ISBN 91-7170-049-8 ISRN SICS/D--91/04--SE

An Automatic Partial Evaluator for Full Prolog

by

Dan Sahlin

A Dissertation submitted
for the Degree of Doctor of Technology
of The Royal Institute of Technology
Department of Telecommunication and Computer Systems
Stockholm, Sweden

March 1991

The Royal Institute of Technology (KTH)
Department of Telecommunication and
Computer Systems
S-100 44 Stockholm
Sweden

SICS
Swedish Institute of Computer Science
Box 1263
S-164 28 Kista
Sweden

TRITA-TCS-9101
ISSN 0284-4397

SICS Dissertation Series 04
ISSN 1101-1335

Abstract

A partial evaluator for Prolog takes a program and a query and returns a program specialized for all instances of that query. The intention is that the generated program will execute more efficiently than the original one for those instances.

This thesis presents “Mixtus”, an automatic partial evaluator for *full* Prolog, i.e. including non-logical features such as cut, assert, var, write etc.

Mixtus does not need any annotations to guide it in its process, but will *automatically* generate a program given only an input program and a query.

A definition of partial evaluation based on the procedural semantics rather than declarative semantics is presented. A program transformed by Mixtus will behave identically as the original one, including side-effects and the order of the solutions returned.

The three basic program transformations “unfolding”, “folding” and “definition” are adapted to full Prolog. The most important transformation is “unfolding” which is used by all partial evaluators to eliminate run-time procedure calls. The “folding” and “definition” transformations generate new recursive predicates specialized for the query.

A loop prevention mechanism specifically tailored for partial evaluation together with a mechanism for “generalized restart”, will guarantee the termination of the partial evaluation for all programs.

Special attention is given to built-in predicates, the cut in particular. To facilitate the execution and removal of cuts during partial evaluation a module for determinacy analysis is used. As negation in Prolog is based on the cut, it will also be treated correctly. Some of the other predicates that are given extra care are `dif/2`, `findall/3` and `bagof/3`.

Many program transformations possible in a logic programming framework will not work correctly for full Prolog. Built-in and generated predicates are therefore classified according to their declarative “purity” so that the “impure” predicates will not hinder the optimization process unnecessarily.

It is a non-trivial task to implement a partial evaluator, so the basic features of the implementation of Mixtus are also presented.

A number of sample programs are partially evaluated, and the effect is discussed.

Descriptors

Logic programming, Prolog, partial evaluation, mixed computation, program transformation, fold/unfold transformation, loop prevention, determinacy analysis, partial deduction.

Acknowledgements

The work presented in this thesis has been carried out at the Logic Programming Systems Laboratory (LPSLAB) of the Swedish Institute of Computer Science (SICS).

One decade ago, my post-graduate studies started at the Department of Computer Systems at the Royal Institute of Technology. I was hired by the head of the department, Lars-Erik Thorelli, and spent five very pleasant and rewarding years at the department, studying, teaching and researching. The newly formed logic programming group I joined was dynamically lead by Seif Haridi, who was to become my thesis advisor. Lars-Erik Thorelli was one of the key initiators in the founding of SICS in 1985, which the members of Seif 's group joined that year. A couple of years later, Seif encouraged me to start working on partial evaluation for Prolog, work which has lead up to this thesis. I am truly grateful for Seif's continuing enthusiastic support, in which he sometimes has believed more in my work than I have done myself.

I would like to thank all my friends and colleagues who have helped me with my thesis work, in particular Torkel Franzén, Feliks Kluźniak, Mats Carlsson, Thomas Sjöland, Bogdan Hausman, Kent Boortz, and Martin Nilsson. Tony Kusalik sent me the source code for the first seven sample programs in chapter 6.

Special thanks goes to Mattias Waldau, Thomas Lindgren, Micha Meier, Steve Prestwich, Roland Karlsson, Hans Nilsson, and Ingvar Olsson for valuable suggestions on how to improve the Mixtus system.

Finally, I wish to thank my parents, Sylvia and Folke, for all the support and encouragement I have received during my 27 years of studying.

SICS is supported by the Swedish National Board for Technical Development (STU), Televerket (Swedish Telecom), Telefonaktiebolaget LM Ericsson, Asea Brown Boveri AB, IBM Svenska AB, Bofors Electronics AB, and FMV (Defence Material Administration).

Table of Contents

1 Introduction

1.1 The title says it all

The words in the title of this thesis taken backwards and explained one by one will highlight the main concepts.

Prolog

The programming language Prolog is based on a few, but very powerful ideas, some from logic and some related to the implementation on a computer. In retrospect it seems inevitable that some day mankind would discover the powerful mechanisms of Prolog. The key inventors, or perhaps discoverers, of Prolog are Alain Colmerauer and his group in Marseille [Colm73] and Robert Kowalski in Edinburgh [Kowa74].

In this thesis it will be assumed that the reader is well acquainted with Prolog. A standard introductory text for programming is [ClMe84]. More advanced text-books are [Ster86b, NiMa90, Ross89].

Full

A distinction is normally made between "pure" Prolog and "full" Prolog. The former contains only the logical features of the language. On the other hand, full Prolog is primarily viewed as a programming language and it also contains non-logical features not found in pure Prolog. Some of these features are essential to making Prolog a practical programming language.

for

From Webster's Unabridged Dictionary:

for, *prep*. [M.E. *for*; AS. *for, fore*, before, for, on account of, through; D. *voor*; Ice. *fyrir*, before; Sw. *för*, before, for; Goth. *faúr*; L. *pro*, before; Gr. *pro*, Sans. *pra*, before, away.] …

4. suitable to; appropriate or adapted to; as, this vase is *for* flowers

Evaluator

An evaluator for full Prolog will take a program and a starting query and will evaluate it fully according to the rules for Prolog. The result will be bindings of the query variables and possibly some side effects such as writing to the terminal.

Partial

A partial evaluator takes a partially specified query and program and evaluates only some parts of the program as not everything is fully specified. The result is a new program, specialized to handle all instances of the original query.

Automatic

An automatic partial evaluator does not need any user annotations at all to direct it in its work, and it will anyway terminate within a finite time.

An

indef. art. sing. An automatic partial evaluator for full Prolog named Mixtus will be presented. The indefinite article "an" is appropriate as there are several possible ways of constructing such a program. Nevertheless, only one, Mixtus, will be presented here which motivates the singular article "an" of the title. Thus Mixtus is a compromise, and it has been designed with the following objectives:

- Mixtus should generate reasonably efficient code. The partial evaluation process should also not take too long or generate too large a program.
- Mixtus should have a relatively simple structure. This may conflict with the previous goal, generating efficient code, but simplicity facilitates the understanding of the basic mechanisms of the system. By trying out the partial evaluator on a large number of examples we have tried to incorporate only the parts that are most important for generating efficient code.

The following two objectives have already been mentioned:

- Mixtus should be able to handle full Prolog. Many built-in predicates such as `var/1` and cut lack declarative semantics, and are treated with extra care. Even for Prolog programs that do not contain any non-logical predicates, the generated program must present all solutions in the same order as the original program. We will return to our definition of partial evaluation which incorporates these conditions. Many optimizations possible for pure Prolog are not possible for full Prolog, but Mixtus tries to identify the pure parts of the program and handle them separately.
- Mixtus should be fully automatic. Most partial evaluators typically need annotations describing when to do "unfolding" of predicates. For an inexperienced user it may be very cumbersome to provide that information. As all annotations are optional and of a general nature, partial evaluation immediately becomes available to all users. With or without annotations, Mixtus is guaranteed to terminate for all input.

Secondary objectives were:

- Mixtus should be idempotent, i.e. if applied twice the generated program is unchanged. There are some cases where this is not true, and this is due to the fact that Mixtus is guaranteed to terminate for all input, and a partial evaluation has been stopped prematurely.
- Mixtus should generate legible output. The generated Prolog program has predicate names closely related to the original predicate names, which makes it easy to compare the two programs.

Version 0.3 of Mixtus which incorporates all features discussed in this thesis has been implemented in SICStus Prolog version 0.7 [Carl88].

1.2 Overview

In chapter 1, a definition of partial evaluation for full Prolog is given. The basic transformations “unfolding”, “folding” and “definition” are presented and their influence on the performance of the generated program is discussed.

In chapter 2, an automatic partial evaluator for pure Prolog is presented. Basic rules for when to unfold and fold are given here which make the partial evaluator “automatic”. Termination is guaranteed by the mechanisms for loop prevention and generalized restart.

In chapter 3, as a prerequisite for the partial evaluator for full Prolog, definitions are given of the three classes of predicates: side-effect predicates, propagation sensitive predicates and logical predicates. A method for determinacy analysis of predicates is also presented.

In chapter 4, our partial evaluator for full Prolog is presented. It is shown how to avoid left propagation of bindings and how to handle the cut and other built-in predicates.

In chapter 5, the main features of the implementation of Mixtus in Prolog are presented. This chapter is fairly independent of the other chapters, and it provides a separate path to understanding Mixtus, as seen from the implementor’s point of view.

In chapter 6, examples are given of partially evaluated programs.

Chapter 7 contains a discussion of the work.

An earlier summary of mainly chapters 1, 2 and 4 has been published as [Sahl90b]. Parts of chapter 3 are to be published in [Sahl91]. The example in section 6.5 comes from [Sahl90a].

1.3 Brief historical background

The term “partial evaluation” appears to have been used first in a paper by Lombardi and Raphael in 1964 [Lomb64], where a simple partial evaluator for LISP was presented.

In 1971 Futamura showed that a partial evaluator applied to an interpreter (for a language X, written in a language Y) given a program will yield a new program (in the language Y) [Futa71].

This is called the first Futamura projection, and is probably the most useful and widely used aspect of partial evaluation, as the overhead of the interpreter might be eliminated. The second Futamura projection shows how to get a compiler, and the third how to get a compiler-compiler. These latter two projections are seldom applied in practice, as it is required that the partial evaluator is an autoprojector, i.e. self-applicable.

One of the first larger more practical partial evaluators REDFUN was constructed by Sandewall's group and is described in [Beck76]. There the third Futamura projection, how to get a compiler-compiler, was also presented, probably for the first time. Haraldsson in his thesis [Hara77] describes the REDFUN-2 system which was capable of handling a large subset of LISP.

Partial evaluation techniques have however always been used in computer science. This is quite natural as the partial evaluation framework covers almost all program optimization techniques.

A.P. Ershov who independently developed the closely related technique of mixed computation [Ersh77] describes his (re-)discovery as follows in [Ersh88b]:

> "The most exciting feeling was a clear vision that mixed computation explains and integrates everything: code generation, compilation and compiler construction, variety of compilation schemes, many ad hoc optimizations: constant propagation, procedure integration, loop unrolling, dead code elimination, loop unloading, not speaking of casing, compile time facilities, etc. Outside of compilation field mixed computation embraces all issues of program specialization and adaptation, deriving ad hoc algorithms from universal ones, modular systems simplification, fighting overheads and redundancy wherever they appear."

Ershov's initial findings were summarized in [Ersh77a] where also the term "mixed computation" was introduced. In a later paper [Ersh80] the state of the art of mixed computation at that time is summarized. Independently of these activities, and apparently earlier, the language REFAL used partial evaluation techniques [Turc79].

After 1976 the number of research activities in this field grew rapidly, especially in the Soviet Union. Most results were for imperative or functional programming languages.

In a functional programming framework partial evaluation is usually derived from Kleene's S_n^m-theorem [Klee52: Theorem XXIII p342]. Successful attempts at creating self-applicable partial evaluators for functional programming languages have been reported by Jones et. al. [Jone89] and Bondorf [Bond90]

These activities will not be reported here, because we are concerned with logic programming. Numerous references can be found in [Sest88].

In 1981, Komorowski in his thesis [Komo81] described the first partial evaluator for pure Prolog. He introduced the now standard method to base the partial evaluator on an interpreter.

For some reason, most of the logic programming community failed to grasp the implications of partial evaluation for Prolog, and the level of activities was very low during the first half of the eighties. In 1982, Kahn presented a partial evaluator for LISP written in Prolog [Kahn82], and two years later he applied a version of that partial evaluator to an interpreter for Prolog, thereby compiling Prolog to LISP [Kahn84].

The first partial evaluator for full Prolog was presented in 1984 by Venken [Venk84]. Many of the techniques introduced in his paper are now standard methods, also used by Mixtus: the partial evaluator is a continuation-based meta-interpreter written in Prolog and a classification of built-in predicates is made according to whether they have a side-effect or not. The system is also automatic in the sense that no user annotations are needed.

A source of inspiration particularly relevant to our work was an article published in 1986 by Takeuchi and Furukawa [Take86]. Their short partial evaluator written in Prolog was shown to be capable of eliminating the overhead when meta-programming techniques were used.

An important milestone in the history of partial evaluation/mixed computation was the workshop arranged in Gammel Avernæs, Denmark in 1987 . In the following we will make frequent references to proceedings of this workshop [Bjør88, Fuch88]. Our historical note ends here, just when activities in partial evaluation of Prolog were expanding. Subsequent relevant work will be mentioned in the following as our work is related to it and is summarized in section 7.2.

1.4 Partial evaluation vs. partial deduction

Lloyd and Shepherdson in [Lloy87] describe partial evaluation for logic programming as follows: “Given a program *P* and a goal *G*, partial evaluation produces a new program *P'*, which is *P* ‘specialized’ to the goal *G*. The intention is that *G* should have the same (correct and computed) answers wrt *P* and *P'*, and that *G* should run more efficiently for *P'* than for *P*.”

Definitions of and the conditions for soundness and completeness of partial evaluation for the declarative (the answer is a logical consequence) and “procedural” (based on SLDNF) semantics are given in that paper. Komorowski has recently [Komo89] named this view on partial evaluation of logic programming “partial deduction”, which seems quite appropriate.

A Prolog program may however contain many non-logical features and features relating to the actual execution such as side-effects. This has little to do with deductions, and the term “partial evaluation” seems more appropriate for a system that is capable of handling full Prolog.

For pure Prolog programs (i.e. without built-in predicates), the Prolog execution mechanism can be viewed as an incomplete but sound theorem prover. This means that if the query Q to a program P returns an answer substitution θ, then Qθ is a logical consequence of P. A partial evaluator for full Prolog, on the other hand, must take into account the actual execution of the program rather than the simpler concept of logical consequence.

1.5 Some uses of partial evaluation

Traditionally a text on partial evaluation starts with an explanation of how partial evaluation can be used to generate “compilers” and “compiler-compilers”. For excellent descriptions of how this is done in general the reader is referred to [Futa82, Fuji88b].

The emphasis of Mixtus is to make it a *practical* tool for programming in Prolog. Nobody expects to have to guide a Prolog compiler with annotations to make it possible to compile a Prolog program. Likewise, to be of practical use, the partial evaluator has to be *automatic*, and not need any help at all to do a reasonable job. As in a compiler, some general optional switches might be desirable to set the level of thoroughness; a longer transformation time may yield a better result. Of course, neither the compiler nor the partial evaluator is allowed to loop indefinitely for any program. Some more specific (but optional) switches might also help the system produce a better result.

Program structures recommended for methodological reasons are often associated with a degradation of execution performance. For instance, calling a general procedure instead of performing the actions directly in-line generally results in slower execution. The traditional solution in imperative programming languages has been to introduce “macros”[1] which are expanded before the program is compiled. The main disadvantage of this solution is the complex semantics. For example, if the simple C function “square” is converted to a macro by “#define square(x) x*x”, then the call “square(i+1)” is expanded to “i+1*i+1” and the call “square(i++)” is expanded to “i++*i++” which is probably not at all what was intended. A step in the right direction, retaining the semantics, is the possibility to declare certain procedures in-line, as in C++ [Stro86]. However, a compiler or macro expander is usually not expected to unfold loops or recursive procedure calls. This is in contrast with partial evaluation, which opens new possibilities of writing structured code without sacrificing performance.

