(Gi, Fi),
        (set_member(V, Fi) ->
                (set_subset(Fj, Ti) ->
                        max_type(TypeIn, fcn, Type);
                        max_type(TypeIn, mcf, Type));
                (set_subset(Fj, Ti) ->
                        max_type(TypeIn, exf, Type);
                        max_type(TypeIn, exmcf, Type))),
        N is Nin + 1,
        cover_type2(CIs, Gs, NumVars, Fj, V, Type, N, TypeOut, Nout).
cover_type2([_|CIs], Gs, NumVars, Fj, V, TypeIn, Nin, TypeOut, Nout) :-
        cover_type2(CIs, Gs, NumVars, Fj, V, TypeIn, Nin, TypeOut, Nout).

% The best vector to cover is the one with worst type, or, if types
% are equal, with the least number of possible covers.
best_vector(dummy, _, _, _, Gj2, V2, Type2, N2, Gj2, V2, Type2, N2) :- !.
best_vector(Gj1, V1, Type1, N1, dummy, _, _, _, Gj1, V1, Type1, N1) :- !.
best_vector(Gj1, V1, Type, N1, Gj2, _, Type, N2, Gj1, V1, Type, N1) :-
        function_number(Gj1, J), function_number(Gj2, J),
        N1 < N2, !.
best_vector(Gj1, _, Type, N1, Gj2, V2, Type, N2, Gj2, V2, Type, N2) :-
        function_number(Gj1, J), function_number(Gj2, J),
```

```
        N1 >= N2, !.
best_vector(Gj1, V1, Type, N1, Gj2, _, Type, _, Gj1, V1, Type, N1) :-
        (Type = exp ; Type = var),
        function_number(Gj1, J1), function_number(Gj2, J2),
        J1 > J2, !.
best_vector(Gj1, _, Type, _, Gj2, V2, Type, N2, Gj2, V2, Type, N2) :-
        (Type = exp ; Type = var),
        function_number(Gj1, J1), function_number(Gj2, J2),
        J1 < J2, !.
best_vector(Gj1, V1, Type, N1, Gj2, _, Type, _, Gj1, V1, Type, N1) :-
        + (Type = exp ; Type = var),
        function_number(Gj1, J1), function_number(Gj2, J2),
        J1 < J2, !.
best_vector(Gj1, _, Type, _, Gj2, V2, Type, N2, Gj2, V2, Type, N2) :-
        + (Type = exp ; Type = var),
        function_number(Gj1, J1), function_number(Gj2, J2),
        J1 > J2, !.
best_vector(Gj1, V1, Type1, N1, _, _, Type2, _, Gj1, V1, Type1, N1) :-
        type_order(Type2, Type1), !.
best_vector(_, _, Type1, _, Gj2, V2, Type2, N2, Gj2, V2, Type2, N2) :-
        type_order(Type1, Type2), !.

max_type(T1, T2, T1) :- type_order(T1, T2), !.
max_type(T1, T2, T2) :- + type_order(T1, T2), !.

%  Order of types

type_order(cov, exp).
type_order(cov, var).
type_order(cov, fcn).
type_order(cov, mcf).
type_order(cov, exf).
type_order(cov, exmcf).
type_order(cov, nf).
type_order(exp, var).
type_order(exp, fcn).
type_order(exp, mcf).
type_order(exp, exf).
type_order(exp, exmcf).
type_order(exp, nf).
type_order(var, fcn).
type_order(var, mcf).
type_order(var, exf).
type_order(var, exmcf).
type_order(var, nf).
type_order(fcn, mcf).
type_order(fcn, exf).
type_order(fcn, exmcf).
type_order(fcn, nf).
type_order(mcf, exf).
type_order(mcf, exmcf).
type_order(mcf, nf).
type_order(exf, exmcf).
type_order(exf, nf).
type_order(exmcf, nf).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%  Cover_vector will cover the specified vector and
%  generate new circuit information.
%  Using backtracking, all possible coverings are generated.
%  The ordering of the possible coverings is approximately that
%  given in Davidson's paper, but has been simplified.

cover_vector(NumVars, NumGsIn, GsIn, Gj, Vector, NumGsOut, GsOut) :-
        immediate_predecessors(Gj, IPs),
        conceivable_inputs(Gj, CIs),
        vector_types(Type),
        cover_vector(Type, IPs, CIs, Gj, Vector, NumVars, NumGsIn, GsIn,
                NumGsOut, GsOut).

vector_types(var).
vector_types(exp).
vector_types(fcn).
vector_types(mcf).
vector_types(exf).
vector_types(exmcf).
vector_types(nf).

cover_vector(exp, [I|_], _, Gj, V, _, NumGs, GsIn, NumGs, GsOut) :-
        function(I, GsIn, Gi),
        true_set(Gi, Ti),
        + set_member(V, Ti),
        update_circuit(GsIn, Gi, Gj, V, GsIn, GsOut).
cover_vector(exp, [_|IPs], _, Gj, V, NumVars, NumGs, GsIn, NumGs, GsOut) :-
        cover_vector(exp, IPs, _, Gj, V, NumVars, NumGs, GsIn, NumGs, GsOut).
cover_vector(var, _, [I|_], Gj, V, NumVars, NumGs, GsIn, NumGs, GsOut) :-
        I < NumVars,
        function(I, GsIn, Gi),
        false_set(Gi, Fi),
        set_member(V, Fi),
        update_circuit(GsIn, Gi, Gj, V, GsIn, GsOut).
cover_vector(var, _, [_|CIs], Gj, V, NumVars, NumGs, GsIn, NumGs, GsOut) :-
        cover_vector(var, _, CIs, Gj, V, NumVars, NumGs, GsIn, NumGs, GsOut).
cover_vector(fcn, _, [I|_], Gj, V, NumVars, NumGs, GsIn, NumGs, GsOut) :-
        I >= NumVars,
        function(I, GsIn, Gi),
        false_set(Gi, Fi),
        set_member(V, Fi),
        true_set(Gi, Ti),
        false_set(Gj, Fj),
        set_subset(Fj, Ti),
        update_circuit(GsIn, Gi, Gj, V, GsIn, GsOut).
cover_vector(fcn, _, [_|CIs], Gj, V, NumVars, NumGs, GsIn, NumGs, GsOut) :-
        cover_vector(fcn, _, CIs, Gj, V, NumVars, NumGs, GsIn, NumGs, GsOut).
cover_vector(mcf, _, [I|_], Gj, V, NumVars, NumGs, GsIn, NumGs, GsOut) :-
        I >= NumVars,
        function(I, GsIn, Gi),
        false_set(Gi, Fi),
        set_member(V, Fi),
        true_set(Gi, Ti),
        false_set(Gj, Fj),
        + set_subset(Fj, Ti),
        update_circuit(GsIn, Gi, Gj, V, GsIn, GsOut).
cover_vector(mcf, _, [_|CIs], Gj, V, NumVars, NumGs, GsIn, NumGs, GsOut) :-
        cover_vector(mcf, _, CIs, Gj, V, NumVars, NumGs, GsIn, NumGs, GsOut).
cover_vector(exf, _, [I|_], Gj, V, NumVars, NumGs, GsIn, NumGs, GsOut) :-
        I >= NumVars,
        function(I, GsIn, Gi),
        false_set(Gi, Fi),
        + set_member(V, Fi),
        true_set(Gi, Ti),
        + set_member(V, Ti),
        false_set(Gj, Fj),
        set_subset(Fj, Ti),
        update_circuit(GsIn, Gi, Gj, V, GsIn, GsOut).
cover_vector(exf, _, [_|CIs], Gj, V, NumVars, NumGs, GsIn, NumGs, GsOut) :-
        cover_vector(exf, _, CIs, Gj, V, NumVars, NumGs, GsIn, NumGs, GsOut).
cover_vector(exmcf, _, [I|_], Gj, V, NumVars, NumGs, GsIn, NumGs, GsOut) :-
        I >= NumVars,
        function(I, GsIn, Gi),
        false_set(Gi, Fi),
        + set_member(V, Fi),
        true_set(Gi, Ti),
        + set_member(V, Ti),
        false_set(Gj, Fj),
        + set_subset(Fj, Ti),
        update_circuit(GsIn, Gi, Gj, V, GsIn, GsOut).
cover_vector(exmcf, _, [_|CIs], Gj, V, NumVars, NumGs, GsIn, NumGs, GsOut) :-
        cover_vector(exmcf, _, CIs, Gj, V, NumVars, NumGs, GsIn, NumGs, GsOut).
cover_vector(nf, _, _, Gj, V, NumVars, NumGsIn, GsIn, NumGsOut, GsOut) :-
        NumGsOut is NumGsIn + 1,
        false_set(Gj, Fj),
        new_function_CIs(GsIn,
                function(NumGsIn,Fj,[V],[],[],[],[],[]),
                NumVars, Gs, Gi),
        update_circuit(Gs, Gi, Gj, V, Gs, GsOut).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