Although the main reason for doing partial evaluation with Mixtus is to gain execution efficiency, this is not always the case for other partial evaluators. The program “Distill” is a partial evaluator for the PostScript printer language [Reid89]. Although the generated program does run faster than the original one, it is equally important that it is smaller, as this makes it much more economical in storage and distribution. Another advantage is that proprietary libraries (for instance ProcSets for the Macintosh) in the original program have totally disappeared by unfolding, which should make it legal to distribute the generated program to anyone.

An interesting use of partial evaluation has been reported in [AlKa90] where Mixtus was successfully applied to a Prolog program that was executed in or-parallel. On a single processor the program ran about five times faster after all meta-calls in the program were eliminated by partial evaluation. But the most interesting fact was that the speed-up figures when using multi-

[1] Macros for Prolog procedure calls and terms have been proposed by [Kond88].

processors were also improved as the meta-calls use global memory. Without partial evaluation the slower global memory had to be used, and the speed-up on a 6 processor machine was only 3.86. With partial evaluation the reported speed-up was 5.10.

Comparatively few published texts seem to exist which use partial evaluation to improve the performance of scientific applications. The examples known to us are [Moge86, BeWe90]. None of these were however done in Prolog.

The most common and promising use of partial evaluation is to remove the overhead of interpretation. Some examples will be given in this thesis on such usage. Successful application of partial evaluation on interpreters has been reported frequently [Lakh89a, Lakh89b, Ster86a, Take86, LeSa88].

1.6 Partial evaluation and mixed computation for full Prolog

The standard definition of partial deduction of Prolog by Lloyd and Shepherdson [Lloy87] is purely logical, and cannot cope with execution of Prolog. For instance, there is no notion of "first solution" in a logical framework. It is therefore necessary to introduce a new definition of partial evaluation for full Prolog.

Partial evaluation and mixed computation [Ersh88a] are closely related, as seen from the definitions below, which are modified for full Prolog.

Partial evaluation for full Prolog

Let P be a Prolog program, i.e. a list of clauses.

Let Q be a query to a Prolog program.

Let P[Q] be the results of running the program P with the query Q. By results, we here mean the sequence of side-effects caused by running the program, such as writing to the terminal or to files and producing printouts of the bindings of the variables in Q. It is also assumed that an exhaustive search is made so that the program is forced to produce all solutions. The program may then either terminate, loop or produce infinitely many solutions.

A *partial evaluator* for full Prolog is a function $\Pi_{partial}$ taking a program P and a query Q and returning a new program P'. $\Pi_{partial}$ has the following properties:
For all instances $Q\theta$ of Q:

① P'[$Q\theta$] produces the same results as P[$Q\theta$] and in the same order.

② The computation of $\Pi_{partial}$(P,Q) will always terminate for all P and Q.

(This definition will be slightly modified in the following pages.)

Mixed computation for full Prolog

Most versions of the partial evaluator presented here will use a definition more resembling mixed computation.

Let P, Q and P[Q] be defined as above. A mixed computator for full Prolog is a function Π_{mixed} taking a program P and a query Q and returning a pair $\langle P',Q' \rangle$ consisting of a new program P' and a new literal Q' having the following properties:

- The arity of Q' is the number of distinct variables in Q, and all arguments of Q' are distinct variables.
- Q and Q' must have different functor names.

For example, if Q=`q(f(X),3,g(a,X,Y))` we may then have Q'=`qx(X,Y)`. The process of generating Q' given Q is called *reparametrization* [SmHi90].

The remaining conditions on P' are the same as for partial evaluation but with P'[Q]θ replaced by P'[Q'θ].

Relating partial evaluation and mixed computation

Any partial evaluator can be considered a mixed computator by letting

$$\Pi_{mixed}(P,Q) = \langle \Pi_{partial}(P,Q) + \{Q' :- Q.\}, Q' \rangle$$

where Q' is related to Q as described above, and Q' is not defined in $\Pi_{partial}(P,Q)$. The notation A+B denotes the concatenation A and B.

Conversely, a partial evaluator may easily be obtained from a mixed computator by extending the returned program by the clause "Q :- Q'.". That is,

$$\Pi_{partial}(P,Q) = P' + \{Q :- Q'.\} \quad \text{where} \quad \langle P',Q' \rangle = \Pi_{mixed}(P,Q).$$

provided the predicate Q is not defined in P'. If that is the case, the predicate Q and all calls to Q in P' are renamed so that there is no interference between the old definition of Q and the new one (Q :- Q').

As will be shown in the following, this method of converting mixed computation into partial evaluation is used by Mixtus.

Input to Prolog

In the definitions above input to Prolog (i.e. predicates such as `read/1` and `get/1`) has been totally ignored, and in this section it will be explained why we will continue to do so.

For a real Prolog program the result depends on the input. Let the sequence of input values be denoted by *I*. Then the functions P[] and P'[] will have two arguments: the query Q and the sequence of input values *I*. The first property of a partial evaluator then becomes:

① For all input sequences *I*, P'[Qθ,*I*] produces the same results as P[Qθ,*I*] and in the same order.

The input sequence *I*, will not be known at all to the partial evaluator. This means that a call `read(X)` in the program can bind `X` to any value. As will be shown later, this is the default assumption for variables occurring in built-in predicates, so no special precautions will be necessary to handle input predicates. For all practical purposes the complications due to input can be ignored in the partial evaluator. Thus, in the following, the input sequence *I* will not be spelled out in the definitions.

Partial evaluation of a partial program

The program P given to the partial evaluator may contain goals that refer to predicates which do not occur in P at all. The partial evaluator must leave these goals as they are and assume the worst, i.e. that calling any of these goals may cause arbitrary side-effects and instantiations.

The capability of the partial evaluator of optimizing just a partial program increases its usefulness when giving partial data to the system. Instead of just having the possibility of giving partial data (=Q) to the top-level query, the user can also exclude a part of the program from partial evaluation. A program P can be split up into two parts: P_1 and P_2. Part P_1 can be partially evaluated into P'_1, and the new combined program $P' + P_2$ will behave identically to P_1+P_2 for all instances of Q. This method is attractive if one part is frequently changed and replaced (the P_2 part) and the rest of the program is constant.

This capability of leaving out a part of a program was one of the basic mechanisms in the partial evaluator of Komorowski [Komo81].

A corollary for logical programs

The above definition of partial evaluation implies the following corollary about logical programs.

COROLLARY

Let P be a pure logic program, i.e. one without non-logical built-in predicates.

Let $P' = \Pi_{partial}(P,Q)$.

If P'[Q] returns an answer substitution θ, then $P \models Q\theta$.

This follows immediately from the fact that P[Q] and P'[Q] return exactly the same answer substitutions (due to the definition of partial evaluation) and the fact that if P[Q] returns an answer substitution θ, then $P \models Q\theta$ (which is a property of pure Prolog).

A logical optimization: some loops are removed

If we are allowed to remove non-productive potential loops, composite goals like G,`false` may be replaced by `false` if “G” does not have any side-effects. This optimization may be very significant if “G” starts a complicated computation. It does not seem feasible to restrict this optimization to the cases where “G” is guaranteed to terminate, as this is a very complex property

to deduce, and in general not even decidable. However, it is very straightforward to check if the call may cause any side-effects.

The above definition of partial evaluation does not allow this optimization, since by sometimes avoiding infinite loops the resulting program could produce more solutions than the original one. Condition ① of that definition is therefore relaxed into the following.
For all instances Qθ of Q:

① P'[Qθ] produces all the results produced by P[Qθ] and in the same order. P'[Qθ] may then produce more results if the execution of P[Qθ] does not terminate.

It may produce more results only if P[Qθ] starts looping after producing a certain result. Alternatively, it may loop itself when P[Qθ] loops. It will always produce all results, including duplicates, in the same order as P[Qθ].

As an abbreviation of the above condition we write $P \leq_Q P'$. This partial ordering of programs is extended to goals as follows. For two goals, G and G', the relation $G \leq G'$ holds iff the relation $P \leq_Q P'$ holds for all queries Q and programs P and P' where P' is identical with P except that an occurrence of G has been replaced by G'. This definition will be used when classifying predicates (chapter 3).

Unfortunately the above relaxed definition allows the new program P' to produce arbitrary results. But it is hard to conceive that a programmer using Prolog as a procedural language would write a program where a non-productive infinite loop is an essential part of the program. Thus, all normal programs written with the procedural behavior in mind will not contain such infinite loops, and all those programs will behave identically after partial evaluation.

There is however one class of Prolog programs that may naturally get into infinite non-productive loops. A programmer who views Prolog as a theorem prover, using the declarative semantics and disregarding the execution procedure, may easily get into infinite non-productive loops. Also, with the new definition of partial evaluation the above essential corollary does no longer follow, since nothing is said about the extra solutions produced. To constrain P' to produce only logical consequences, the contents of the corollary is added as a third condition in the definition:

③ For a pure logic program P, if P'[Q] returns an answer substitution θ, then $P \vdash Q\theta$.

This condition holds for Mixtus, since the only extra transformations allowed are:

- (G,`false`) is replaced by `false`
- (G,t_1=t_2) is replaced by (t_1=t_2,G)

both of which are perfectly logical.

If, for some reason, this optimization is not desirable, it is possible to set the mode of the system (using the flag *preserve_loops*) so that the original definition of partial evaluation is used,

where the result produced by the new program is always identical to the result produced by the original program.

Jones et. al. [Jones89] seem to take a similar approach to removing loops when partially evaluating a functional programming language.

A pragmatic optimization: some errors are removed

To further facilitate partial evaluation, it will be assumed that the original program P will not produce any error conditions when built-in predicates are used. For example, it might be advantageous to transform `X<3,X=2` in the original program into `X=2` as the difference in behavior will only be visible when `X` is unbound or bound to anything but an arithmetic expression, thereby producing a run-time error.

Excluded built-in predicates

In most Prolog implementations there are a few rarely used predicates that are very closely related to the actual implementation. For example, the predicates `ancestors/1` and `sub-goal_of/1` inspect the goal-stack. If such predicates occur in a program to be partially evaluated, the program produced may behave differently than the original one. These issues are discussed in more detail in section 4.2.6.

Other important properties not covered by the definitions

As seen from the above definition, one correct partial evaluator is the identity function which simply returns the input program P, ignoring the query Q! The main reason for making the partial evaluation is to make the program run faster. Unfortunately, the efficiency of a program is difficult to measure without executing the program. Thus any specific claims of speed-up are not part of the definition of partial evaluation above. Instead, the transformations made by the partial evaluator are carefully chosen so that in most reasonable implementations they will normally make the generated program run faster, or at least not slower. Where various implementations may differ in a significant way, the SICStus Prolog implementation [Carl88] based on the Warren Abstract Machine (WAM) [Warr83] is chosen as the standard.

It is not very difficult to implement a partial evaluator that is guaranteed to terminate for all input programs and queries. The user may however expect more than that; he may ask for a predictable maximum execution time. This has not been implemented and it is difficult to predict the time a partial evaluation will take.

1.7 Basic transformations

The three basic transformations used in this partial evaluator are *unfolding*, *folding* and *definition*.

1.7.1 Transformation 1: Unfolding

1.7.1.1 Logical unfolding

Let the goal A be defined by the clauses $\{(A_i :- B_i) | i \in I\}$. The unfolding of A in the clause "F :- Before,A,After" in logic programming is the following transformation[2]:

F :- Before, A, After. *where* A *is defined by* $\{(A_i :- B_i) | i \in I\}$

$\Downarrow$ *unfolding*

$\{(F :- Before, B_i, After)\theta \mid i \in I \wedge mgu(A,A_i) = \theta \wedge \theta \neq false\}$.

Logical Unfolding

For example, consider the following definitions of `p/1`, `q/1` and `r/1`.

```
p(X):-q(X), r(X).
q(a).
q(A):-q(A).
r(b).
r(a).
```

The result of unfolding `r(X)` in the clause defining `p/1` is

```
p(b):-q(b).
p(a):-q(a).
```

Although the clauses produced can be shown to have the same declarative semantics, the Prolog semantics is quite different. If the query `p(X)` is given to the original clauses, the result `X=a` will be returned repeatedly, whereas the query given to the transformed clauses will loop immediately, never producing any solutions. Other examples can easily be constructed where the order of the solutions is changed by logical unfolding.

So, the unfolding transformation used in logic programming is not applicable even to pure Prolog. This should not come as a surprise as the declarative semantics does not at all deal with in which order solutions are produced. The fact that the correctness of these transformations for a logic programming framework has been proven by Tamaki and Sato [Tama84] is therefore unfortunately of little use for us.

1.7.1.2 Unfolding for Prolog

What is needed for Prolog is an unfolding transformation that does not reverse the order of the solutions. This can be accomplished by ensuring that all choice-points are created in the same

[2] mgu(A,B) is the function that returns the most general unifier of the two terms A and B. If no such unifier exists, we use the convention that the special value "false" is returned instead.

order, before and after the transformation. The following unfolding transformation will guarantee this, as a disjunction is explicitly incorporated into the clause, keeping the choice-point of the unfolded call in the right position.

F :- Before, A, After. *where* A *is defined by* $\{(A_i :- B_i) \mid i \in I\}$

$\Downarrow$ *unfolding*

F :- Before, $\bigvee_{i \in I}(A=A_i, B_i)$, After.

Unfolding for Prolog if A *is not the first goal*

F:- A, After. *where* A *is defined by* $\{(A_i :- B_i) \mid i \in I\}$

$\Downarrow$ *unfolding*

$\{(F :- B_i, After)\theta \mid i \in I \wedge mgu(A,A_i)=\theta \wedge \theta \neq false\}$

Unfolding for Prolog if A *is the first goal*

The unfolding rule if A is the first goal can be seen as an optimization, where the following transformation rules for disjunctions and equalities have been used.

$F :- \bigvee_{i \in I} G_i$, After. $\Rightarrow$ $\{(F :- G_i, After) \mid i \in I\}$

Unfolding a disjunction

$F :- t_1=t_2$, After. $\Rightarrow$ $\begin{cases}(F :- After)\theta & \text{if } \theta=mgu(t_1,t_2) \wedge \theta \neq false \\ \{\} & \textit{otherwise no clause}\end{cases}$

Unfolding an equality

Returning to the example above, unfolding `r(X)` now instead produces

```
p(X):-q(X), (r(X)=r(b); r(X)=r(a)).
```

Simplifying the unifications as elaborated below we get

```
p(X):-q(X), (X=b; X=a).
```

If the disjunction formed by the unfolding operation is empty, it is replaced by the goal `false`.

The logical unfolding transformation is correct even for Prolog if Before or A does not leave any choice-point, as will be further commented on below.

Simplifying unifications without left-propagation

The unifications "$A=A_i$" in the disjunction above should always be simplified as much as possible without any harmful left-propagation of bindings. This means that a variable in A may be unified with a variable in A_i, if that variable is not already unified with another variable in A. The remaining unifications must be kept explicit.

For example, if "$A=A_i$" is `p(A,B,C)=p(X,X,s(Y))`, then only `A` may be unified with `X`, and `B=A, C=s(Y)` are kept as explicit unifications.

Avoiding recomputations

Unfolding for Prolog also avoids some recomputations, as shown by the following example. Let the following program be partially evaluated with respect to `h(X)`.

```
h(X):- p(X),q(X),r(X).
q(1).   q(2).
```

Logical unfolding will duplicate the call to `p/1`:

```
h(1):- p(1),r(1).
h(2):- p(2),r(2).
```

If an exhaustive search is made `p/1` may be started twice with a possibly complex computation. Unfolding for Prolog does not have this problem, since there still is only one call to `p/1`.

```
h2(X):- p(X),(X=1; X=2), r(X).
```

There are however cases where the logical unfolding can be used for Prolog. If the goals before the goal to be unfolded are deterministic (i.e. do not leave any choice-points) and logical, the order of the solutions will not be changed by letting bindings from the disjunct propagate left. As shown in the example above, the call to `p/1` gets duplicated in that case. Normally this is a disadvantage, but it might be advantageous. Say that `p(1)` and `p(2)` fail almost immediately, whereas `p(X)` starts a long computation. Then it is probably better to propagate the bindings. We have chosen not to propagate the bindings of the disjunct as we cannot determine in general whether it is an optimization.

Venken introduced the technique of creating a disjunction when unfolding for full Prolog [Venk84]. His main motivation was to avoid left propagation of bindings, and not to preserve the order of solutions.

1.7.2 Transformation 2: Definition

Given an arbitrary goal, a new predicate defined by a single clause can be generated. The distinct variables of the goal form the parameters of the new clause head, and the new predicate gets a new unique name. The body of the clause is the old goal.

For example, let the goal be `p(a,X,Y,X)`. Then the new clause defining the new predicate could be `p1(X,Y):-p(a,X,Y,X)`.

1.7.3 Transformation 3: Folding

This is the process by which a goal that is an instance of the body of a clause is replaced by the corresponding instance of the head of the clause. This is a correct transformation if the predicate is defined by a single clause and if all variables in the body of that clause occur also in the head, in which case the head and the body are "equivalent". More formally, assume there is a clause produced by "definition"

Anew' :- A'.

then the goal A which is an instance of A' (i.e. A=A'θ) may be folded in the clause

Head :- Before, A, After.

This will produce the clause

Head :- Before, Anew'θ, After.

The conditions for performing folding will be somewhat relaxed in section 2.4.2.

Burstall and Darlington [BuDa77] pointed out that the folding transformation, or something equivalent, is necessary for constructing new recursive predicates. All partial evaluators for Prolog described in the literature use unfolding as their basic mechanism, and some also have folding and definition. Sometimes folding plus definition is called "rewrite" [Lakh90]. The main problem in a partial evaluator is to know when to apply the various transformations. The heuristics used by Mixtus are shown in section 2.2.5.

The folding and definition transformations above are not as general as they could be, as folding is only performed on a simple, non-composite goal. This is a design decision made for Mixtus which simplifies it and facilitates making it automatic. Thus, some program transformations which use folding for composite goals are not possible [Tama84, Lakh88]. Our experience indicates that the most important class of programs to be partially evaluated, the interpreters, folding for composite goals does not seem to be required for getting satisfactory results.

1.8 A small example of partial evaluation

The following example will illustrate all three basic transformations mentioned above.

Program P:

```
p([],F,A,X) :- X is F-A.
p([H|R],F,A,X) :- F1 is H*F, p(R,F1,A,X).
```

The predicate call `p(L,F,A,X)` will multiply `F` by the product of all numbers in the list `L`, then subtract `A` and finally return the result in `X`.

Query Q:

```
p(U,V,5,W).
```

The partial evaluation starts by generating a new query Q':

```
p2(U,V,W).
```

where `p2/3` is defined by the single clause

```
p2(U,V,W) :- p(U,V,5,W).
```

Unfolding the goal in the body of that clause will produce two clauses

```
p2([],V,W) :- W is V-5.
p2([H|R],V,W) :- F1 is H*V, p(R,V,5,W).
```

Finally, `p(R,V,5,W)` is folded into `p2(R,V,W)` which will make the resulting program P':

```
p2([],V,W) :- W is V-5.
p2([H|R],V,W) :- F1 is H*V, p2(R,V,W).
```

Notice that the constant argument 5 is no longer carried around through the recursion, but only appears where it is used, i.e. in the subtraction.

Finally the relation between `p/4` and `p2/3` is established, as mentioned in section 1.6, by adding the clause

```
p(U,V,5,W) :- p2(U,V,W).
```

which replaces the original clauses for `p/4`.

1.9 Sources of performance improvements

The most common reason for doing partial evaluation is to get a program that runs faster than the original one. The main sources of improved performance in the partial evaluator are the following:

Generation of new, possibly recursive, predicates. The partial evaluator performs an analysis of how the various predicates are called, and for each instantiation pattern of a predicate, a new predicate specialized for that call pattern is generated. All arguments that are instantiated at partial evaluation time are removed, and only the uninstantiated variables remain.

Unfolding of user-defined predicates. This will remove a predicate call. This transformation is particularly favorable if there are no matching clauses, and the call can be eliminated altogether.

In the conjunction (A,B), if B is found to fail, then the whole conjunct may be replaced by false, and no call is made at all to A.

In real Prolog programs with side-effects not all of these optimizations are possible, and this will be elaborated in detail in chapter 4.

Evaluating built-in predicates. All built-in predicates that can be evaluated should of course be evaluated at partial evaluation time, and this is also an important source of efficiency improvement. Usually, when a built-in predicate cannot be evaluated, it is because it is insufficiently instantiated. For example, `var(X)` cannot be evaluated, as we do not yet know if `X` will be bound to a non-variable at run-time. Some built-in predicates cannot be evaluated as they have side-effects that must be delayed until actual execution time, such as `write(17)` or `read(X)`.

To facilitate unfolding and evaluation, the handling of variable bindings is very important. In the partial evaluator we distinguish between right-propagation and left-propagation of bindings.

Right-propagation of bindings. This takes place for instance when "`X=`*t*,e[`X`]" (where *t* is an arbitrary term and e[`X`] is an expression containing the variable `X`) is transformed into e[*t*]. Without right-propagation not much can be done by a partial evaluator.

There are however some disadvantages of right-propagation with respect to execution efficiency. If the variable occurs several times in the same predicate call or in conjunctive predicate calls, there is a danger of creating several copies instead of just one of the structure. If the structure is large, this may be a major disadvantage. At present, most Prolog compilers will not generate code that avoids creating multiple copies of duplicate structures in the same clause. Fortunately, in most cases the structure is either very small or partial evaluation will totally "consume" and eliminate the structures.

For a disjunction, there is a similar but harder problem of right-propagation. During partial evaluation "`X=`*t*,(e_1[`X`]; e_2[`X`])" becomes transformed into "(e_1[*t*]; e_2[*t*])". Thus there is a danger that backtracking will cause the same structure *t* to be created again, which was not the case in the original code. It is not clear that it is always better to move the creation of *t* back to right before the disjunction. It may be the case that backtracking will not occur or that a branch fails, so that the structure *t* is just created once or not at all. As the choice is unclear the present partial evaluator does not try to move the bindings out of the disjunction, with the possibility of generating sub-optimal code.

Left-propagation of bindings. In contrast to right-propagation, for full Prolog extreme care has to be taken when propagating bindings left. A non-logical predicate such a "var" makes

`var(X),X=3` quite a different thing from `var(3)`. For purely logical predicates it is quite safe to move the bindings left. Having more instantiated predicate calls makes it easier for the partial evaluator to do unfolding, so the newly instantiated predicates are examined once again. This may in turn give rise to new left-propagated instantiations, so care has to be taken to prevent the partial evaluator from looping.

As in right-propagation, there is a problem concerning duplication of terms in left-propagation. For example, it might be disadvantageous to transform `p(X,X),X=`bigterm into "`p(`bigterm,bigterm`)`". On the other hand, as `p(`bigterm,bigterm`)` is reexamined and partially evaluated again, it might be advantageous. Mixtus always performs left-propagation if it is allowed, thereby being potentially sub-optimal.

1.10 Implementation-dependent problems

The efficiency improvement of some program transformations is highly dependent on the actual Prolog implementation. There are two major methods to implement Prolog: structure sharing [Warr77] and structure copying (which actually includes some structure sharing techniques) [Warr83]. Most systems, among them SICStus Prolog, now employ the latter method, and this method will be assumed to be used in this partial evaluator. Even for the same system, a certain program transformation may be disadvantageous for interpreted code but advantageous for compiled code. Although the ultimate goal of this partial evaluator is to improve execution performance, due to these implementation differences we have made the radical decision not to introduce any formalism for estimating the efficiency of a Prolog program.

Thus no claim is made that Mixtus always improves the program given. In fact, for some programs, as has already been mentioned, execution performance deteriorates. These cases are however rare, and for a wide range of programs as shown in chapter 6, the programs are substantially improved.

2 Partial evaluation of pure Prolog

A partial evaluator for pure Prolog, i.e. without non-logical features, will be presented in this chapter. Since it is based on the basic transformations, as described in the previous chapter, the order and multiplicity of the solutions will be preserved in the program produced by partial evaluation. The algorithms presented in this chapter are the core of Mixtus, and will remain almost unchanged for the full system.

A distinctive feature of Mixtus is the automatic choice of program transformations using general loop prevention algorithms which guarantee the termination of the process.

2.1 Generating residues

The standard technique for implementing a partial evaluator for Prolog is to write it as a specialized meta-interpreter in Prolog [Venk84]. According to the definition of a partial evaluator it takes a program and a partially instantiated query and returns a program specialized for all instances of that query. Thus, the meta-interpreter is given the original query and simplifies it, consistently obeying the following principle:

In the partial evaluator a term t represents all instances tθ of t.

Let us illustrate this principle by partially evaluating the built-in predicate `length/2`. For instance, `length([A,B],X)` can safely be evaluated to `X=2`. On the other hand, `length(L,X)` cannot be completely evaluated and is left as it is. Those remaining goals that are not evaluated are called the *residue,* and are used in the construction of the generated program output by the partial evaluator.

Two descriptions will be given of the basic algorithm of Mixtus. The first is more informal and implementation-oriented. The second is more formal and justifies in detail the transformations used.

2.2 Informal description of the basic procedure

2.2.1 The top-level query

Mixtus only accepts a top-level query that is a call to a user-defined predicate Q. Internally Mixtus more resembles mixed computation, where a new query Q' is returned together with the new program. A reparametrization therefore is first performed on Q, producing Q'. The mixed computation then continues with the clause Q':-Q producing a new program, including a new definition of Q'. As described in section 1.6, the clause Q:-Q' is finally added to the program,

provided the old predicate Q is no longer used in the new program. In section 2.7 it will be shown how it is guaranteed that the old definition of Q is no longer needed.

2.2.2 Handling clauses

The first clause to be partially evaluated is Q':-Q, as described above. This is done using the procedure $\Pi_{clauses}$, which takes as input a sequence of clauses defining a predicate, and returns a new sequence of clauses which is the partial evaluation of the predicate. As a side-effect during partial evaluation, $\Pi_{clauses}$ may possibly create clauses for other predicates, as explained below.

The result of partially evaluating this sequence of clauses is the concatenation of the partial evaluations of each single clause.

Each clause is partially evaluated by partially evaluating each goal in the body of the clause, one by one. There are two kinds of goals: calls to built-in predicates and calls to user-defined predicates.

2.2.3 Handling calls to built-in predicates

A large and important part of any partial evaluator for Prolog is the module which takes care of partially evaluating all built-in predicates. The simplest idea is having a database of the conditions under which each built-in predicate can be safely evaluated. A predicate is only considered evaluable if a finite set of substitutions is produced.

```
evaluable(true).
evaluable(X=Y).
evaluable(length(L,X)) :- \+ open_tail(L); integer(X).
open_tail(L) :- var(L).
open_tail(L) :- nonvar(L), L=[H|T], open_tail(T).
```

That is, calls to `true/1` and `=/2` can always be evaluated, whereas `length/2` has some conditions guaranteeing that a finite set of solutions will be returned.

We can here view an evaluation of built-in predicates as follows.

Let G be a call to a built-in predicate in the clause “F:−Before,G,After”. If G is not evaluable in the sense above, the clause is returned unchanged. If G is evaluable, an execution of G will produce a finite set of substitutions θ_i and the clause is replaced by a new clause

$$F{:-}Before, \bigvee_i G{=}G\theta_i, After.$$

If the Before-part is empty, the disjunction comes first in the clause and a transformation into a list of clauses performed where the unification $G=G\theta_i$ has been performed in every clause.

$$\{(F :- After.)\theta_i\}_i$$

An example: The clause `p(X,Y):-length([A,B,C],X),r(X)` when `length/3` is evaluated, becomes `p(3,Y):-r(3)`.

When partial evaluation for full Prolog is described, a more general mechanism for handling built-in predicates will be shown. Many predicates get special treatment, e.g. `is/2`, where all ground and thus evaluable subexpressions are evaluated even if the whole expression cannot be evaluated. For instance, `X is Y+2*3` is transformed into `X is Y+6`. These methods will be elaborated in more detail in section 4.2.

2.2.4 The goal-stack

Mixtus maintains a stack containing the goals it encounters, much as in an ordinary Prolog interpreter. If a user-defined goal A is encountered, the clause (Anew':-A') is pushed on the stack.

The body of the clause, A', is a variable variant of A, where all variables in A' are new variables not occurring anywhere else. This is necessary so that further instantiations of variables in A will not change any variables.

The head of the clause, Anew', is created by extracting all variables from A' and creating a functor which has a name different from all predicates. This process is called reparametrization [SmHi90]. For readability in Mixtus the name is formed by concatenation of the names of the first functors, possibly followed by a digit to make the name unique.

For example, if A' =`pred(term(X,Y,3),Y)` we then could have Anew= `predterm(X,Y)`.

Only the parts of A' that can vary, i.e. the variables, are kept in Anew'. As will be shown below, due to folding an instance of A' may sometimes be replaced by the corresponding instance of Anew', thereby eliminating known constants from the arguments of the call.

2.2.5 The basic mechanism of Mixtus

The basic mechanism of Mixtus will decide what to do when a call to a user-defined predicate is encountered. The predicate call may either be unfolded, folded, kept as it is or replaced by a call to a newly generated predicate. In other partial evaluators this choice is normally guided by user annotations. Our goal is to have the choice made automatically, and how that is achieved is shown below. This procedure is perhaps the most distinctive feature of Mixtus.

The loop prevention mechanism, which will be elaborated in more detail later, will guarantee the termination of the procedure.

Folding	**If** A is an instance of a goal A' in a clause in the goal-stack **then** replace A by the corresponding instance of Anew'. **else**
Loop prevention	**If** by inspecting the goal-stack the loop prevention mechanism indicates a possible infinite loop **then** the goal A is kept as it is **else**
Recursive invocation of partial evaluator	Let the clauses defining A be "H_i:–B_i". Let the substitution resulting from unifying A with a head H_i be θ_i. $\Pi_{clauses}$ is called with the clauses "(Anew :- B_i)θ_i" as input and the goal-stack extended with (Anew':-A').
Definition	**If** the definition of Anew in the returned clauses is directly or indirectly recursive **then** no unfolding is done and the goal A is replaced by Anew. The clauses returned by $\Pi_{clauses}$ define Anew, and they are added to the resulting program.
Unfolding	**else** unfolding can be done, using Anew and the newly generated clauses.

The main procedure for partial evaluation of user-defined goals

As can be seen from the above procedure, the tests for folding and loop prevention are made before the recursive invocation of the partial evaluator. This will eliminate the danger of going into an infinite recursion since the loop prevention will prevent infinite recursion. The test for folding is done before the test for loops as it is better if a call is made to Anew, whose clauses are the result of a partial evaluation, than to A, which simply uses the original clauses.