update_circuit([], _, _, _, _, []).
update_circuit([function(K,Tk,Fk,CIk,IPk,ISk,Pk,Sk)|GsIn],
                Gi, Gj, V, Gs,
                [function(K,Tko,Fko,CIko,IPko,ISko,Pko,Sko)|GsOut]) :-
        Gi = function(I,_,Fi,_,IPi,ISi,Pi,_),
        Gj = function(J,_,Fj,_,_,_,_,Sj),
        set_union([I], Pi, PiI),
        set_union([J], Sj, SjJ),
        (K = J ->
                set_union(Tk, Fi, Tk2);
                Tk2 = Tk),
        (K = I ->
                set_union(Tk2, Fj, Tk3);
                Tk3 = Tk2),
        ((set_member(K, IPi); set_member(K, ISi)) ->
                set_union(Tk3, [V], Tko);
                Tko = Tk3),
        (K = I ->
                set_union(Fk, [V], Fko);
                Fko = Fk),
        ((set_member(K, Pi); K = I) ->
                set_difference(CIk, SjJ, CIk2);
                CIk2 = CIk),
        ((set_member(I, CIk), set_member(V, Fk)) ->
                set_difference(CIk2, [I], CIk3);
                CIk3 = CIk2),
        (K = I ->
                exclude_if_vector_in_false_set(CIk3, Gs, V, CIk4);
                CIk4 = CIk3),
        (K = J ->
                set_difference(CIk4, [I], CIko);
                CIko = CIk4),
        (K = J ->
                set_union(IPk, [I], IPko);
                IPko = IPk),
        (K = I ->
                set_union(ISk, [J], ISko);
                ISko = ISk),
        (set_member(K, SjJ) ->
                set_union(Pk, PiI, Pko);
                Pko = Pk),
        (set_member(K, PiI) ->
                set_union(Sk, SjJ, Sko);
                Sko = Sk),
        update_circuit(GsIn, Gi, Gj, V, Gs, GsOut).

exclude_if_vector_in_false_set([], _, _, []).
exclude_if_vector_in_false_set([K|CIsIn], Gs, V, CIsOut) :-
        function(K, Gs, Gk),
        false_set(Gk, Fk),
        set_member(V, Fk), !,
        exclude_if_vector_in_false_set(CIsIn, Gs, V, CIsOut).
exclude_if_vector_in_false_set([K|CIsIn], Gs, V, [K|CIsOut]) :-
        exclude_if_vector_in_false_set(CIsIn, Gs, V, CIsOut).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

add_necessary_functions(NumVars, NumGsIn, GsIn, NumGsOut, GsOut) :-
        add_necessary_functions(NumVars, NumVars, NumGsIn, GsIn,
                                NumGsOut, GsOut).

add_necessary_functions(NumGs, _, NumGs, Gs, NumGs, Gs) :- !.
add_necessary_functions(K, NumVars, NumGsIn, GsIn, NumGsOut, GsOut) :-
        function(K, GsIn, Gk),
        function_type(NumVars, NumGsIn, GsIn, Gk, nf, V), !,
        false_set(Gk, Fk),
        new_function_CIs(GsIn,
                function(NumGsIn,Fk,[V],[],[],[],[],[]),
                NumVars, Gs, Gl),
        function(K, Gs, Gk1),
        update_circuit(Gs, Gl, Gk1, V, Gs, Gs1),
        NumGs1 is NumGsIn + 1,
        K1 is K + 1,
        add_necessary_functions(K1, NumVars, NumGs1, Gs1, NumGsOut, GsOut).
```

```
add_necessary_functions(K, NumVars, NumGsIn, GsIn, NumGsOut, GsOut) :-
        K1 is K + 1,
        add_necessary_functions(K1, NumVars, NumGsIn, GsIn, NumGsOut, GsOut).

new_function_CIs(GsIn, function(L,Tl,Fl,_,IPl,ISl,Pl,Sl), NumVars,
                [GlOut|GsOut], GlOut) :-
        new_function_CIs(GsIn, L, Fl, NumVars, GsOut, [], CIlo),
        GlOut = function(L,Tl,Fl,CIlo,IPl,ISl,Pl,Sl).

new_function_CIs([], _, _, _, [], CIl, CIl).
new_function_CIs([function(K,Tk,Fk,CIk,IPk,ISk,Pk,Sk)|GsIn], L, Fl, NumVars,
                [function(K,Tk,Fk,CIko,IPk,ISk,Pk,Sk)|GsOut], CIlIn, CIlOut) :-
        set_intersection(Fl, Fk, []), !,
        (K >= NumVars ->
                set_union(CIk, [L], CIko);
                CIko = CIk),
        new_function_CIs(GsIn, L, Fl, NumVars, GsOut, [K|CIlIn], CIlOut).
new_function_CIs([Gk|GsIn], L, Fl, NumVars, [Gk|GsOut], CIlIn, CIlOut) :-
        new_function_CIs(GsIn, L, Fl, NumVars, GsOut, CIlIn, CIlOut).

function_type(NumVars, NumGs, Gs, Gk, Type, Vector) :-
        true_set(Gk, Tk),
        select_vector(Tk, Gk, NumVars, NumGs, Gs,
                dummy, 0, nf, 999, _, Vector, Type, _).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Cost and constraint predicates:

% very simple bound for now

test_bounds(_, NumGs, _) :-
        access(bound, Bound),
        NumGs < Bound.

update_bounds(_, NumGs, _) :-
        set(bound, NumGs).