A new predicate is generated if Anew after recursive invocation of the partial evaluator is found to be directly or indirectly recursive. A quick test to determine if Anew is recursive is described in section 5.4.

Some examples using the above algorithm will be given later in this text.

2.3 Formal description of the basic procedure

In this section the basic procedure of Mixtus is formally described, and its correctness is shown based on the correctness of the basic transformations *unfolding, folding* and *definition*.

Some words about the notation. Both predicate definitions and programs are lists of clauses, sometimes denoted by a list generator $\{Clause_i\}_i$ or $\{Clause_i \mid conditions\}$. A single clause is sometimes considered a singleton list of clauses.

For simplicity and clarity, the concept of *p-program* is introduced.

2.3.1 P-program

DEFINITION OF p-program

A p-program (pair-program) S is a pair of lists of clauses ‹Cls, P›. All clauses in the first component have the same head functor.

P-programs are very convenient as they enable us to focus our attention to just a few clauses (Cls) which are to be transformed.

DEFINITION OF the result of a query to a p-program

Consider a query Q to the p-program ‹Cls, P›. If the clauses in Cls do not have the same head functor as the query Q, then the results of the query Q to the p-program ‹Cls, P› are the same as the query Q to the program P.

Otherwise, the results are the same as the query Q' to the program Cls'+P.

Q' is here just a simple renaming of the head functor to a new unique functor (and not a reparametrization). Cls' is obtained by renaming all head functors in Cls to the functor of Q'.

For example, the results of the query `q(A,B)` put to the p-program

```
‹q(2,X):-q(X,3)., q(1,3).›
```

are the same as the results of the query `q'(A,B)` put to the Prolog program

```
q'(2,X):-q(X,3).
q(1,3).
```

That is, the execution will return `A=2,B=1` and nothing else.

DEFINITION OF THE RELATIONS $\Rightarrow$ AND $\Leftrightarrow$

Let each of S_1 and S_2 be a program or a p-program. The relation $S_1 \Rightarrow S_2$ holds if all predicates defined in S_1 are also defined in S_2 and all queries using these predicates return the same results and in the same order for both programs.

As we deal only with pure Prolog, the only results of an execution are bindings of the query variables.

$S_1 \Leftrightarrow S_2$ holds iff $S_1 \Rightarrow S_2$ and $S_1 \Rightarrow S_2$.

2.3.2 The basic transformations for p-programs

The three basic transformations, *definition*, *unfolding* and *folding* are here defined for p-programs and their correctness conditions stated. As can be seen below, all transformations are closely related to the similar transformations of a Prolog program and are assumed correct in the following. Note that only *definition* extends the program so that the relation $\Rightarrow$ must be used.

The definition transformation

$\langle$Clause, P$\rangle \Rightarrow \langle$Clause, P+(Anew:-Body)$\rangle$

if Anew neither has a definition nor is called in P or Clause.

The folding transformation

$\langle$(Head:-Before,A,After), P$\rangle \Leftrightarrow \langle$(Head:-Before,F$\psi$,After), P$\rangle$

if there is a clause F:–G in P, where F is defined by just this clause and all variables in G occur in F. A is an instance of G such that A=Gψ.

The unfolding transformation

$\langle$(Head:-Before,A,After), P+$\{A_i :- B_i\}_i\rangle$

$\Leftrightarrow \langle$(Head :- Before, $\bigvee_i$(A=A_i, B_i), After), P+$\{A_i :- B_i\}_i\rangle$

or, if A is the first goal, i.e. Before is empty,

$\Leftrightarrow \langle\{$(Head :- B_i, After)θ | mgu(A,A_i)=$\theta \wedge \theta \neq$false$\}$, P+$\{A_i :- B_i\}_i\rangle$

2.3.3 The partial evaluation procedure

Although Mixtus externally behaves like a partial evaluator, internally the procedure it uses more resembles mixed computation. In section 1.6 it is shown how to obtain a partial evaluator from a mixed computator.

Four mutually recursive functions are used in the partial evaluation procedure: $\Pi_{clauses}$, Π_{clause}, Π_{goals} and Π_{goal}. The mixed computation Π_{mixed} starts with a query Q and a program P.

DEFINITION OF Π_{mixed}

```
procedure Π_mixed(P,Q)=⟨P', Q'⟩
Π_mixed(P,Q)≡⟨Clauses+P'', Q'⟩
        where ⟨Clauses, P''⟩=Π_clauses⟨Q':-Q, P, [] ⟩
        where Q' = reparametrization(Q,P)
```

The procedure call reparametrization(Q,P) returns a new unique predicate name distinct from all predicate names in Q and P, with the distinct variables of Q as arguments.

CONDITION FOR CORRECTNESS Π_{mixed}

Recall the conditions in the definition of a mixed computator. For all instances Qθ of Q:

① P'[Q'θ] produces the same results as P[Qθ] and in the same order.

② The computation of Π_{mixed}(P,Q) will always terminate for all P and Q.

PROOF

Item ② is discussed in the section on loop-prevention, where strategies are shown which guarantee termination.

Item ① is proven here. Our starting point is the program (Q':-Q)+P which can handle all queries that are instances of Q'.

$Q'\text{:-}Q+P \Leftrightarrow Q'\text{:-}Q, P$ as Q' is not defined in P

$\Rightarrow \langle Clauses, P'' \rangle$ by correctness of $\Pi_{clauses}$ as defined below

$\Leftrightarrow$ Clauses+P'' as the predicate of Clauses is not defined in P''

which is the program returned by Π_{mixed} together with the new query Q'.

DEFINITION OF $\Pi_{clauses}$

```
procedure Π_clauses⟨Clauses, Program, Goalstack⟩=⟨Clauses_out, Program_out⟩
Π_clauses⟨Clauses, Program, Goalstack⟩ ≡
    if Clauses=[] then ⟨[], Program⟩
    else  let [Clause|Clauses']=Clauses in
          let ⟨Clauses'', Program'' ⟩=Π_clause⟨Clause, Program, Goalstack⟩ in
          let ⟨Clauses''', Program'''⟩=Π_clauses⟨Clauses', Program'', Goalstack in ⟩
                ⟨Clauses''+Clauses''', Program''' ⟩
```

The definition of $\Pi_{clauses}$ as shown above differs in several formal respects from the version presented in the informal description of the algorithm. These differences are necessary to make $\Pi_{clauses}$ truly functional.

The program is no longer changed as a side-effect of the execution. Instead the program occurs as an explicit input argument, and the new program is the second component of the result.

The third argument of $\Pi_{clauses}$ is a list of clauses, with elements of the form $(Anew_i :- A_i)$.

CONDITION FOR CORRECTNESS OF $\Pi_{clauses}$

$\langle Clauses, Program+Goalstack \rangle \Rightarrow \langle Clauses_{out}, Program_{out}+Goalstack \rangle$

For the proof we first need the following lemma.

LEMMA

If $\langle A, P_1 \rangle \Rightarrow \langle A', P_2 \rangle$ and $\langle B, P_2 \rangle \Rightarrow \langle B', P_3 \rangle$ then $\langle A+B, P_1 \rangle \Rightarrow \langle A'+B', P_3 \rangle$.

This follows from the definition of the result of a query to a p-program and the transitivity of the relation $\Rightarrow$.

PROOF OF THE CORRECTNESS

The base case when Clauses=[] is correct as the correctness then means that

$\langle [], Program+Goalstack \rangle \Rightarrow \langle [], Program+Goalstack \rangle$

For the induction we have Clauses=Clauses'+Clause. From the correctness of Π_{clause} (see below) we get

‹Clauses, Program+Goalstack› ⇒ ‹Clauses", Program"+Goalstack›.

By induction of the number of clauses we get

‹Clauses', Program"+Goalstack› ⇒ ‹Clauses'", Program'"+Goalstack›.

By use of the lemma above the correctness of $\Pi_{clauses}$ is proven.

DEFINITION OF Π_{clause}

```
procedure Π_clause⟨Clause, Program, Goalstack⟩= ⟨Clauses_out, Program_out⟩
Π_clause⟨Clause, Program, Goalstack⟩ ≡
    let H:-G_1,...,G_n = Clause
    let ⟨Clauses_1, Program_1⟩ = ⟨{Clause}, Program⟩
    for each goal G_k in G_1,...,G_n
        let ⟨Clauses_k+1, Program_k+1⟩=Π_goals⟨G_k, Clauses_k, Program_k, Goalstack⟩
    end for
    return ⟨Clauses_n+1, Program_n+1⟩
```

CONDITION FOR CORRECTNESS OF Π_{clause}

‹Clause, Program+Goalstack›⇒‹$Clauses_{out}$, $Program_{out}$+Goalstack›

PROOF

Assuming that Π_{goals} is correct (which is shown below), then for each iteration in the for-loop we have

‹$Clauses_k$, $Program_k$+Goalstack› ⇒ ‹$Clauses_{k+1}$, $Program_{k+1}$+Goalstack›

The correctness then follows from the transitivity of ⇒.

DEFINITION OF Π_{goals}

```
procedure Π_goals⟨G, Clauses, Program, Goalstack⟩= ⟨Clauses_out, Program_out⟩
Π_goals⟨G,Clauses, Program, Goalstack⟩ ≡
    if Clauses=[] then ⟨[], Program⟩
    else  let [Clause|Clauses']=Clauses in
          let ⟨Clauses", Program"  ⟩=Π_goal⟨G, Clause, Program, Goalstack⟩ in
          let ⟨Clauses'", Program'"⟩=Π_goals⟨G, Clauses', Program", Goalstack⟩ in
              ⟨Clauses"+Clauses'", Program'"  ⟩
```

CONDITION FOR CORRECTNESS OF Π_{goals}

‹Clauses, Program+Goalstack›⇒‹$Clauses_{out}$, $Program_{out}$+Goalstack›

PROOF

This procedure has a structure very similar to $\Pi_{clauses}$, but relies on the correctness of Π_{goal} instead, which will be shown below.

DEFINITION OF Π_{goal}

```
procedure Π_goal(A, Clause, Program, Goalstack) = ⟨Clauses_out, Program_out⟩
Π_goal(A, Clause, Program, Goalstack) ≡
let (Head:-Before,A,After) = Clause
if A is a call to a built-in predicate then
        if evaluable(A) then
                let θ_i be the substitutions resulting from executing A
                if Before is empty then
                        return ⟨{(Head :- After)θ_i }_i , Program⟩
                else
                        return ⟨(Head:-Before, ⋁_i A=Aθ_i , After ), Program⟩
        else
                return ⟨(Head:-Before,A,After), Program⟩
else A is a call to a user-defined predicate
        if F:–G ∈ Goalstack such that exists a substitution ψ such that A=Gψ then
1:              return ⟨(Head:-Before,Fψ,After), Program⟩
        else if loop_prevention(A, Goalstack)
2:              return ⟨(Head:-Before,A,After), Program⟩
        else
```

3&4: let A be defined by the clauses $\{H_i :\!- B_i\}_i$
let Anew = reparametrization(A, Program+Goalstack)
let A',Anew' be a variable variant of A,Anew
let ρ be the substitution Anewρ=Anew'
let $\langle\{(\text{Anew}' :\!- C_j)\varphi_j\}_j, \text{Program}_2\rangle =$
$\Pi_{\text{clauses}}(\{(\text{Anew}' :\!- B_i)\theta_i\}_i, \text{Program}, \text{Goalstack}+(\text{Anew}' :\!- \text{A}'))$
if the predicate Anew is recursive in $\text{Program}_2+\{(\text{Anew}' :\!- C_j)\varphi_j\}_j$
3: return $\langle\langle \text{Head} :\!- \text{Before,Anew,After}), \text{Program}_2+\{(\text{Anew}' :\!- C_j)\varphi_j\}_j \rangle$
else
4: if Before is empty
return $\langle\{(\text{Head} :\!- C_j, \text{After})\rho\varphi_j\}_j, \text{Program}_2\rangle$
else
return $\langle(\text{Head} :\!- \text{Before}, \bigvee_i(\text{Anew}=\text{Anew}' \ \varphi_j C_j \varphi_j), \text{After}),$
$\text{Program}_2+\{(\text{Anew}' :\!- C_j)\varphi_j\}_j\rangle$

CONDITION FOR CORRECTNESS OF Π_{goal}

‹Clause, Program+Goalstack› $\Rightarrow$ ‹Clauses_{out}, Program_{out}+Goalstack›

PROOF

Let the Clause be "Head:-Before,A,After", where A is the goal indicated by the first argument of Π_{goal}.

If A is a call to a built-in goal, the treatment depends whether the predicate is evaluable or not. In the latter case, the goal is kept as it is, which is obviously correct, and in the former, we will assume that the built-in predicate is handled correctly.

Otherwise A is a call to a user-defined predicate and the basic mechanism for Mixtus is used as described in the previous section.

Then there are four cases: folding, loop prevention, definition and unfolding. It will be shown below that the correctness is maintained for all of these cases. Our initial p-program is

S = ‹(Head:-Before,A,After), Program+Goalstack›

Case 1: Folding

If there is a clause F:–G in "Goalstack" and A is an instance of G (i.e. A=Gψ), then through *folding* we get

Clauses_{out} = Head:-Before,Fψ,After

Program_{out} = Program

This result is correct, since it corresponds to a folding on the p-program S, producing

$S \Leftrightarrow$ ‹(Head:-Before,Fψ,After), Program+Goalstack›

Case 2: Loop prevention

In case of loop prevention, the original clause is kept as it is, and the output program is the same as the input program. Clearly, this identity transformation is correct.

Cases 3 & 4:

In both these cases a recursive invocation is made to the partial evaluator. Let Anew be a reparametrization of A. Let Anew' be a variable variant of Anew and let the substitution ρ be such that Anewρ=Anew'. Then let Aρ=A'.

First *definition* is used to produce a new clause (Anew':-A') which is added to the set of foldable clauses. The p-program then becomes

$S \Rightarrow$ ‹(Head:-Before,A,After), Program+Goalstack+(Anew':-A')› = S_2

Then *folding* is applied to the goal A, producing

$S_2 \Leftrightarrow$ ‹(Head:-Before,Anew,After), Program+Goalstack+(Anew':-A')› = S_3

We now focus our attention on the predicate Anew and its definition, which will be partially evaluated. The program P' will handle all queries to Anew just as S_3 does.

P' = Program+Goalstack+(Anew':-A')

Similarly, the p-program S' will also handle all queries to Anew just as S_3 does.

P' $\Rightarrow$ ‹Anew':-A', Program+Goalstack+(Anew':-A')› = S'

Assuming that A is defined by the clauses $\{H_i :\!- B_i\}_i$, the first goal in the body of the clause (Anew':-A') is *unfolded*

S' $\Leftrightarrow$

‹{(Anew':-B_i)θ|θ=mgu(A',H_i)∧θ≠false},Program+Goalstack+Anew':-A'› =S_1'

Partial evaluation of the newly generated clauses is now recursively invoked with procedure

$\Pi_{clauses}\langle\{$(Anew':-B_i)$\theta_i\}_i$, Program, Goalstack+(Anew':-A'.) $\rangle$

which returns the tuple $\langle Clauses_2, Program_2\rangle$, where $Clauses_2$ is of the form $\{$(Anew':-C_j)$\varphi_j\}_j$. Assuming the correctness of $\Pi_{clauses}$ we get

S_1' $\Rightarrow$‹{(Anew':-C_j)$\varphi_j\}_j$, $Program_2$+Goalstack+(Anew':-A')› = S_2'

All calls to the predicate (Anew':-A') are handled equally well by its partially evaluated version $\{$(Anew':-C_j)$\varphi_j\}_j$ which can replace it and we get

S_2' $\Rightarrow$P$rogram_2$+Goalstack+$\{$(Anew':-C)$\varphi_j\}_j$ = P"

As P' $\Rightarrow$P" then P' may be replaced by P" in S_3

$S_3 \Rightarrow \langle$(Head:-Before,Anew,After), $Program_2$+Goalstack+{(Anew':-C)φ_j}$_j\rangle = S_4$

We have now come to the point where cases 3 and 4 differ.

Case 3: definition

If $Clauses_2$ are found to be recursive, the result from Π_{goal} is

$Clauses_{out}$ = Head:-Before,Anew,After
$Program_{out} = Program_2 + \{(Anew':-C_j)\varphi_j\}_j$

which makes

$S \Rightarrow S_4 =$ ‹(Head:-Before,Anew,After), $Program_2$+Goalstack+{(Anew':-C)φ_j}$_j$›

Case 4: unfolding

Otherwise, if $Clauses_2$ are not recursive, *unfolding* is done of Anew, and if Before is non-empty we get

$$S \Rightarrow S_4 = \langle(\text{Head} :- \text{Before}, \bigvee_i (\text{Anew}=\text{Anew}' \ \varphi_j C_j \varphi_j), \text{After}),$$
$$Program_2 + Goalstack + \{(Anew':-C)\varphi_j\}_j\rangle$$

whereas if Before is empty

$S \Rightarrow S_4 =$ ‹{(Head :- C_j, After)$\rho\varphi_j\}_j$, $Program_2$+Goalstack+{(Anew':-C)φ_j}$_j$›

As we know that the predicate Anew is not called anywhere (it was non-recursive), we can remove its definition {(Anew':-C)φ_j}$_j$ from the final program. The result then becomes

$$Clauses_{out} = (\text{Head} :- \text{Before}, \bigvee_i (\text{Anew}=\text{Anew}' \ \varphi_j C_j \varphi_j), \text{After})$$
$$\text{or } \{(\text{Head} :- C_j, \text{After})\rho\varphi_j\}_j$$
$Program_{out} = Program_2$

We have now shown that the correctness is maintained for all cases.

2.4 Reparametrization

As already mentioned, reparametrization is the process of taking a goal A, extracting all the distinct variables in A and creating a new atomic goal Anew having these variables as arguments and a new unique functor name. Reparametrization is used in the “definition” transformation. Some refinements of and problems with reparametrization are discussed in this section.

2.4.1 Order of variables

It is generally preferable to arrange it so that the order of the variables in Anew corresponds to a left to right depth first traversal of the variables in A. This will improve the possibilities of indexing, as indexing is only performed on the first argument in SICStus Prolog, and a common

recommendation regarding programming style is to put the instantiated input arguments first in a goal.

2.4.2 Trimming variables

Another refinement of reparametrization in Mixtus is the following: If some variables in a call occur neither before nor after the call, these variables are not included in Anew. These variables are called "trimmed variables". The other variables are called the "surrounding variables".

For example, let us consider partial evaluation of the query `in_list(X,L)` and this program:

```
in_list(X,L) :- select(X,L,Lrest).

select(X,[X|Lrest],Lrest).
select(X,[Y|L],[Y|Lrest]) :- select(X,L,Lrest).
```

The variable `Lrest` in the call `select(X,L,Lrest)` is a trimmed variable as it does not occur anywhere else in the first clause. Thus the following clause is added to the goal-stack.

```
select2(X2,L2):-select(X2,L2,Lrest2).
```

Variables occurring in the body of this clause but not in the head (such as `Lrest2`) are called "internal variables". As shown in [Tama84] folding of a goal is correct if after unification of the goal and the body of the defining clause, all the internal variables remain uninstantiated, distinct from each other and from other variables surrounding the goal. These conditions must be checked for in all subsequent applications of the folding operation.

In our example partial evaluation is now recursively invoked to yield clauses

```
select2(X,[X|Lrest]).
select2(X,[Y|L]) :- select(X,L,Lrest).
```

Again the variable `Lrest` in the recursive call does not occur elsewhere in the clause. Before folding may be applied to the recursive call it must first be checked whether the above condition is satisfied. As this is the case, the final result becomes

```
in_list(X,L) :- select2(X,L).

select2(X,[X|Lrest]).
select2(X,[Y|L]) :- select2(X,L).
```

and the unused argument of `select/3` has been trimmed away.

To summarize, the changes needed to introduce trimming are:

1. Only the variables surrounding the goal are kept in Anew.
2. Using the folding clause

   Anew' :- A'.

the clause

Head :- Before, A, After.

can be folded into

Head :- Before, Anew' θ, After.

provided

- There is a substitution θ such that A=A' θ.
- All internal variables in A' after the substitution A'θ remain variables distinct from each other and from the variables in Head, Before and After.

2.4.3 Problems with a large number of arguments

When a predicate call containing a large compound term is partially evaluated, the compound term will be eliminated, and only the distinct variables of the term will appear as separate arguments. The advantage of this transformation is that at run-time the corresponding structure does not have to be created and the access to the components of the structure becomes faster. The disadvantage is that more arguments may have to be carried around in the predicate call.

In most implementations based on WAM, the first 4–5 arguments of a predicate are particularly efficiently handled, as they are stored in machine registers. A program transformation which increases the number of arguments may therefore be disadvantageous. The current partial evaluator does not carry out any checks to prevent the creation of predicates with a large number of arguments.

2.4.4 Reparametrization might be disadvantageous

Right-propagation of bindings is crucial for enabling the partial evaluator to produce more specialized and efficient code. As mentioned earlier, there is a danger that a term becomes duplicated as it gets propagated. This would make the resulting program less efficient than the original one. Normally this does not happen, as the term is "consumed" either by partial evaluation of a built-in predicate or, as described above, by partial evaluation of a user-defined predicate. In the latter case, the term is normally removed as the goal is replaced by a new goal containing just the variables in the original goal. In some very special cases, this reparametrization mechanism may not be advantageous. For example, given the goal `p(t(A),L)` and the program

```
p(X,[]).
p(X,[H|T]) :- q(X,H), p(X,T).
```

partial evaluation will produce the new goal `p2(A,L)` and the program

```
p2(A,[]).
p2(A,[H|T]) :- q(t(A),H), p2(A,T).
```

The problem is that the same structure `t(A)` is recreated for each recursion instead of just once. Mixtus makes no attempts at recognizing and handling these cases specifically, and may therefore sometimes generate poor code. Fortunately, cases like this seem to occur very seldom.

2.5 Loop prevention

Due to the unsolvability of the halting problem, a loop prevention mechanism can never be perfect; it will always be a compromise. In this case, we have chosen to diagnose some terminating computations as loops, thereby guaranteeing termination of the program in all circumstances.

This is quite a different issue than “loop check” or “loop detection” as described for instance in [Bol90], where only procedure calls which are sure to loop are considered looping. These execution branches may be pruned as they cannot produce any solutions. It seems the algorithms for loop detection and loop prevention have little in common. At the suggestion of Roland Bol the term “loop prevention” is therefore used here to distinguish the two mechanisms. Note however that most literature on partial evaluation has been using the term loop detection for what we here call loop prevention.

A loop prevention mechanism may be arbitrarily refined and complicated. If the mechanism is too simple, it may be unsatisfactory. One shortcoming can be that many non-loops are classified as loops. This will stop the partial evaluation prematurely, and the quality of the code generated will not be so good. Another shortcoming may be that it takes too much time (i.e. a recursion has to be carried out to a great depth) before detecting that a looping program actually loops. This will make the partial evaluation take a very long time. Normally the code produced will be of reasonably good quality (although possibly unnecessarily large), but there are cases where it could be improved by an earlier loop prevention. The goal is to find an advanced mechanism that will quickly and almost always correctly determine whether a program loops.

Here a sequence of increasingly complex alternative loop preventions are presented for a goal-stack Goalstack and a goal A. Although the Goalstack consists of clauses of the form ($Anew_i$':-A_i'), we are in the following only interested in the A_i's, and will ignore the $Anew_i$'s.

For the purposes of loop prevention, detailed Prolog semantics may be ignored and the goals in the goal stack can be viewed simply as a sequence of Prolog terms: t_1,t_2,…. Thus the mechanisms presented are valid for any programming language. The goal of loop prevention is to prevent the goal stack from growing ad infinitum. It is guaranteed that the test loop_prevention(A,Goalstack) will eventually succeed for any goal stack that grows indefinitely. That is, for any infinite sequence of Prolog terms t_1,t_2,…. there exists initial segment of n terms t_1,t_2,….t_n (=Goalstack), and an element t_{n+1} (=A), such that loop_prevention(A,Goalstack) succeeds.

We will now describe a number of loop_prevention tests all having the following form:

DEFINITION of loop_prevention:

"loop_prevention(A,Goalstack)" succeeds if there are at least *maxrec* terms A_i' in Goalstack such that the test "may_loop(A_i',A)" succeeds.

Different variants of "may_loop" are given below all satisfying the following condition.

CONDITION for may_loop:

Let may_loop be constructed so that for any infinite sequence of Prolog terms $t_1,t_2,\ldots$ there is an infinite sub-sequence $t_{k_1},t_{k_2},\ldots$ such that may_loop(t_{k_i},t_{k_j}) holds for all i<j.

LEMMA

loop_prevention as defined above using a may_loop satisfying this condition is guaranteed to prevent the stack from growing ad infinitum.

PROOF

According to the condition for may_loop, if there is an infinite sequence of Prolog terms, there is an infinite sub-sequence $t_{k_1},t_{k_2},\ldots$ such that may_loop(t_{k_i},t_{k_j}) holds for all i<j. Then for any *maxrec* there exists a term $t_{k_{maxrec+1}}$ such that may_loop($t_{k_i},t_{k_{maxrec+1}}$) holds for the *maxrec* terms t_{k_i}, i≤*maxrec*. This is the test in loop_prevention, so it is guaranteed that an infinitely growing stack will be detected.

Three kinds of may_loop

To simplify the proofs below, the following definition is given.

DEFINITION

A "may_loop-ordered" sequence is a (finite or infinite) sequence $t_1,t_2,\ldots$ such that may_loop(t_i,t_j) holds for all i<j.

Below three kinds of may_loop satisfying the condition above are given.

Partition

may_loop forms a finite *partition* as it can be defined as may_loop(s,t) ≡ f(s)=f(t) for some function f having a finite range.

PROOF that the conditions for may_loop are satisfied:

Using the function f any infinite sequence can be partitioned into at most N disjoint sub-sequences (where N is the cardinality of the range of f). At least one of these sub-sequences must be infinite. As this infinite sequence is "may_loop-ordered", the conditions for may_loop are satisfied.

Natural numbers

may_loop can be written as may_loop(s,t) ≡ $f(s) \leq f(t)$ for some function f with values in the natural numbers.

PROOF that the conditions for may_loop are satisfied:

For the relation >, defined by s>t ≡ ¬may_loop(s,t), there cannot be any infinite decreasing sequence $t_1 > t_2 > \ldots$ as the converse of > is a well-ordering. But, according to the lemma below, there then must exist an infinite non-decreasing sub-sequence $t_{k_1} \not> t_{k_2} \not> \ldots$ Due to the transitivity of $\not>$ this means that $t_{k_i} \not> t_{k_j}$ if i<j. As may_loop(s,t) is s$\not>$t then $t_{k_1}, t_{k_2} \ldots$ is may_loop-ordered and the proof is completed.

LEMMA

If an infinite sequence $t_1, t_2, \ldots$ does not have an infinite non-decreasing sub-sequence $t_{k_1} \not> t_{k_2} \not> \ldots$ (where $k_i < k_{i+1}$) it must have an infinite decreasing sub-sequence $t_{m_1} > t_{m_2} >, \ldots$ (where $m_i < m_{i+1}$).

PROOF of lemma

According to the assumption, there is a finite non-extendible non-decreasing sub-sequence $t_{k_1} \not> t_{k_2} \not> \ldots$. Let its last element be t_{m_1}. Then $t_{m_1} > t_p$ for all $p > m_1$.

The sub-sequence $t_{m_1+1}, t_{m_1+2}, \ldots$ also have an infinite non-decreasing sub-sequence. The same argument as above may be applied again, and the last value of the non-decreasing sub-sequence is called t_{m_2}. As all elements (including t_{m_2}) in the sub-sequence $t_{m_1+1}, t_{m_1+2}, \ldots$ are less than t_{m_1}, then $t_{m_1} > t_{m_2}$.

A new sub-sequence $t_{m_2+1}, t_{m_2+2}, \ldots$ may now be formed, and a new t_{m_3} being $t_{m_2} > t_{m_3}$ can be found. This procedure may be repeated indefinitely, producing an infinite decreasing sequence $t_{m_1} > t_{m_2} > t_{m_3} \ldots$.

Conjunction

may_loop is a finite *conjunction* of other functions fulfilling the conditions for may_loop, that is, may_loop(s,t) ≡ $\wedge_i$ may_loop$_i$(s,t)

PROOF that the conditions for may_loop are satisfied:

The proof is given just for a single conjunction, but can by using induction be extended to a finite conjunction.

Assume that may_loop$_1$ and may_loop$_2$ fulfil the conditions for a may_loop. Then given an infinite sequence there is an infinite sub-sequence that is a may_loop$_1$-ordered. Using this infinite sub-sequence there must be an infinite sub-sub-sequence that is may_loop$_2$-ordered. As this sub-sub-sequence is also may_loop$_1$-ordered, we have an infinite sequence which is may_loop-ordered and the proof is concluded.

We will now give several instances of may_loop satisfying the condition above that we have been found to be very useful.

Loop prevention: predicate name

may_loop(A',A) succeeds if A' and A are calls to the same predicate, i.e. when viewed as terms they have the same outermost functor. This is an example of using a partition to define may_loop, which guarantees that the conditions for may_loop are satisfied.

This extremely simple test used with *maxrec*=10 has been surprisingly successful.

Loop prevention: functor names

A related necessary condition for an infinite loop is that the same functor name occurs again in the same argument position. Since Prolog is capable of creating new functor names dynamically (using `name/2`), some precautions have to be taken so that this test is not applied to those dynamically created names. A distinction is therefore made between the functors in the original program (the static functors) and the other "dynamic" functors. All dynamic functors are considered the same when compared in this test.

```
may_loop(t,s) ≡
    if arity(t)=arity(s) then
        ∀ i∈{1,…arity(s)} may_loop_argument(argument(t,i),argument(s,i))
    else
        false
where
    may_loop_argument(t,s) =
        if s is not a static functor then
            return true
        else if t and s have the same functor then
            return true
        else
            return false
and where
    arity(s) returns the number of arguments of s
    argument(x,i) returns the i:th argument of x
```

The conditions for may_loop are satisfied, because the above test is a finite conjunction of may_loop_argument tests, and these in turn are based on a finite partitioning of the Prolog terms.