% set and access for systems that don't support them
set(N, A) :-
        (recorded(N, _, Ref) -> erase(Ref) ; true),
        recorda(N, A, _).

access(N, A) :-
        recorded(N, A, _).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Output predicates:

%  for now just dump everything

output_results(NumVars, NumGs, Gs) :-
        NumGates is NumGs - NumVars,
        write(NumGates), write(' gates'), nl,
        write_gates(Gs), nl,
        write('searching for a better solution...'), nl, nl.

write_gates([]).
write_gates([Gi|Gs]) :-
        function_number(Gi, I),
        write('gate #'), write(I), write(' inputs:   '),
        immediate_predecessors(Gi, IPi),
        write(IPi), nl,
        write_gates(Gs).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%  Retrieve the specified function from the function list.
%  function(FunctionNumber, FunctionList, Function).
function(I, [Gi|_], Gi) :- function_number(Gi, I), !.
function(I, [_|Gs], Gi) :- function(I, Gs, Gi).

function_number(        function(I,_,_,_,_,_,_,_), I).
true_set(               function(_,T,_,_,_,_,_,_), T).
false_set(              function(_,_,F,_,_,_,_,_), F).
conceivable_inputs(     function(_,_,_,CI,_,_,_,_), CI).
immediate_predecessors( function(_,_,_,_,IP,_,_,_), IP).
immediate_successors(   function(_,_,_,_,_,IS,_,_), IS).
predecessors(           function(_,_,_,_,_,_,P,_), P).
successors(             function(_,_,_,_,_,_,_,S), S).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Set operations assume that the sets are represented by an ordered list
%  of integers.

set_union([],     [],     []).
set_union([],     [X|L2], [X|L2]).
set_union([X|L1], [],     [X|L1]).
set_union([X|L1], [X|L2], [X|L3]) :-       set_union(L1, L2,     L3).
set_union([X|L1], [Y|L2], [X|L3]) :- X < Y, set_union(L1, [Y|L2], L3).
set_union([X|L1], [Y|L2], [Y|L3]) :- X > Y, set_union([X|L1], L2, L3).

set_intersection([],     [],     []).
set_intersection([],     [_|_],  []).
set_intersection([_|_],  [],     []).
set_intersection([X|L1], [X|L2], [X|L3]) :-    set_intersection(L1, L2,     L3).
set_intersection([X|L1], [Y|L2], L3) :- X < Y, set_intersection(L1, [Y|L2], L3).
set_intersection([X|L1], [Y|L2], L3) :- X > Y, set_intersection([X|L1], L2, L3).

set_difference([],     [],     []).
set_difference([],     [_|_],  []).
set_difference([X|L1], [],     [X|L1]).
set_difference([X|L1], [X|L2], L3) :-           set_difference(L1, L2,     L3).
set_difference([X|L1], [Y|L2], [X|L3]) :- X < Y, set_difference(L1, [Y|L2], L3).
set_difference([X|L1], [Y|L2], L3) :-     X > Y, set_difference([X|L1], L2, L3).

set_subset([],     _).
set_subset([X|L1], [X|L2]) :-       set_subset(L1, L2).
set_subset([X|L1], [Y|L2]) :- X > Y, set_subset([X|L1], L2).

set_member(X, [X|_]).
set_member(X, [Y|T]) :- X > Y, set_member(X, T).
```

```
%
%  chat
%
%  chat_parser
%
%  Fernando C. N. Pereira and David H. D. Warren

main :- string(X),
        determinate_say(X,_),
        fail.
main.

%  query set

string([what,rivers,are,there,?]).
string([does,afghanistan,border,china,?]).
string([what,is,the,capital,of,upper_volta,?]).
string([where,is,the,largest,country,?]).
string([which,country,’‘’,s,capital,is,london,?]).
string([which,countries,are,european,?]).
string([how,large,is,the,smallest,american,country,?]).
string([what,is,the,ocean,that,borders,african,countries,
     and,that,borders,asian,countries,?]).
string([what,are,the,capitals,of,the,countries,bordering,the,baltic,?]).
string([which,countries,are,bordered,by,two,seas,?]).
string([how,many,countries,does,the,danube,flow,through,?]).
string([what,is,the,total,area,of,countries,south,of,the,equator,
     and,not,in,australasia,?]).
string([what,is,the,average,area,of,the,countries,in,each,continent,?]).
string([is,there,more,than,one,country,in,each,continent,?]).
string([is,there,some,ocean,that,does,not,border,any,country,?]).
string([what,are,the,countries,from,which,a,river,flows,
     into,the,black_sea,?]).

%  determinate_say

determinate_say(X,Y) :-
  say(X,Y), !.

%------------------------------------------------------------------------------
%
%  xgrun
%
%------------------------------------------------------------------------------

terminal(T,S,S,x(_,terminal,T,X),X).
terminal(T,[T|S],S,X,X) :-
  gap(X).

gap(x(gap,_,_,_)).
gap([]).

virtual(NT,x(_,nonterminal,NT,X),X).

%------------------------------------------------------------------------------
%
%  clotab
%
%------------------------------------------------------------------------------

% normal form masks

is_pp(#(1,_,_,_)).

is_pred(#(_,1,_,_)).

is_trace(#(_,_,1,_)).

is_adv(#(_,_,_,1)).

trace(#(_,_,1,_),#(0,0,0,0)).

trace(#(0,0,1,0)).

adv(#(0,0,0,1)).

empty(#(0,0,0,0)).

np_all(#(1,1,1,0)).

s_all(#(1,0,1,1)).

np_no_trace(#(1,1,0,0)).

% mask operations

myplus(#(B1,B2,B3,B4),#(C1,C2,C3,C4),#(D1,D2,D3,D4)) :-
  or(B1,C1,D1),
  or(B2,C2,D2),
  or(B3,C3,D3),
  or(B4,C4,D4).

minus(#(B1,B2,B3,B4),#(C1,C2,C3,C4),#(D1,D2,D3,D4)) :-
  anot(B1,C1,D1),
  anot(B2,C2,D2),
  anot(B3,C3,D3),
  anot(B4,C4,D4).

or(1,_,1).
or(0,1,1).
or(0,0,0).

anot(X,0,X).
anot(X,1,0).

% noun phrase position features

role(subj,_,#(1,0,0)).
role(compl,_,#(0,_,_)).
role(undef,main,#(_,0,_)).
role(undef,aux,#(0,_,_)).
role(undef,decl,_).
role(nil,_,_).

subj_case(#(1,0,0)).
verb_case(#(0,1,0)).
prep_case(#(0,0,1)).
compl_case(#(0,_,_)).

%------------------------------------------------------------------------------
%
%  newg
%
%------------------------------------------------------------------------------

say(X,Y) :-
  sentence(Y,X,[],[],[]).

sentence(B,C,D,E,F) :-
  declarative(B,C,G,E,H),
  terminator(.,G,D,H,F).
sentence(B,C,D,E,F) :-
  wh_question(B,C,G,E,H),
  terminator(?,G,D,H,F).
sentence(B,C,D,E,F) :-
  topic(C,G,E,H),
  wh_question(B,G,I,H,J),
  terminator(?,I,D,J,F).
sentence(B,C,D,E,F) :-
  yn_question(B,C,G,E,H),
  terminator(?,G,D,H,F).
sentence(B,C,D,E,F) :-
  imperative(B,C,G,E,H),
  terminator(!,G,D,H,F).

pp(B,C,D,E,F,F,G,H) :-
  virtual(pp(B,C,D,E),G,H).
pp(pp(B,C),D,E,F,G,H,I,J) :-
  prep(B,G,K,I,L),
  prep_case(M),
  np(C,N,M,O,D,E,F,K,H,L,J).