Loop prevention: structures

A natural extension of comparing functor names is to compare whole structures. One problem is that there are an infinite number of structures even if they are only composed of static functors. For instance, the sequence s(0), s(s(0)), s(s(s(0))) may be extended infinitely. But there are only a

finite number of structures having a certain maximum nesting depth, say *max_depth*. This leads to the following loop prevention test:

```
may_loop(t,s) ≡ may_loop2(t,s,max_depth)
where
   may_loop2(t,s,depth) ≡
      if depth=0 or s is not a static functor then
          return true
      else if t and s have the same functor then
          return ∀ i∈{1,…arity(s)} may_loop2(argument(s,i),argument(t,i),depth-1)
      else
          return false
```

The above test is not capable of handling some cases where deeper structures are used and a deeper recursion is needed to compute the result. For instance, the query

```
append([1,2,3,4,5,6,7,8,9,a,b,c,d,e,f|A],B,C)
```

is better completely evaluated. (`append/3` is defined in appendix B) The following loop prevention criterion will handle this case correctly.

Loop prevention: total argument size

As long as the total size of the arguments shrinks (in the append-case above: the list gets shorter), it is safe to recurse as this process cannot go on indefinitely. This leads to the following version of loop prevention.

may_loop(t,s) ≡ termsize(t) ≤ termsize(s)

where the function termsize(x) is defined recursively on Prolog terms as

$$\text{termsize}(x) = \begin{cases} 1 \text{ if x is a constant or a variable} \\ 1 + \sum_{i=1}^{\text{arity}(x)} \text{termsize}(\text{argument}(x,i)) \text{ if x is a compound term} \end{cases}$$

Loop prevention: single argument size

Similarly, the size of a single argument may be used as a loop prevention check. As long as that argument shrinks, there is no danger of going into a loop. This will handle programs where one argument grows during recursion but some other argument gets smaller.

For example, given the query `reverse([1,2,3],[],X)` and the program

```
reverse([],A,A).
reverse([A|B],C,D) :- reverse(B,[A|C],D).
```

the first argument shrinks whereas the second argument grows.

Combining several loop preventions

As shown earlier, a new necessary condition for looping can be created by forming the conjunction of several different loop prevention tests "may_loop". For example, a conjunction of may_loop-tests may be formed that checks for every argument that it does not grow or stay the same in size.

Although that method is insensitive to the order of the arguments, it is sensitive to whether the "active" arguments are imbedded into a "passive" structure. Take the `reverse2/1` program which is identical to `reverse/3` above except in having all arguments imbedded in a structure.

```
reverse2(f([],A,A)).
reverse2(f([A|B],C,D)) :- reverse2(f(B,[A|C],D)).
```

Given the query `reverse2(f([1,2,3],[],X))` the size of the single argument will not decrease, the sequence of calls being

```
reverse2(f([1,2,3],[],X))
reverse2(f([2,3],[1],X))
reverse2(f([3],[2,1],X))
```

Whereas the just presented version of may_loop would have signalled for a loop, the following refined version will not do so. If the arguments of the structure are compared in size instead of the argument of the predicate, the loop prevention mechanism will be able to detect that the above call does not loop. This can be achieved by combining the test for comparing structures with the test for comparing the size of the structures. The only change needed is that when the maximum depth is reached, instead of signalling a potential loop, those substructures are compared in size, and we get

may_loop(t,s) ≡ may_loop2(t,s,*max_depth*)
where
  may_loop2(t,s,depth) ≡
    if depth=0 or s is not a static functor then
      return termsize(t) ≤ termsize(s)
    else if t and s have the same functor then
      return ∀ i∈{1,…arity(s)} may_loop2(argument(s,i),argument(t,i),depth-1)
    else
      return false

The correctness of the above test with respect to the conditions for may_loop follows from the fact that it may be viewed as a conjunction of

- the may_loop using structures
- a finite conjunction of may_loop tests that use the total argument size

A good loop prevention mechanism is crucial for making the partial evaluator able to automatically determine whether it is worth unfolding. The above method has proved itself very useful for a long range of problems. All examples in chapter 6 have been run automatically with *maxrec*=2 and *max_depth*=4.

2.6 Some examples of partial evaluation of pure Prolog programs

Some examples will illustrate the basic ideas on how the partial evaluator will automatically choose between folding, unfolding and definition.

These examples are meant to be read in conjunction with the description of the main procedure for partial evaluation in section 2.2.5. The examples are illustrated by a table of the following form. A is the user-defined goal to be partially evaluated. A and Anew are related as described above. For each clause of A a unification is made and the corresponding instance of Anew is created. Partial evaluation then continues with the body of the clause.

<table>
<tr><td>goal-stack</td><td>Anew</td><td>A</td><td></td></tr>
<tr><td>clause i
head H_i</td><td>Anewθ
new head after unification</td><td>Aθ (i.e. $H_i\theta$)
new head after unification</td><td rowspan="2">θ = the substitution resulting from unifying A and H_i</td></tr>
<tr><td>body B_i</td><td colspan="2">$B_i\theta$
body after unification</td></tr>
</table>

Example: folding

Let the query be `m(X,[1|Y])` and the program be the member program:

```
clause 1:       m(A,[A|B]).
clause 2:       m(C,[D|E]) :- m(C,E).
```

We then get the following table.

	Anew	A	substitution θ
goal-stack	m2(X,Y)	m(X,[1|Y])	
clause 1	m2(1,Y).	m(1,[1|Y]).	{1/X}
clause 2 head	m2(X,Y)	m(X,[1|Y])	{X/C, 1/D, Y/E}
body	:- m(X,Y).		
As neither folding nor loop prevention is applicable, partial evaluation then continues with the goal m(X,Y) in the body.			
goal-stack	m3(X,Y)	m(X,Y)	
clause 1	m3(X,[X|B2]).	m(X,[X|B2]).	{X/A2, [X|B2]/Y}
clause 2 head	m3(X,[D2|E2])	m(X,[D2|E2])	{X/C2, [D2|E2]/Y}
body	:- m(X,E2).		
folding	:- m3(X,E2).		

That is, the new query is `m2(X,Y)` and the generated clauses are

```
m2(1,Y).
m2(X,Y) :- m3(X,Y).
m3(X,[X|B2]).
m3(X,[D2|E2]) :- m3(X,E2).
```

As `m2/2` is the top-level predicate and `m3/2` is recursive, no unfolding is made for these predicates. Note that `m3/2` happens to be identical to `m/2`.

Example: loop prevention and unfolding

The predicate call `r(X,[],3,Z)` will reverse the elements in the list `X`, multiply each element by 3 and return the new list in `Z`.

```
r([],L,P,L).
r([B|C],D,P,E) :- F is B*P, r(C,[F|D],P,E).
```

A partial evaluation of `r(X,[],3,Z)` is summarized by the following table.

	Anew	A	substitution θ
goal-stack	r2(X,Z)	r(X,[],3,Z)	
clause 1	r2([],[]).	r([],[],3,[]).	{[]/X, []/L, 3/P, []/Z}
clause 2 head	r2([B|C],Z)	r([B|C],[],3,Z)	{[B|C]/X, []/D, 3/P, Z/E}
body	:- F is B*3, r(C,[F],3,Z).		
No folding or loop prevention, the partial evaluator is recursively called with r(C,[F],3,Z).			
goal-stack	r3(C,F,Z)	r(C,[F],3,Z)	
clause 1	r3([],F,[F]).	r([],[F],3,[F]).	{[]/C,[F]/L2,3/P2, [F]/Z}
clause 2 head	r3([B2|C2],F,Z)	r([B2|C2],[F],3,Z)	{[B2|C2]/C,[F]/D2, 3/P2,Z/E2}
body	:- F2 is B2*3, r(C2,[F2,F],3,Z)		
No folding is possible. The goal stack is now r(X,[],3,Z), r(C,[F],3,Z), r(C2,[F2,F],3,Z). The loop prevention mechanism indicates a possible loop if *max_depth*=1 and *maxrec*=2 (see section 2.5). The goal is therefore kept as it is, referring to the original clauses.			

To summarize, the clauses generated so far are

```
r2([],[]).
r2([B|C],Z) :- F is B*3, r3(C,F,Z)
r3([],F,[F]).
r3([B2|C2],F,Z) :- F2 is B2*3, r(C2,[F2,F],3,Z).
```

The call `r3(C,F,Z)` shall now be *unfolded* as the predicate `r3/3` is not recursive. For simplicity, we here use the logical unfolding instead of Prolog unfolding as the goal `is/2` cannot create any choice-points.

```
r2([],[]).
r2([B],[F]):- F is B*3.
r2([B,B2|C2],F,Z):-
        F is B*3, F2 is B2*3, r(C2,[F2,F],3,Z).
```

As can be seen from the above code, special cases have been generated for lists of length 0 and 1, whereas longer lists are handled by the original predicate `r/4`. By changing the loop prevention criterion, we could have generated even more cases for longer lists, but we would always have to resort to using the original predicate for longer lists. It is questionable whether generating more and more clauses actually yields an execution performance improvement in this

case. In the next section, we show how to avoid calling the original predicate by generating a predicate further specialized so that the number 3 need not be carried around as an explicit argument.

2.7 Generalized restart

The mechanism for generalized restart, as described below, will ensure that folding will always eventually apply and that the partial evaluation procedure will always terminate anyway. As folding replaces a goal with a new goal, this means that there will be no references in the program produced by partial evaluation to the original program. The main advantage of this is that more user-defined predicates are optimized. A minor advantage is that no renaming is needed, as described in section 1.6, when converting the result from mixed computation into a result for partial evaluation. Thus, the role of loop prevention is now to trigger the mechanism for generalized restart rather than to terminate the partial evaluation prematurely.

2.7.1 Anti-unification

The mechanism for generalized restart uses “anti-unification” [Reyn70, Lass88] (also called inductive generalization in [Plot70], most common generalization in [Reyn70] and most specific generalization in [Benk89]). First some definitions:

A term t_1 *is more general* than a term t_2, if t_2 is an instance of t_1, that is, there exists a substitution θ such that $t_2=t_1\theta$. Let this be written as $t_1 \geq t_2$ or alternatively $t_2 \leq t_1$.

A term t is *the unification of* two terms t_1 and t_2 if
- $t_1 \geq t$
- $t_2 \geq t$
- for all terms s, if $t_1 \geq s$ and $t_2 \geq s$ then $t \geq s$.

A term t is *the anti-unification of* two terms t_1 and t_2 if
- $t_1 \leq t$
- $t_2 \leq t$
- for all terms s, if $t_1 \leq s$ and $t_2 \leq s$ then $t \leq s$.

In contrast to the unification of two terms, there always exists a term which is the anti-unification of two arbitrary terms. In both cases, this term is unique up to variable renaming. The classical reference for unification is Robinson [Robi65]. The algorithm for anti-unification given in [Plot70] is implemented by `anti_unify/3` as shown in appendix A.3.

Some examples will illustrate the very useful concept of anti-unification:

`anti_unify(f(a,b,c),f(c,b,a),A)` returns `A=f(X,b,Y).`
`anti_unify(f(a,a,b),f(b,b,b),A)` returns `A=f(X,X,b).`

2.7.2 Generalized restart

When a loop is detected, the basic algorithm will just keep a call to the original code. By looking at the goal-stack in the last example more carefully we can do better. The three recursive calls to `r/4` were

```
r(X,[],3,Z), r(C,[F],3,Z), r(C2,[F2,F],3,Z)
```

These goals form a may_loop-ordered sequence, as defined in section 2.5. Clearly the second argument varies and the third argument is constant. In general this can be found out by doing an "anti-unification" between recursive calls.

An anti-unification generates the most specific term which is more general than all terms in a set. This term is unique up to renaming of variables. In this case we get the term `r(S,T,3,U)`. A "generalized restart" means that whenever a loop is detected, a generalization of the recursive calls is generated, and partial evaluation is restarted with that new goal.

For the above example we get after restart:

	Anew	A	substitution θ
goal-stack	r4(S,T,U)	r(S,T,3,U)	
clause 1	r4([],T,T).	r([],T,3,T).	{[]/S, Y/L, 3/P, T/U}
clause 2 head	r4([B|C],T,U)	r([B|C],T,3,U)	{[B|C]/S, T/D, 3/P, U/E}
body	:- F is B*3, r(C,[F|T],3,U).		
folding	:- F is B*3, r4(C,[F|T],U).		

Due to folding, the generated predicate `r4/3` is now recursive

```
r4([],T,T).
r4([B|C],T,U):- F is B*3, r4(C,[F|T],U).
```

The query `r(X,[],3,Z)` now corresponds to the query `r4(X,[],Z)`.

Prestwich [Pres90] also uses anti-unification to guarantee folding and thus termination, but he uses a special generalization function on atomic goals which has to be specified by the user. Let the generalization function be g and the current goal A. If the goal stack contains a goal A' such that g(A)=g(A'), then A' will be anti-unified with g(A), producing a goal general enough to permit folding. As shown in comparison of partial evaluators, section 6.1.8, with a good choice of generalization function this method works fairly well, but in some instances generalizes too early, creating sub-optimal code. One advantage of our method is that our loop prevention method is capable of considering the size of the arguments.

2.7.3 Guaranteeing that folding will eventually apply

Just doing a single generalized restart will not guarantee that we will get an instance of the query such that folding can be applied. But if no folding occurs, the loop prevention will trigger another generalized restart. There is however no danger that the process of loop prevention and generalized restart will go on indefinitely. The reason is simply that at each generalized restart we generate a strictly more general instance of the previous attempted restart, and this sequence of more and more general queries ends with a query in which all arguments are distinct variables. Any other call to the same predicate will obviously be an instance of this query, and folding can be applied.

Note that it is not essential in the above argument that it is the *least* general generalization of the two queries; *any* more general term will do. By using the anti-unifier we do not generalize unnecessarily, and the restarted query is as specific as possible, thereby facilitating the partial evaluation procedure.

2.8 No recomputation of variable variants

All generated predicates are stored along with the call that prompted their creation. Whenever a predicate call is to be partially evaluated, this database of generated predicates is consulted and the stored definition is used instead if the current call is a variable variant of the stored call.

This optimization is crucial to make it viable to apply the partial evaluator to larger programs.

2.9 Automatic vs. manual partial evaluation

In many other partial evaluators for Prolog explicit annotations for each predicate indicate whether unfolding should be performed. If too few calls are unfolded, poor code is produced, and if too many calls are unfolded there is a danger of infinite unfolding or that the generated program becomes too large.

Thus to provide these annotations requires skill, but once they are given the partial evaluation procedure can become quite fast and simple. For partial evaluation of meta-interpreters the following can be an attractive approach: the expert programmer writes the meta-interpreter together with unfolding annotations, whereas the user provides the object program. Sterling shows that by partial evaluation the overhead of one or several meta-interpreters can be removed [Ster86a]. General strategies for generating unfolding annotations is a current research issue which is discussed in [Lakh90].

There are however techniques to limit the unfolding without annotation by inspecting the goal-stack. For example, Venken [Venk84] simply stops unfolding when the first recursive call is found.

In Mixtus the goal-stack is also used to prevent infinite unfolding, as follows. A goal is only unfolded if it cannot be folded. The loop prevention mechanism together with the generalized restart will guarantee that infinite unfolding cannot happen. Mixtus normally folds whenever

possible, that is, when the current goal is an instance of a goal on the goal-stack. This instance-test is sometimes used as a loop prevention strategy [LeSa88, Take86], although it does not guarantee termination by itself.

For some programs the Mixtus strategy of folding whenever possible will prevent useful unfolding. For example, partial evaluation of the query `p(X)` and the following program will not unfold the goal `p(3)`.

```
p(3).
p(X) :- p(3).
```

An alternative folding strategy is to fold only if the current goal is a variant of a goal in the goal-stack. That strategy would have been much more successful on the above program. If the Mixtus parameter *fold_instance* is unset, folding will not normally be performed if a goal is just an instance. Instead, it is required that the goal is a variant of a goal in the goal-stack. An example of such a program is given in section 6.1.2. However, if a loop is suspected by the loop prevention mechanism, the current goal will be folded even if it is an instance.