topic(B,C,D,x(gap,nonterminal,pp(E,compl,F,G),H)) :-
  pp(E,compl,F,G,B,I,D,J),
  opt_comma(I,C,J,H).

opt_comma(B,C,D,E) :-
  ‘(’,’,B,C,D,E).
opt_comma(B,B,C,C).

declarative(decl(B),C,D,E,F) :-
  s(B,G,C,D,E,F).

wh_question(whq(B,C),D,E,F,G) :-
  variable_q(B,H,I,J,D,K,F,L),
  question(I,J,C,K,E,L,G).

np(B,C,D,E,F,G,H,I,J,K) :-
  virtual(np(B,C,D,E,F,G,H),J,K).
np(np(B,C,[]),B,D,def,E,F,G,H,I,J,K) :-
  is_pp(F),
  pers_pron(C,B,L,H,I,J,K),
  empty(G),
  role(L,decl,D).
np(np(B,C,D),B,E,F,G,H,I,J,K,L,M) :-
  is_pp(H),
  np_head(C,B,F+N,O,D,J,P,L,Q),
  np_all(R),
  np_compls(N,B,G,O,R,I,P,K,Q,M).
np(part(B,C),3+D,E,indef,F,G,H,I,J,K,L) :-
  is_pp(G),
  determiner(B,D,indef,I,M,K,N),
  ‘(of,M,O,N,P),
  s_all(Q),
  prep_case(R),
  np(C,3+plu,R,def,F,Q,H,O,J,P,L).

variable_q(B,C,D,E,F,G,H,x(gap,nonterminal,np(I,C,E,J,K,L,M),N)) :-
  whq(B,C,I,D,F,G,H,N),
  trace(L,M).
variable_q(B,C,compl,D,E,F,G,x(gap,nonterminal,pp(pp(H,I),compl,J,K),L)) :-
  prep(H,E,M,G,N),
  whq(B,C,I,O,M,F,N,L),
  trace(J,K),
  compl_case(D).
variable_q(B,C,compl,D,E,F,G,x(gap,nonterminal,
      adv_phrase(pp(H,np(C,np_head(int_det(B),[],I),[])),J,K),L)) :-
  context_pron(H,I,E,F,G,L),
  trace(J,K),
  verb_case(D).
variable_q(B,C,compl,D,E,F,G,
      x(gap,nonterminal,predicate(adj,value(H,wh(B)),I),J)) :-
  ‘(how,E,K,G,L),
  adj(quant,H,K,F,L,J),
  empty(I),
  verb_case(D).

adv_phrase(B,C,D,E,E,F,G) :-
  virtual(adv_phrase(B,C,D),F,G).
adv_phrase(pp(B,C),D,E,F,G,H,I) :-
  loc_pred(B,F,J,H,K),
  pp(pp(prep(of),C),compl,D,E,J,G,K,I).

predicate(B,C,D,E,E,F,G) :-
  virtual(predicate(B,C,D),F,G).
predicate(B,C,D,E,F,G,H) :-
  adj_phrase(C,D,E,F,G,H).
predicate(neg,B,C,D,E,F,G) :-
  s_all(H),
  pp(B,compl,H,C,D,E,F,G).
predicate(B,C,D,E,F,G,H) :-
  s_all(I),
  adv_phrase(C,I,D,E,F,G,H).

whq(B,C,D,undef,E,F,G,H) :-
  int_det(B,C,E,I,G,J),
  s_all(K),
```

B.

```
   np(D,C,L,M,subj,K,N,I,F,J,H).
whq(B,3+C,np(3+C,wh(B),[]),D,E,F,G,H) :-
   int_pron(D,E,F,G,H).

int_det(B,3+C,D,E,F,G) :-
   whose(B,C,D,E,F,G).
int_det(B,3+C,D,E,F,G) :-
   int_art(B,C,D,E,F,G).

gen_marker(B,B,C,D) :-
   virtual(gen_marker,C,D).
gen_marker(B,C,D,E) :-
   '(''',B,F,D,G),
   an_s(F,C,G,E).

whose(B,C,D,E,F,x(nogap,nonterminal,np_head0(wh(B),C,proper),
          x(nogap,nonterminal,gen_marker,G))) :-
   '(whose,D,E,F,G).

question(B,C,D,E,F,G,H) :-
   subj_question(B),
   role(subj,I,C),
   s(D,J,E,F,G,H).
question(B,C,D,E,F,G,H) :-
   fronted_verb(B,C,E,I,G,J),
   s(D,K,I,F,J,H).

det(B,C,D,E,E,F,G) :-
   virtual(det(B,C,D),F,G).
det(det(B),C,D,E,F,G,H) :-
   terminal(I,E,F,G,H),
   det(I,C,B,D).
det(generic,B,generic,C,C,D,D).

int_art(B,C,D,E,F,x(nogap,nonterminal,det(G,C,def),H)) :-
   int_art(B,C,G,D,E,F,H).

subj_question(subj).
subj_question(undef).

yn_question(q(B),C,D,E,F) :-
   fronted_verb(nil,G,C,H,E,I),
   s(B,J,H,D,I,F).

verb_form(B,C,D,E,F,F,G,H) :-
   virtual(verb_form(B,C,D,E),G,H).
verb_form(B,C,D,E,F,G,H,I) :-
   terminal(J,F,G,H,I),
   verb_form(J,B,C,D).

neg(B,C,D,D,E,F) :-
   virtual(neg(B,C),E,F).
neg(aux+B,neg,C,D,E,F) :-
   '(not,C,D,E,F).
neg(B,pos,C,C,D,D).

fronted_verb(B,C,D,E,F,x(gap,nonterminal,verb_form(G,H,I,J),
           x(nogap,nonterminal,neg(K,L),M))) :-
   verb_form(G,H,I,N,D,O,F,P),
   verb_type(G,aux+Q),
   role(B,J,C),
   neg(R,L,O,E,P,M).

imperative(imp(B),C,D,E,F) :-
   imperative_verb(C,G,E,H),
   s(B,I,G,D,H,F).

imperative_verb(B,C,D,x(nogap,terminal,you,x(nogap,nonterminal,
                verb_form(E,imp+fin,2+sin,main),F))) :-
   verb_form(E,inf,G,H,B,C,D,F).

s(s(B,C,D,E),F,G,H,I,J) :-
   subj(B,K,L,G,M,I,N),
   verb(C,K,L,O,M,P,N,Q),
   empty(R),
   s_all(S),
   verb_args(L,O,D,R,T,P,U,Q,V),
   minus(S,T,W),
   myplus(S,T,X),
   verb_mods(E,W,X,F,U,H,V,J).

subj(there,B,C+be,D,E,F,G) :-
   '(there,D,E,F,G).
subj(B,C,D,E,F,G,H) :-
   s_all(I),
   subj_case(J),
   np(B,C,J,K,subj,I,L,E,F,G,H).

np_head(B,C,D,E,F,G,H,I,J) :-
   np_head0(K,L,M,G,N,I,O),
   possessive(K,L,M,P,P,B,C,D,E,F,N,H,O,J).

np_head0(B,C,D,E,E,F,G) :-
   virtual(np_head0(B,C,D),F,G).
np_head0(name(B),3+sin,def+proper,C,D,E,F) :-
   name(B,C,D,E,F).
np_head0(np_head(B,C,D),3+E,F+common,G,H,I,J) :-
   determiner(B,E,F,G,K,I,L),
   adjs(C,K,M,L,N),
   noun(D,E,M,H,N,J).
np_head0(B,C,def+proper,D,E,F,x(nogap,nonterminal,gen_marker,G)) :-
   poss_pron(B,C,D,E,F,G).
np_head0(np_head(B,[],C),3+sin,indef+common,D,E,F,G) :-
   quantifier_pron(B,C,D,E,F,G).

np_compls(proper,B,C,[],D,E,F,F,G,G) :-
   empty(E).
np_compls(common,B,C,D,E,F,G,H,I,J) :-
   np_all(K),
   np_mods(B,C,L,D,E,M,K,N,G,O,I,P),
   relative(B,L,M,N,F,O,H,P,J).

possessive(B,C,D,[],E,F,G,H,I,J,K,L,M,N) :-
```

```
   gen_case(K,O,M,P),
   np_head0(Q,R,S,O,T,P,U),
   possessive(Q,R,S,V,[pp(poss,np(C,B,E))|V],F,G,H,I,J,T,L,U,N).
possessive(B,C,D,E,F,B,C,D,E,F,G,G,H,H).

gen_case(B,C,D,x(nogap,terminal,the,E)) :-
   gen_marker(B,C,D,E).

an_s(B,C,D,E) :-
   '(s,B,C,D,E).
an_s(B,B,C,C).

determiner(B,C,D,E,F,G,H) :-
   det(B,C,D,E,F,G,H).
determiner(B,C,D,E,F,G,H) :-
   quant_phrase(B,C,D,E,F,G,H).

quant_phrase(quant(B,C),D,E,F,G,H,I) :-
   quant(B,E,F,J,H,K),
   number(C,D,J,G,K,I).

quant(B,indef,C,D,E,F) :-
   neg_adv(G,B,C,H,E,I),
   comp_adv(G,H,J,I,K),
   '(than,J,D,K,F).
quant(B,indef,C,D,E,F) :-
   '(at,C,G,E,H),
   sup_adv(I,G,D,H,F),
   sup_op(I,B).
quant(the,def,B,C,D,E) :-
   '(the,B,C,D,E).
quant(same,indef,B,B,C,C).

neg_adv(B,not+B,C,D,E,F) :-
   '(not,C,D,E,F).
neg_adv(B,B,C,C,D,D).

sup_op(least,not+less).
sup_op(most,not+more).

np_mods(B,C,D,[E|F],G,H,I,J,K,L,M,N) :-
   np_mod(B,C,E,G,O,K,P,M,Q),
   trace(R),
   myplus(R,O,S),
   minus(G,S,T),
   myplus(O,G,U),
   np_mods(B,C,D,F,T,H,U,J,P,L,Q,N).
np_mods(B,C,D,D,E,E,F,F,G,G,H,H).

np_mod(B,C,D,E,F,G,H,I,J) :-
   pp(D,C,E,F,G,H,I,J).
np_mod(B,C,D,E,F,G,H,I,J) :-
   reduced_relative(B,D,E,F,G,H,I,J).

verb_mods([B|C],D,E,F,G,H,I,J) :-
   verb_mod(B,D,K,G,L,I,M),
   trace(N),
   myplus(N,K,O),
   minus(D,O,P),
   myplus(K,D,Q),
   verb_mods(C,P,Q,F,L,H,M,J).
verb_mods([],B,C,C,D,D,E,E).

verb_mod(B,C,D,E,F,G,H) :-
   adv_phrase(B,C,D,E,F,G,H).
verb_mod(B,C,D,E,F,G,H) :-
   is_adv(C),
   adverb(B,E,F,G,H),
   empty(D).
verb_mod(B,C,D,E,F,G,H) :-
   pp(B,compl,C,D,E,F,G,H).

adjs([B|C],D,E,F,G) :-
   pre_adj(B,D,H,F,I),
   adjs(C,H,E,I,G).
adjs([],B,B,C,C).

pre_adj(B,C,D,E,F) :-
   adj(G,B,C,D,E,F).
pre_adj(B,C,D,E,F) :-
   sup_phrase(B,C,D,E,F).

sup_phrase(sup(most,B),C,D,E,F) :-
   sup_adj(B,C,D,E,F).
sup_phrase(sup(B,C),D,E,F,G) :-
   sup_adv(B,D,I,F,J),
   adj(quant,C,I,E,J,G).

comp_phrase(comp(B,C,D),E,F,G,H,I) :-
   comp(B,C,F,J,H,K),
   np_no_trace(L),
   prep_case(M),
   np(D,N,M,O,compl,L,E,J,G,K,I).

comp(B,C,D,E,F,G) :-
   comp_adv(B,D,H,F,I),
   adj(quant,C,H,J,I,K),
   '(than,J,E,K,G).
comp(more,B,C,D,E,F) :-
   rel_adj(B,C,G,E,H),
   '(than,G,D,H,F).
comp(same,B,C,D,E,F) :-
   '(as,C,G,E,H),
   adj(quant,B,G,I,H,J),
   '(as,I,D,J,F).

relative(B,[C],D,E,F,G,H,I,J) :-
   is_pred(D),
   rel_conj(B,K,C,F,G,H,I,J).
relative(B,[],C,D,D,E,E,F,F).

rel_conj(B,C,D,E,F,G,H,I) :-
   rel(B,J,K,F,L,H,M),
```

```
   rel_rest(B,C,J,D,K,E,L,G,M,I).

rel_rest(B,C,D,E,F,G,H,I,J,K) :-
   conj(C,L,D,M,E,H,N,J,O),
   rel_conj(B,L,M,G,N,I,O,K).
rel_rest(B,C,D,D,E,E,F,F,G,G).

rel(B,rel(C,D),E,F,G,H,I) :-
   open(F,J,H,K),
   variable(B,C,J,L,K,M),
   s(D,N,L,O,M,P),
   trace(Q),
   minus(N,Q,E),
   close(O,G,P,I).

variable(B,C,D,E,F,x(gap,nonterminal,np(np(B,wh(C),[]),B,G,H,I,J,K),L)) :-
   '(that,D,E,F,L),
   trace(J,K).
variable(B,C,D,E,F,x(gap,nonterminal,np(G,H,I,J,K,L,M),N)) :-
   wh(C,B,G,H,I,D,E,F,N),
   trace(L,M).
variable(B,C,D,E,F,x(gap,nonterminal,pp(pp(G,H),compl,I,J),K)) :-
   prep(G,D,L,F,M),
   wh(C,B,H,N,O,L,E,M,K),
   trace(I,J),
   compl_case(O).

wh(B,C,np(C,wh(B),[]),C,D,E,F,G,H) :-
   rel_pron(I,E,F,G,H),
   role(I,decl,D).
wh(B,C,np(D,E,[pp(F,G)]),D,H,I,J,K,L) :-
   np_head0(E,D,M+common,I,N,K,O),
   prep(F,N,P,O,Q),
   wh(B,C,G,R,S,P,J,Q,L).
wh(B,C,D,E,F,G,H,I,J) :-
   whose(B,C,G,K,I,L),
   s_all(M),
   np(D,E,F,def,subj,M,N,K,H,L,J).

reduced_relative(B,C,D,E,F,G,H,I) :-
   is_pred(D),
   reduced_rel_conj(B,J,C,E,F,G,H,I).

reduced_rel_conj(B,C,D,E,F,G,H,I) :-
   reduced_rel(B,J,K,F,L,H,M),
   reduced_rel_rest(B,C,J,D,K,E,L,G,M,I).