As mentioned earlier, Mixtus will never perform a folding of a composite goal, e.g. a conjunction. Many important program transformations as shown in [Tama84, Lakh88] cannot be performed given this restriction. Our initial investigations indicate that folding of composite goals seldom maintains the semantics of full Prolog. We are not aware of any automatic partial evaluator that is capable of folding composite goals.

3 Classification of predicates

All generated and predefined predicates are categorized with respect to the following properties:

- The number of solutions, as computed by the module for determinacy analysis.
- Purity, i.e. the occurrence and character of non-logical predicates such as `var(X)`, `write(X)` etc.
- Whether it contains cut.

The analysis of these properties will be dealt with in the following sections. The use of the properties will be shown in the next chapter, where partial evaluation of full Prolog is described.

3.1 Determinacy analysis

The number of solutions returned by calling a predicate is the most important and most difficult property of a predicate that we attempt to compute. For instance, knowing that a predicate cannot fail, enables the partial evaluator to perform a cut immediately following this predicate call, as in

```
p(X) :- q(X),!.
p(3).
```

If we know that `q(X)` above cannot fail for any instance of `X`, we are sure to reach the cut and the second clause can be eliminated. Furthermore, if it is known that `q(X)` succeeds only once, the cut can be eliminated altogether. As cuts severely complicate and hinder the partial evaluator, knowledge about the number of solutions a predicate may produce is essential.

In general, the number of solutions produced by a goal depends on how it is called (what the arguments are) and whether backtracking will be attempted. By the *number of solutions* we will in the following mean the number of times the goal will succeed if backtracked into. As is the case throughout, it is the procedural behavior of the Prolog program that is of interest when counting the number of solutions. Thus we count the total number of solutions produced by the goal and not the number of distinct solutions.

Besides succeeding or failing, a goal when called may loop. The simplest case is

```
loop :- loop.
```

`loop` will be assigned a "number of solutions" denoted by $\mathcal{L}$. Some goals produce a finite number of solutions and then loop. For instance, calling `loop2` will succeed twice and then loop:

```
loop2.
loop2.
loop2 :- loop. % loop defined as above
```

An exact analysis of a goal with exhaustive backtracking would thus consider the following cases:

0	the goal fails	$\lightning$	the goal loops
1	the goal succeeds once	1'	the goal succeeds once and then loops
2	the goal succeeds twice	2'	the goal succeeds twice and then loops
3	the goal succeeds three times	3'	the goal succeeds three times and then loops
etc.			

We are not interested in the exact number of solutions of a goal, but the example above where cut could be removed, shows that it is fruitful to distinguish between zero, one or many solutions. In the description below, 2 denotes two *or more* solutions and 2' denotes two *or more* solutions and then a loop. 2' is also used to denote the case where an infinite number of solutions is produced. As will be seen, this interpretation of 2' is essential when the number of solutions of recursively defined predicates is to be computed.

The analysis below does not consider the arguments of the goal, and an exact determination cannot be made of the number of solutions. Instead the analysis will apply to any instance of a given goal (and thus to a predicate), yielding a subset of {$\lightning$, 0, 1, 1', 2, 2'}, called a *solution set*, which contains the number of solutions for any instance of the goal.

As the arguments of the goal are irrelevant to the analysis presented below, we can talk about the solution set in general for a predicate and not just for a particular goal.

The following table gives some examples of how the built-in predicates are classified. Most built-in predicates have the solution set {0,1}, and this is also true for most user-defined predicates in a typical Prolog program.

Solution set	*Typical built-in predicates*
{0}	`fail/0, false/0`
{1}	`true/0, write/1, assert/1`
{2'}	`repeat/0`
{0, 1}	`read/1, var/1, =/2, is/2`
{0, 1, 2'}	`length/2`
{0, 1, 2}	`retract/1`
{∠, 0, 1, 2}	`bagof/3, setof/3`

All generated predicates are also analyzed with respect to the number of solutions they produce. The analysis is only based on the classification of the predicates called by the generated predicate, whether they are other previously generated (thus already classified) predicates or built-in predicates.

User-defined predicates are never analyzed by the algorithm; Mixtus will always first partially evaluate them producing "generated" predicates. The reason for this is that all goals are partially evaluated left-to-right in a clause, and only the goals to the left of the current goal are analyzed. For example, the conditions for performing or removing a cut only depend on the goals to the left of the cut.

It might seem odd that the analysis does not consider the variables in the program being analyzed. Normally, this would have been an essential part of the analysis, but as the analysis is only carried out on programs generated by the partial evaluator, these programs have a much simpler structure; the partial evaluator has already been considering the variables and simplified the code considerably. Experience has shown that this simplified analysis is very close to what can be hoped for.

Transformation to a single clause

The definition of a predicate may consist of several clauses, each having a body consisting of a possibly composite goal. To simplify the analysis, all clauses are combined into one as follows.

$p(t_{11},\ldots,t_{1n})$:- b_1.
:
$p(t_{m1},\ldots,t_{mn})$:- b_m.

is transformed into[3]

[3] This transformation is actually not quite correct due to the awkward syntax and semantics of (Test –> Then) (which means exactly the same as (Test –> Then; `false`)).

For example,

```
p(A) :- q(A) –> r(A).
p(A) :- s(A).
```

is not the same as

```
p(A) :- q(A) –> r(A); s(A).
```

$p(X_1,\ldots X_n)$:-

$p(X_1,\ldots X_n)=p(t_{11},\ldots,t_{1n})$, b_1;

:

$p(X_1,\ldots X_n)=p(t_{m1},\ldots,t_{mn})$, b_m.

One optimization essential for the analysis is also performed: if all $t_{i1},\ldots,t_{in}$ are distinct variables, the unification $p(X_1,\ldots X_n)=p(t_{i1},\ldots,t_{in})$ is replaced by "true" and b_i is replaced by $b_i[X_1/t_{i1},\ldots X_n/t_{in}]$. This will replace the unification (with the solution set {0,1}) by "true" (having the solution set {1}) and the outcome of the analysis usually becomes more informative.

As head unification now always succeeds, the number of solutions of the predicate is simply the number of solutions of its body.

Correctness of solution sets

In the following it will be shown for each goal G how a solution set G_{sols} is computed. G_{sols} is *correct* iff for all instances $G\theta$ of G the number of solutions of $G\theta$ is a member of G_{sols}.

Although this definition of correctness makes {∠, 0, 1, 1', 2, 2'} a correct solution set for any G, below it will be shown how a much smaller set specialized for G can be constructed. In those constructions $(A\bullet B)_{sols}$ is computed given A_{sols} and B_{sols}, that is $(A\bullet B)_{sols} = f_\bullet(A_{sols},B_{sols})$, where • is a logical connective such as "," or ";". The functions $f_\bullet$ are always defined so that the following condition holds:

Soundness for solution set functions

A solution set function $f_\bullet$ is *sound* if the values of the function $f_\bullet$ are defined so that $k \in f_\bullet(A_{sols},B_{sols})$ *if* there exist two goals A and B with solution sets A_{sols} and B_{sols} respectively such that there is an instance of A•B with *k* solutions.

Or, in other words, if A_{sols} is a correct solution set for A and B_{sols} is a correct solution set for B, then $f_\bullet(A_{sols},B_{sols})$ is a correct solution set for A•B.

It can be shown that a composition of sound solution set functions always yields a new sound solution set function.

Optimality for solution set functions

A solution set function $f_\bullet$ is *optimal* if the values of the function $f_\bullet$ are defined so that $k \in f_\bullet(A_{sols},B_{sols})$ *only if* there exist two goals A and B with solution sets A_{sols} and B_{sols} respectively such that there is an instance of A•B with *k* solutions.

The elementary solution set functions (for disjunction, conjunction and if-then-else), presented below are optimal in this sense. However, in general it is not guaranteed that a com-

---

This issue is also discussed in section 4.2.2.2.

position of optimal solution set functions remain optimal. Thus, the arguments given here do not guarantee that the computed solution set for a goal A contains *only* those k for which there is an instance of A with k solutions.

3.1.1 Goals without a cut

First it is shown how the solution set is computed for a goal not containing cut, depending on its form.

If G is a call to a built-in or previously generated already analyzed predicate, G_{sols} is simply found in the database describing those predicates. If G is a totally unknown predicate, the worst is assumed and G_{sols}={ℒ, 0, 1, 1', 2, 2'}. Otherwise G must be a composite goal or a recursive call.

Disjunction

For a disjunction, if the number of solutions is known for each of its components, it is fairly straightforward to compute the number of solutions for the disjunction. For instance, if the first part of the disjunction succeeds once (=1) and the second part fails (=0), the number of solutions for the disjunction is 1. If the first part succeeds at least twice (=2) and the second part succeeds once and then loops (=1'), the whole disjunction succeeds at least twice and then loops (=2').

If this argument is systematically carried out for all cases, we get the following table for "s_1+s_2" where s_1 is the number of solutions of the first disjunct and s_2 is the number of solutions for the second.

s_1 \ s_2	ℒ	0	1	1'	2	2'
ℒ	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ
0	ℒ	0	1	1'	2	2'
1	1'	1	2	2'	2	2'
1'	1'	1'	1'	1'	1'	1'
2	2'	2	2	2'	2	2'
2'	2'	2'	2'	2'	2'	2'

s_1+s_2 (a disjunction)

In general, we do not know the number of solutions, but rather have a solution set containing the number of solutions. The function "+" on elements is therefore extended to sets by the following definition

$$f_{;}(S_1,S_2) \equiv \{s_1+s_2 \mid s_1 \in S_1 \wedge s_2 \in S_2\}$$

The solution set for a disjunction then becomes

$$(A;B)_{sols} = f_{;}(A_{sols},B_{sols})$$

The correctness of the above extension follows from the fact that both disjuncts are called with the same substitution.

Conjunction

Unfortunately, in a conjunction, both conjuncts are usually not called with the same substitution. This makes it necessary to use a more complicated method.

The elements in the solution set for a conjunction are found by the following table where $x|y \in S$ is an abbreviation for $x \in S \vee y \in S$.

$0 \in (A,B)_{sols}$ iff $0 \in A_{sols} \vee$ $1\|2 \in A_{sols} \wedge 0 \in B_{sols}$	$\mathcal{L} \in (A,B)_{sols}$ iff $\mathcal{L} \in A_{sols} \vee$ $1\|1'\|2\|2' \in A_{sols} \wedge \mathcal{L} \in B_{sols} \vee$ $1'\|2' \in A_{sols} \wedge 0 \in B_{sols}$
$1 \in (A,B)_{sols}$ iff $1 \in A_{sols} \wedge 1 \in B_{sols} \vee$ $2 \in A_{sols} \wedge 0 \in B_{sols} \wedge 1 \in B_{sols}$	$1' \notin (A,B)_{sols}$ iff $1\|1'\|2\|2' \in A_{sols} \wedge 1' \notin B_{sols} \vee$ $1' \notin A_{sols} \wedge 1 \in B_{sols} \vee$ $2' \notin A_{sols} \wedge 1 \in B_{sols} \wedge 0 \in B_{sols} \vee$ $2\|2' \in A_{sols} \wedge 1 \in B_{sols} \wedge \mathcal{L} \in B_{sols}$
$2 \in (A,B)_{sols}$ iff $2 \in A_{sols} \wedge 1 \in B_{sols} \vee$ $1\|2 \in A_{sols} \wedge 2 \in B_{sols}$	$2' \notin (A,B)_{sols}$ iff $1\|1'\|2\|2' \in A_{sols} \wedge 2' \notin B_{sols} \vee$ $1'\|2' \in A_{sols} \wedge 2 \in B_{sols} \vee$ $2' \notin A_{sols} \wedge 1 \in B_{sols} \vee$ $2' \notin A_{sols} \wedge 1\|2 \in B_{sols} \wedge 0 \in B_{sols} \vee$ $2 \in A_{sols} \wedge 1\|2 \in B_{sols} \wedge \mathcal{L}\|1'\|2' \notin B_{sols}$

The rules above are formed by arguments of the following form:

- (A,B) may fail ($0 \in (A,B)_{sols}$) if and only if either A may fail ($0 \in A_{sols}$) or A may succeed and B may fail ($0 \in B_{sols}$). In the latter case, A must not loop on backtracking ($1|2 \in A_{sols}$).
- (A,B) may loop ($\mathcal{L} \in (A,B)_{sols}$) if and only if either A may loop, A may succeed ($1|1'|2|2' \in A_{sols}$) and B may loop, or A may loops after backtracking and B may fail. In the latter case we also include the special kind of loop caused by A having an infinite number of solutions ($2' \notin A_{sols}$) and B failing. An example of this is the conjunction (`repeat,false`) which loops indefinitely.

The other rules are constructed by similar arguments, and the rules are presented in a non-minimal form to simplify these arguments. In this way a solution set function $f,(A_{sols},B_{sols}) = (A,B)_{sols}$ is defined by the above rules. The above argument has however only suggested how to show that the solution set functions are sound, not that they are optimal, in the sense of the

definition of optimality for solution set functions. What also needs to be done is to find concrete goals for each case above.

Usually this is very simple, as in the first case above where the conjunct fails in two situations. The first situation is when the first conjunct fails and a concrete goal for doing that is "fail". In the second situation the second conjunct fails (being `false`) and the first conjunct succeeds once (being `true`) or twice (being `true;true`).

Sometimes it can be harder to find the concrete goals. For instance, take the second situation when $1 \in (A,B)_{sols}$, that is "$2 \in A_{sols} \wedge 0 \in B_{sols} \wedge 1 \in B_{sols}$". Are there really two goals A, with $A_{sols}=\{2\}$, and B, with $B_{sols} =\{0,1\}$, such that $1 \in (A,B)_{sols}$?

Yes, let A be `(X=a; var(X); nonvar(X); nonvar(X))`. Clearly A will deliver at least two solutions for all substitutions. Then let B be `nonvar(X)` which may either fail or succeed once. The conjunction (A,B) will produce exactly one solution if called with X unbound.

All other cases can similarly be checked to find concrete goals satisfying the conditions, thereby verifying the optimality of the solution set function.

Note that optimality is not needed for the rules to be sound. What we achieve by establishing optimality for the rules is just to show that we cannot leave out any of the sufficient conditions for putting a certain value k into $(A,B)_{sols}$. We have made no attempt to formulate the rules as economically as possible, and some simplifications not changing the meaning are possible.

As a corollary, the table below shows the result of applying the solution set function to singleton sets. It is expected that each produced solution set contains at least one element. In fact, the table shows that exactly one element is produced in all cases, which indicates that the analysis is capable of giving a reasonably informative result.

S_1 \ S_2	{0}	{ℒ}	{1}	{1'}	{2}	{2'}
{0}	{0}	{0}	{0}	{0}	{0}	{0}
{ℒ}	{ℒ}	{ℒ}	{ℒ}	{ℒ}	{ℒ}	{ℒ}
{1}	{0}	{ℒ}	{1}	{1'}	{2}	{2'}
{1'}	{ℒ}	{ℒ}	{1'}	{1'}	{2'}	{2'}
{2}	{0}	{ℒ}	{2}	{1'}	{2}	{2'}
{2'}	{ℒ}	{ℒ}	{2'}	{1'}	{2'}	{2'}

f,(S_1,S_2) for singleton sets

If-then-else

The solution set for an if-then-else construct is a function of the solution set of its components:

$$(\text{Test -> Then; Else})_{sols} =$$
$$(\text{if } \angle \in \text{Test}_{sols} \text{ then } \{\angle\} \text{ else } \{\}) \cup$$
$$(\text{if } 0 \in \text{Test}_{sols} \text{ then Else}_{sols} \text{ else } \{\}) \cup$$
$$(\text{if } 1|1'|2|2' \in \text{Test}_{sols} \text{ then Then}_{sols} \text{ else } \{\})$$

That is, if the test may loop, the whole construct may loop. If the test may fail, the solution set must contain the solution set of the else-part. If the test may succeed, the solution set must contain the solution set of the then-part.