reduced_rel_rest(B,C,D,E,F,G,H,I,J,K) :-
   conj(C,L,D,M,E,H,N,J,O),
   reduced_rel_conj(B,L,M,G,N,I,O,K).
reduced_rel_rest(B,C,D,D,E,E,F,F,G,G).

reduced_rel(B,reduced_rel(C,D),E,F,G,H,I) :-
   open(F,J,H,K),
   reduced_wh(B,C,J,L,K,M),
   s(D,N,L,O,M,P),
   trace(Q),
   minus(N,Q,E),
   close(O,G,P,I).

reduced_wh(B,C,D,E,F,x(nogap,nonterminal,
         np(np(B,wh(C),[]),B,G,H,I,J,K),x(nogap,nonterminal,
         verb_form(be,pres+fin,B,main),x(nogap,nonterminal,
         neg(L,M),x(nogap,nonterminal,predicate(M,N,O),P))))) :-
   neg(Q,M,D,R,F,S),
   predicate(M,N,O,R,E,S,P),
   trace(J,K),
   subj_case(G).
reduced_wh(B,C,D,E,F,x(nogap,nonterminal,
         np(np(B,wh(C),[]),B,G,H,I,J,K),x(nogap,nonterminal,
         verb(L,M,N,O),P))) :-
   participle(L,N,O,D,E,F,P),
   trace(J,K),
   subj_case(G).
reduced_wh(B,C,D,E,F,x(nogap,nonterminal,
         np(G,H,I,J,K,L,M),x(gap,nonterminal,
         np(np(B,wh(C),[]),B,N,O,P,Q,R),S))) :-
   s_all(T),
   subj_case(I),
   verb_case(N),
   np(G,H,U,J,subj,T,V,D,E,F,S),
   trace(L,M),
   trace(Q,R).

verb(B,C,D,E,F,F,G,H) :-
   virtual(verb(B,C,D,E),G,H).
verb(verb(B,C,D+fin,E,F),G,H,C,I,J,K,L) :-
   verb_form(M,D+fin,G,N,I,O,K,P),
   verb_type(M,Q),
   neg(Q,F,O,R,P,S),
   rest_verb(N,M,B,C,E,R,J,S,L),
   verb_type(B,H).

rest_verb(aux,have,B,C,[perf|D],E,F,G,H) :-
   verb_form(I,past+part,J,K,E,L,G,M),
   have(I,B,C,D,L,F,M,H).
rest_verb(aux,be,B,C,D,E,F,G,H) :-
   verb_form(I,J,K,L,E,M,G,N),
   be(J,I,B,C,D,M,F,N,H).
rest_verb(aux,do,B,active,[],C,D,E,F) :-
   verb_form(B,inf,G,H,C,D,E,F).
rest_verb(main,B,B,active,[],C,C,D,D).

have(be,B,C,D,E,F,G,H) :-
   verb_form(I,J,K,L,E,M,G,N),
   be(J,I,B,C,D,M,F,N,H).
have(B,B,active,[],C,C,D,D).

be(past+part,B,B,passive,[],C,C,D,D).
be(pres+part,B,C,D,[prog],E,F,G,H) :-
   passive(B,C,D,E,F,G,H).

passive(be,B,passive,C,D,E,F) :-
   verb_form(B,past+part,G,H,C,D,E,F),
   verb_type(B,I),
   passive(I).
passive(B,B,active,C,C,D,D).

participle(verb(B,C,inf,D,E),F,C,G,H,I,J) :-
   neg(K,E,G,L,I,M),
   verb_form(B,N,O,P,L,H,M,J),
   participle(N,C,D),
   verb_type(B,F).

passive(B+trans).
passive(B+ditrans).

participle(pres+part,active,[prog]).
participle(past+part,passive,[]).

close(B,B,C,D) :-
   virtual(close,C,D).

open(B,B,C,x(gap,nonterminal,close,C)).

verb_args(B+C,D,E,F,G,H,I,J,K) :-
   advs(E,L,M,H,N,J,O),
   verb_args(C,D,L,F,G,N,I,O,K).
verb_args(trans,active,[arg(dir,B)],C,D,E,F,G,H) :-
   verb_arg(np,B,D,E,F,G,H).
verb_args(ditrans,B,[arg(C,D)|E],F,G,H,I,J,K) :-
   verb_arg(np,D,L,H,M,J,N),
   object(C,E,L,G,M,I,N,K).
verb_args(be,B,[void],C,C,D,E,F,G) :-
   terminal(there,D,E,F,G).
verb_args(be,B,[arg(predicate,C)],D,E,F,G,H,I) :-
   pred_conj(J,C,E,F,G,H,I).
verb_args(be,B,[arg(dir,C)],D,E,F,G,H,I) :-
   verb_arg(np,C,E,F,G,H,I).
verb_args(have,active,[arg(dir,B)],C,D,E,F,G,H) :-
   verb_arg(np,B,D,E,F,G,H).
verb_args(B,C,[],D,D,E,E,F,F) :-
   no_args(B).

object(B,C,D,E,F,G,H,I) :-
   adv(J),
   minus(J,D,K),
   advs(C,L,K,F,M,H,N),
   obj(B,L,D,E,M,G,N,I).

obj(ind,[arg(dir,B)],C,D,E,F,G,H) :-
   verb_arg(np,B,D,E,F,G,H).
obj(dir,[],B,B,C,C,D,D).

pred_conj(B,C,D,E,F,G,H) :-
   predicate(I,J,K,E,L,G,M),
   pred_rest(B,J,C,K,D,L,F,M,H).

pred_rest(B,C,D,E,F,G,H,I,J) :-
   conj(B,K,C,L,D,G,M,I,N),
   pred_conj(K,L,F,M,H,N,J).
pred_rest(B,C,C,D,D,E,E,F,F).

verb_arg(np,B,C,D,E,F,G) :-
   s_all(H),
   verb_case(I),
   np(B,J,I,K,compl,H,C,D,E,F,G).

advs([B|C],D,E,F,G,H,I) :-
   is_adv(E),
   adverb(B,F,J,H,K),
   advs(C,D,E,J,G,K,I).
advs(B,B,C,D,D,E,E).

adj_phrase(B,C,D,E,F,G) :-
   adj(H,B,D,E,F,G),
   empty(C).
adj_phrase(B,C,D,E,F,G) :-
   comp_phrase(B,C,D,E,F,G).

no_args(trans).
no_args(ditrans).
no_args(intrans).

conj(conj(B,C),conj(B,D),E,F,conj(B,E,F),G,H,I,J) :-
   conj(B,C,D,G,H,I,J).

noun(B,C,D,E,F,G) :-
   terminal(H,D,E,F,G),
   noun_form(H,B,C).

adj(B,adj(C),D,E,F,G) :-
   terminal(C,D,E,F,G),
   adj(C,B).

prep(prep(B),C,D,E,F) :-
   terminal(B,C,D,E,F),
   prep(B).

rel_adj(adj(B),C,D,E,F) :-
   terminal(G,C,D,E,F),
   rel_adj(G,B).

sup_adj(adj(B),C,D,E,F) :-
   terminal(G,C,D,E,F),
   sup_adj(G,B).

comp_adv(less,B,C,D,E) :-
   '(less,B,C,D,E).
comp_adv(more,B,C,D,E) :-
   '(more,B,C,D,E).

sup_adv(least,B,C,D,E) :-
   '(least,B,C,D,E).
sup_adv(most,B,C,D,E) :-
   '(most,B,C,D,E).
```

B.

```
rel_pron(B,C,D,E,F) :-
   terminal(G,C,D,E,F),
   rel_pron(G,B).

name(B,C,D,E,F) :-
   opt_the(C,G,E,H),
   terminal(B,G,D,H,F),
   name(B).

int_art(B,plu,quant(same,wh(B)),C,D,E,F) :-
   ‘(how,C,G,E,H),
   ‘(many,G,D,H,F).
int_art(B,C,D,E,F,G,H) :-
   terminal(I,E,F,G,H),
   int_art(I,B,C,D).

int_pron(B,C,D,E,F) :-
   terminal(G,C,D,E,F),
   int_pron(G,B).

adverb(adv(B),C,D,E,F) :-
   terminal(B,C,D,E,F),
   adverb(B).

poss_pron(pronoun(B),C+D,E,F,G,H) :-
   terminal(I,E,F,G,H),
   poss_pron(I,B,C,D).

pers_pron(pronoun(B),C+D,E,F,G,H,I) :-
   terminal(J,F,G,H,I),
   pers_pron(J,B,C,D,E).

quantifier_pron(B,C,D,E,F,G) :-
   terminal(H,D,E,F,G),
   quantifier_pron(H,B,C).

context_pron(prep(in),place,B,C,D,E) :-
   ‘(where,B,C,D,E).
context_pron(prep(at),time,B,C,D,E) :-
   ‘(when,B,C,D,E).

number(nb(B),C,D,E,F,G) :-
   terminal(H,D,E,F,G),
   number(H,B,C).

terminator(B,C,D,E,F) :-
   terminal(G,C,D,E,F),
   terminator(G,B).

opt_the(B,B,C,C).
opt_the(B,C,D,E) :-
   ‘(the,B,C,D,E).

conj(B,list,list,C,D,E,F) :-
   terminal(’,’,C,D,E,F).
conj(B,list,’end’,C,D,E,F) :-
   terminal(B,C,D,E,F),
   conj(B).

loc_pred(B,C,D,E,F) :-
   terminal(G,C,D,E,F),
   loc_pred(G,B).

‘(B,C,D,E,F) :-
   terminal(B,C,D,E,F),
   ‘(B).

%----------------------------------------------------------------------------
%
%  newdic
%
%----------------------------------------------------------------------------

word(Word) :- ‘(Word).
word(Word) :- conj(Word).
word(Word) :- adverb(Word).
word(Word) :- sup_adj(Word,_).
word(Word) :- rel_adj(Word,_).
word(Word) :- adj(Word,_).
word(Word) :- name(Word).
word(Word) :- terminator(Word,_).
word(Word) :- pers_pron(Word,_,_,_,_).
word(Word) :- poss_pron(Word,_,_,_).
word(Word) :- rel_pron(Word,_).
word(Word) :- verb_form(Word,_,_,_).
word(Word) :- noun_form(Word,_,_).
word(Word) :- prep(Word).
word(Word) :- quantifier_pron(Word,_,_).
word(Word) :- number(Word,_,_).
word(Word) :- det(Word,_,_,_).
word(Word) :- int_art(Word,_,_,_).
word(Word) :- int_pron(Word,_).
word(Word) :- loc_pred(Word,_).

‘(how).
‘(whose).
‘(there).
‘(of).
‘(’‘’).          % use ‘ instead of ’ to help assembler
‘(’,’).
‘(s).
‘(than).
‘(at).
‘(the).
‘(not).
‘(as).
‘(that).
‘(less).
‘(more).
‘(least).
‘(most).
‘(many).
‘(where).
‘(when).

conj(and).
conj(or).

int_pron(what,undef).
int_pron(which,undef).
int_pron(who,subj).
int_pron(whom,compl).

int_art(what,X,_,int_det(X)).
int_art(which,X,_,int_det(X)).

det(the,No,the(No),def).
det(a,sin,a,indef).
det(an,sin,a,indef).
det(every,sin,every,indef).
det(some,_,some,indef).
det(any,_,any,indef).
det(all,plu,all,indef).
det(each,sin,each,indef).
det(no,_,no,indef).

number(W,I,Nb) :-
   tr_number(W,I),
   ag_number(I,Nb).

tr_number(nb(I),I).
tr_number(one,1).
tr_number(two,2).
tr_number(three,3).
tr_number(four,4).
tr_number(five,5).
tr_number(six,6).
tr_number(seven,7).
tr_number(eight,8).
tr_number(nine,9).
tr_number(ten,10).

ag_number(1,sin).
ag_number(N,plu) :- N>1.

quantifier_pron(everybody,every,person).
quantifier_pron(everyone,every,person).
quantifier_pron(everything,every,thing).
quantifier_pron(somebody,some,person).
quantifier_pron(someone,some,person).
quantifier_pron(something,some,thing).
quantifier_pron(anybody,any,person).
quantifier_pron(anyone,any,person).
quantifier_pron(anything,any,thing).
quantifier_pron(nobody,no,person).
quantifier_pron(nothing,no,thing).

prep(as).
prep(at).
prep(of).
prep(to).
prep(by).
prep(with).
prep(in).
prep(on).
prep(from).
prep(into).
prep(through).

noun_form(Plu,Sin,plu) :- noun_plu(Plu,Sin).
noun_form(Sin,Sin,sin) :- noun_sin(Sin).
noun_form(proportion,proportion,_).
noun_form(percentage,percentage,_).

root_form(1+sin).
root_form(2+_).
root_form(1+plu).
root_form(3+plu).

verb_root(be).
verb_root(have).
verb_root(do).
verb_root(border).
verb_root(contain).
verb_root(drain).
verb_root(exceed).
verb_root(flow).
verb_root(rise).

regular_pres(have).
regular_pres(do).
regular_pres(rise).
regular_pres(border).
regular_pres(contain).
regular_pres(drain).
regular_pres(exceed).
regular_pres(flow).

regular_past(had,have).
regular_past(bordered,border).
regular_past(contained,contain).
regular_past(drained,drain).
regular_past(exceeded,exceed).
regular_past(flowed,flow).

rel_pron(who,subj).
rel_pron(whom,compl).
rel_pron(which,undef).

poss_pron(my,_,1,sin).
poss_pron(your,_,2,_).
poss_pron(his,masc,3,sin).
poss_pron(her,fem,3,sin).
poss_pron(its,neut,3,sin).
poss_pron(our,_,1,plu).
poss_pron(their,_,3,plu).
```

```
pers_pron(i,_,1,sin,subj).
pers_pron(you,_,2,_,_).
pers_pron(he,masc,3,sin,subj).
pers_pron(she,fem,3,sin,subj).
pers_pron(it,neut,3,sin,_).
pers_pron(we,_,1,plu,subj).
pers_pron(them,_,3,plu,subj).
pers_pron(me,_,1,sin,compl(_)).
pers_pron(him,masc,3,sin,compl(_)).
pers_pron(her,fem,3,sin,compl(_)).
pers_pron(us,_,1,plu,compl(_)).
pers_pron(them,_,3,plu,compl(_)).

terminator(.,_).
terminator(?,?).
terminator(!,!).

name(_).

% ==============================

% specialised dictionary

loc_pred(east,prep(eastof)).
loc_pred(west,prep(westof)).
loc_pred(north,prep(northof)).
loc_pred(south,prep(southof)).

adj(minimum,restr).
adj(maximum,restr).
adj(average,restr).
adj(total,restr).
adj(african,restr).
adj(american,restr).
adj(asian,restr).
adj(european,restr).
adj(great,quant).
adj(big,quant).
adj(small,quant).
adj(large,quant).
adj(old,quant).
adj(new,quant).
adj(populous,quant).

rel_adj(greater,great).
rel_adj(less,small).
rel_adj(bigger,big).
rel_adj(smaller,small).
rel_adj(larger,large).
rel_adj(older,old).
rel_adj(newer,new).

sup_adj(biggest,big).
sup_adj(smallest,small).
sup_adj(largest,large).
sup_adj(oldest,old).
sup_adj(newest,new).

noun_sin(average).
noun_sin(total).
noun_sin(sum).
noun_sin(degree).
noun_sin(sqmile).
noun_sin(ksqmile).
noun_sin(thousand).
noun_sin(million).
noun_sin(time).
noun_sin(place).
noun_sin(area).
noun_sin(capital).
noun_sin(city).
noun_sin(continent).
noun_sin(country).
noun_sin(latitude).
noun_sin(longitude).
noun_sin(ocean).
noun_sin(person).
noun_sin(population).
noun_sin(region).
noun_sin(river).
noun_sin(sea).
noun_sin(seamass).
noun_sin(number).

noun_plu(averages,average).
noun_plu(totals,total).
noun_plu(sums,sum).
noun_plu(degrees,degree).
noun_plu(sqmiles,sqmile).
noun_plu(ksqmiles,ksqmile).
noun_plu(million,million).
noun_plu(thousand,thousand).
noun_plu(times,time).
noun_plu(places,place).
noun_plu(areas,area).
noun_plu(capitals,capital).
noun_plu(cities,city).
noun_plu(continents,continent).
noun_plu(countries,country).
noun_plu(latitudes,latitude).
noun_plu(longitudes,longitude).
noun_plu(oceans,ocean).
noun_plu(persons,person).  noun_plu(people,person).
noun_plu(populations,population).
noun_plu(regions,region).
noun_plu(rivers,river).
noun_plu(seas,sea).
noun_plu(seamasses,seamass).
noun_plu(numbers,number).

verb_form(V,V,inf,_) :- verb_root(V).
verb_form(V,V,pres+fin,Agmt) :-
   regular_pres(V),
   root_form(Agmt),
   verb_root(V).
verb_form(Past,Root,past+_,_) :-
   regular_past(Past,Root).

verb_form(am,be,pres+fin,1+sin).
verb_form(are,be,pres+fin,2+sin).
verb_form(is,be,pres+fin,3+sin).
verb_form(are,be,pres+fin,_+plu).
verb_form(was,be,past+fin,1+sin).
verb_form(were,be,past+fin,2+sin).
verb_form(was,be,past+fin,3+sin).
verb_form(were,be,past+fin,_+plu).
verb_form(been,be,past+part,_).
verb_form(being,be,pres+part,_).
verb_form(has,have,pres+fin,3+sin).
verb_form(having,have,pres+part,_).
verb_form(does,do,pres+fin,3+sin).
verb_form(did,do,past+fin,_).
verb_form(doing,do,pres+part,_).
verb_form(done,do,past+part,_).
verb_form(flows,flow,pres+fin,3+sin).
verb_form(flowing,flow,pres+part,_).
verb_form(rises,rise,pres+fin,3+sin).
verb_form(rose,rise,past+fin,_).
verb_form(risen,rise,past+part,_).
verb_form(borders,border,pres+fin,3+sin).
verb_form(bordering,border,pres+part,_).
verb_form(contains,contain,pres+fin,3+sin).
verb_form(containing,contain,pres+part,_).
verb_form(drains,drain,pres+fin,3+sin).
verb_form(draining,drain,pres+part,_).
verb_form(exceeds,exceed,pres+fin,3+sin).
verb_form(exceeding,exceed,pres+part,_).

verb_type(have,aux+have).
verb_type(be,aux+be).
verb_type(do,aux+ditrans).
verb_type(rise,main+intrans).
verb_type(border,main+trans).
verb_type(contain,main+trans).
verb_type(drain,main+intrans).
verb_type(exceed,main+trans).
verb_type(flow,main+intrans).

adverb(yesterday).
adverb(tomorrow).
```

B.

High Performance Prolog Implementation

by

Andrew Taylor

A thesis submitted in fulfilment of
the requirements for the degree of
Doctor of Philosophy

Basser Department
of Computer Science

University of Sydney

June, 1991

To

Classes *Aves*, *Osteichthyes* and *Pteridophyta*

ABSTRACT

Many users have found that powerful features such as unification and non-determinism make Prolog the language of choice for their applications. However, existing Prolog implementations for general-purpose machines force the user to sacrifice a great deal of performance compared to imperative languages, such as C and Pascal. There have been attempts to remedy this deficit in performance by the construction of hardware specially designed to execute Prolog and also by implementing Prolog on parallel machines. In this thesis we have taken a different approach. We have used sophisticated compilation technology to produce *Parma*, a high-performance Prolog implementation for a general-purpose architecture, the MIPS RISC architecture.

Parma's key component is a global static analysis phase which examines the program as a whole to gather information for the subsequent compilation phases. The paradigm for this analysis phase is an elegant technique named abstract interpretation. Briefly, this involves symbolic execution of the program over an abstract domain. Recursion is handled by computing fixpoints. An abstract domain which allows very useful information to be gathered in a reasonable time is described. The algorithms employed by the analysis phase are described and various pragmatic issues discussed.

The discussion of the important design decisions taken in the development of Parma starts at the lowest level with a detailed discussion of the choices in data representation. Subsequently the mechanics, intermediate languages and transformations employed by Parma's compilation phases are described.

Measurements of a set of 19 benchmark programs were taken. These demonstrate that Parma's performance is over an order of magnitude better than that of a typical existing Prolog implementation, SICStus Prolog. They also show that Parma's performance is similar to that of two recent pieces of Prolog hardware, the BAM and the KCM.

From a broader perspective, this research provides a reference point from which to evaluate the ability of compilation technology to bridge the widening gap between language and machine.

ACKNOWLEDGEMENTS

Firstly, I would like to thank my thesis supervisor, Norman Foo, for his encouragement and persistence when my work diverged from his interests. I am indebted to Adrian Walker and his group at IBM Yorktown for providing an interesting and challenging environment in 1987 and early 1988 which brought me out of the doldrums. Their subsequent support of my work was also vital. Generous financial support for the last two years of my work was provided by Sigma Data through their postgraduate scholarship scheme. Qantas, by providing free travel, made it possible for me to attend ICLP ’90 and obtain useful review of my work.

During my time as postgraduate I’ve received important but less tangible support from a number of sources. Greg Ryan, Sue Halmagyi and Linda Dawson encouraged my initial thesis work in what was a difficult environment. Although my area of research isolated me in the department, the enthusiasm and ideas of other students were very important to me. Three people I must thank in particular for this are Bruce Ellis, Terry Jones and Bruce Janson. Rex di Bona and James Ashton deserve special mention for including me in their interesting ventures.

My family not only supported me in this work from start to finish but also in all my preceding study. Lastly, and most importantly, without Monica and Boris this thesis would not have been completed.

CONTENTS

LIST OF FIGURES

LIST OF TABLES