The above rule defines a function $f_{->}$ on solution sets by

$$f_{->}(\text{Test}_{sols},\text{Then}_{sols},\text{Else}_{sols}) = (\text{Test -> Then; Else})_{sols}$$

Backtracking if-then-else

In SICStus Prolog and several other Prolog systems there is also a “backtracking if-then-else” construct. In that construct, written as “if(Test,Then,Else)” in SICStus Prolog, if the Test succeeds, backtracking will be performed into the Test, but the Else part will then never be performed. If the Test fails immediately, Else will be performed as usual. The solution set for backtracking if-then-else is very similar to the non-backtracking form above (the only difference being boxed):

$$\text{if(Test,Then,Else)}_{sols} =$$
$$(\text{if } \angle \in \text{Test}_{sols} \text{ then } \{\angle\} \text{ else } \{\}) \cup$$
$$(\text{if } 0 \in \text{Test}_{sols} \text{ then Else}_{sols} \text{ else } \{\}) \cup$$
$$(\text{if } 1|1'|2|2' \in \text{Test}_{sols} \text{ then } \boxed{(\text{Test,Then})_{sols}} \text{ else } \{\})$$

The corresponding solution set function is

$$f_{if}(\text{Test}_{sols},\text{Then}_{sols},\text{Else}_{sols}) = \text{if(Test,Then,Else)}_{sols}$$

Recursive calls

Finally we have the question of the analysis of recursive predicates. The argument is first carried out for directly recursive predicates. To simplify the notation it is further assumed that the analyzed predicate just contains a single call to itself:

$$p(x_1,\ldots,x_n) \text{ :- } A(p(t_1,\ldots,t_n)).$$

A(g) is a template for the body of the predicate with the recursive call as an argument. The corresponding solution set function for A(g) can be constructed by a composition of solution set functions and is denoted by f_A. The solution set p_{sols} for p is computed by a fix-point iteration starting with $\{\angle\}$, that is

$$p_{sols} \equiv \text{iterate}(\{\angle\})$$

$$\text{where iterate}(S) \equiv \begin{cases} S & \text{if } f_A(S)=S \\ \text{iterate}(f_A(S)) & \text{otherwise} \end{cases}$$

Note that we are not trying to find the least fix-point of f_A; it is always {} and of no interest. It is not obvious that the iteration of f_A starting with $\{\mathcal{L}\}$ will yield any fix point. If $\mathcal{L} \in f_A(\{\mathcal{L}\})$ then by standard results the iteration will yield a fix-point, as it can be shown that f_A is monotonic. However, if $\mathcal{L} \notin f_A(\{\mathcal{L}\})$, there are no general results which ensure that a fix-point will be found. Indeed, if the iteration starts with a different value than $\{\mathcal{L}\}$, there may not exist any fix-point. For example, consider the predicate

```
p :- p -> fail; true.
```

The iteration beginning with {1} will not yield any fix-point:

$f_A(\{1\})=\{0\}$

$f_A(\{0\})=\{1\}$

But starting with $\{\mathcal{L}\}$ will yield the correct fix-point $\{\mathcal{L}\}$ immediately:

$f_A(\{\mathcal{L}\})=\{\mathcal{L}\}$

In order to prove that the iteration starting with $\{\mathcal{L}\}$ will in fact yield a fix-point, and prove at the same time that this fix-point is a correct solution set for p, we will have to describe in somewhat more formal terms what is computed by the function f_A. However, to keep the presentation readable, no greater formality will be introduced than is needed to convince the reader that the details can be filled in.

First some terminology. Given a set of Prolog clauses, we define the corresponding *skeleton clauses* as the expressions obtained by deleting all arguments in the original clauses and replacing every predicate that is not user-defined by its solution set, given by the database. For example, from

```
member(X,L) :- L=[H|R], (X=H; member(X,R)).
special(X) :- X=[a|R], member(a,R).
```

we get

```
member :- {0,1}, ({0,1};member).
special :- {0,1}, member.
```

A skeleton goal is a functor with the arguments deleted. A skeleton execution of a skeleton goal is an ordinary Prolog execution with the difference that whenever a “goal” consisting of a solution set is encountered, any assumption may be made about the behavior of that goal which is consistent with that solution set. Thus, if the solution set is {0,1}, we may assume that the goal when first encountered fails, or we may assume that the goal succeeds and then fails on backtracking.

The solution set of a skeleton goal p is defined as the set of k such that k is the number of solutions of p obtained in one of the, possibly infinitely many, complete executions of p. By a complete execution is meant, as in the case of ordinary Prolog, an execution that exhausts every avenue of backtracking.

OBSERVATIONS

1. Given a set of definitions of Prolog predicates not involving any recursion, the solution sets computed using the rules are precisely the solution sets of the corresponding skeleton goals.
2. Furthermore, the soundness of the computed solution set means that the actual solution set of a predicate, in the sense of the set of k such that k is the number of solutions of some instance of the predicate, is included in the computed solution set of the predicate.

We now want to extend this to the case where a recursively defined predicate is added to the program. For any n, we define a sequence of skeleton clauses as follows:

p_n :- $A_{skel}(p_{n-1})$.
p_{n-1} :- $A_{skel}(p_{n-2})$.
:
p_1 :- A_{skel}(loop).
loop :- loop.

where A_{skel} is the skeleton of A as defined above. By observation 1, $f_A^n(\{\Lambda\})$ is in fact the solution set of the skeleton goal p_n.

LEMMA

For every sufficiently large n, the solution set of p_n is equal to the solution set of the skeleton goal p, where p is defined by

p :- A_{skel}(p).

This lemma both implies that the iteration yields a fix-point, and using the observation 2, that this fix-point is a correct solution set for the original predicate p. So all that is needed is a proof of the lemma.

PROOF OF LEMMA

The definition of p_n can be viewed as unfolding n-1 steps of the recursion in p, but with the looping predicate “loop” called after n-1 recursions. Thus p and p_n behave identically as long as does the depth of the recursive calls to p not exceed n-1. In that case, for every execution of p, we get a corresponding execution of p_n, with the same results.

The first half of the proof is to establish that if there is a complete execution of p with solution number k, then for all sufficiently large n, there is a complete execution of p_n with solution number k. The other half of the proof will establish the converse, which enables us to conclude that the solution sets are the same.

If k=0, we note that a total execution of p which fails can only use a finite depth m of recursive calls to p. Thus any p_n with n>m will have a corresponding total execution. The same argument works for k=1 and k=2.

If k=ℒ, there is a complete execution of p which loops. Either p loops calling another looping predicate, or it calls itself ad infinitum. In either case, the corresponding execution of p_n will loop, for every n.

If k=1', we have a complete execution of p which yields one solution and then loops. For sufficiently large n, the emulation of that execution in executing p_n will yield the first solution. After that, as the emulation continues, there are two possibilities: either the depth of recursion in the original complete execution does not exceed n-1, and so the executions will be identical; or else the execution of p_n will reach the loop, and thus loop anyway.

If k=2', the same argument applies if the complete execution of p produces a finite number of solutions and then loops. However, if the given execution of p produces an infinite number of solutions, we cannot conclude that the corresponding execution of p_n will do the same, since an infinite number of recursions may be necessary to produce the infinitely may solutions. In this case, however, the execution of p_n will eventually loop, and since this behavior is also classified as 2', the conclusion remains correct.

In the other half of the proof, we need to establish that the solution set of p_n is included in that of p for all sufficiently large n. That is, if p_n has an execution with solution number k, then so does p, at least if n is sufficiently large.

If k∈ {0,1,2}, the argument is simple: an execution of p_n with this solution number cannot call the predicate “loop”, so we get a corresponding execution of p by just replacing calls to p_i everywhere by calls to p.

If k=ℒ there is an execution of p_n which loops directly. There are then two possibilities: either there is no call to p_{n-1}, in which case the looping execution carries over directly to p, or there is such a call. In the latter case we know that there is a (partial) execution of p leading to a recursive call. Then there is a complete looping execution of p, since we need only duplicate the execution that led to a recursive call to p whenever p is recursively called.

Next suppose p_n, but not p, has a complete execution with solution number 1'. The part of this execution leading to a solution must be the same as the corresponding execution of p. Then, on backtracking, p_n loops. This must be due to calls to $p_{n'}$ for n'<n, since otherwise the corresponding call to p would also loop. However, there cannot be any call to $p_{n'}$ in the looping

call other than for n'=n-1. For otherwise it would be possible for a call to p to lead to another call to p (without any intervening solution), and in that case there would be a corresponding execution of p with solution number 1'. So, in other words, n=1. Thus, if there is a complete execution with solution number 1' of p_n for n>1, there is such a complete execution of p.

For the case 2', we first note that if a complete execution of p_n yields infinitely many solutions, loop has not been called, so it is a complete execution of p with infinitely many solutions. The case where 2' means a finite number of solutions followed by a loop is dealt with just like the case 1'.

■

Mutually recursive predicates

The problem of finding solution sets for a set of mutually recursive predicates can be solved in a similar way. Instead of a single equation, we get a set of equations, one for each predicate:

$$p^1_{sols} = f_{A^1}(p^1_{sols}\ldots,p^n_{sols}).$$
$$\vdots$$
$$p^n_{sols} = f_{A^n}(p^1_{sols}\ldots,p^n_{sols}).$$

It can be shown that a correct solution set can be found by fix-point iteration on the array $(p^1_{sols}\ldots,p^n_{sols})$, starting with $(\{\not\in\},\ldots,\{\not\in\})$.

Due to the incremental generation of predicates in the partial evaluator, this set of mutually recursive equations is actually never formed in Mixtus. Instead, the not yet generated predicates are classified as $\{\not\in,0,1,1',2,2'\}$.

An Example

It may sometimes be useful to have a predicate that succeeds an infinite number of times. The built-in predicate “repeat” has this behavior, and it could have been defined by

```
repeat.
repeat :- repeat.
```

An analysis will correctly estimate the solution set of “repeat” to {2'} as follows. The predicate is transformed into a single clause using the procedure described above, which first leads to

```
repeat :- (repeat=repeat); (repeat=repeat),repeat.
```

and then after the optimization mentioned we get

```
repeat :- true; true,repeat.
```

Thus the fix-point equation is

$$repeat_{sols} = f_{;}(\{1\},f,(\{1\},repeat_{sols}))$$

the fix-point iteration starts with $repeat_{sols}=\{\not\in\}$:

$f_{;}(\{1\},f,(\{1\},\{\not\subset\})) = f_{;}(\{1\},\{\not\subset\}) = \{1'\}$

The next iteration starts with {1'}

$f_{;}(\{1\},f,(\{1\},\{1'\})) = f_{;}(\{1\},\{1'\}) = \{2'\}$

which is shown to be the derived fix-point by one more iteration

$f_{;}(\{1\},f,(\{1\},\{2'\})) = f_{;}(\{1\},\{2'\}) = \{2'\}$

Another Example

One more example will illustrate the technique of finding a solution set.

```
zero([H|R]) :- H==0 -> true; zero(R).
```

First the predicate is transformed so that the head unification becomes explicit:

```
zero(X) :- X=[H|R], (H==0 -> true; zero(R)).
```

The explicit unification cannot be removed, and since `=/2` is a built-in predicate it is found that $(u=w)_{sols}=\{0,1\}$. Similarly it is found that $(u==w)_{sols}=\{0,1\}$. The fix-point equation then becomes

$zero_{sols} = f,(\{0,1\}, f_{->}(\{0,1\},\{1\},zero_{sols}))$

The fix-point is found by starting the iteration with $zero_{sols} = \{\not\subset\}$, and we get

$f,(\{0,1\}, f_{->}(\{0,1\},\{1\},\{\not\subset\})) = f,(\{0,1\}, \{\}\cup\{\not\subset\}\cup\{1\}) =$

$f,(\{0,1\},\{\not\subset,1\}) = \{\not\subset,0,1\}$

One more loop in the iteration gives

$f,(\{0,1\}, f_{->}(\{0,1\},\{1\},\{\not\subset,0,1\})) = f,(\{0,1\}, \{\not\subset,0,1\}) = \{\not\subset,0,1\}$

and the fix-point is found to be $\{\not\subset,0,1\}$.

The result is in this case is the best possible: zero([]) fails, zero([0]) succeeds once and zero(X) loops.

More Examples

The following table gives the result of the algorithm for some simple predicates.

defining clause	*number of solutions*
`h.`	{1}
`h :- true; true.`	{2}
`h :- true; true; true.`	{2}
`h :- h.`	{$\mathcal{L}$}
`h :- true;h.`	{2'}
`h :- true; (h->false;true).`	{1}
`h :- true; true; loop.`	{2'}
`h(X) :- var(X), h(X).`	{$\mathcal{L}$,0}
`h(X) :- var(X); h(X).`	{$\mathcal{L}$,1',2'}

3.1.2 Goals with cut

We now turn to the treatment of goals containing cut. Each solution set element "x" is now duplicated with one element "x_n" denoting that no cut in the goal has been executed during an exhaustive search, and one element "x_c" denoting that cut has been executed. The new solution sets are now subsets of the universal solution set

$$\mathcal{U} = \{\mathcal{L}_n, 0_n, 1_n, 1'_n, 2_n, 2'_n, \mathcal{L}_c, 0_c, 1_c, 1'_c, 2_c, 2'_c\}$$

As before, the solution set for a goal depends on its form. Also as before, if the goal is a call to a built-in or previously generated predicate the solution set is found in the database. The special built-in predicate "!" (cut) has the solution set $\{1_c\}$.

Also as before, predicates being analyzed are transformed into a single clause. But as cut does not propagate through a procedure call it is not correct to let the solution set for a predicate simply be the solution set for its body. Each solution set element denoting that a cut has been executed must be converted into its cut-free equivalent to give a correct solution set. For example, if the body has the solution set $\{0_n,\mathcal{L}_n,1_c\}$, the predicate has the solution set $\{0_n,\mathcal{L}_n,1_n\}$.

For composite goals the functions $f_,$, $f_;$, $f_{->}$ and f_{if} have to be extended to handle the new elements.

Disjunction

For disjunctions, the extended function $f_;$ is defined using the non-cut version of + by the following (where the subscript i is either n or c):

$$\begin{cases} a_c + b_i = a_c \\ a_n + b_c = (a+b)_c \\ a_n + b_n = (a+b)_n \end{cases}$$

The first case, a_c+b_i, needs an explanation. If the first part of the disjunction performs a cut in the course of an exhaustive search, the second part will never be executed, and does not contribute to the number of solutions.

As in the cut-free case, the solution sets for a disjunction are defined by

$$(A;B)_{sols} = f_{;}(A_{sols},B_{sols})$$

where

$$f_{;}(S_1,S_2) \equiv \{s_1+s_2 \mid s_1 \in S_1 \wedge s_2 \in S_2\}$$

Conjunction

For a conjunction the analysis is considerably more complex. As in the cut-free analysis, conditions are given for when a certain solution set element belongs to the solution set of the conjunction, as shown by the following table.

$0_n \in (A,B)_{sols}$ iff $0_n \in A_{sols} \vee$ $1_n \mid 2_n \in A_{sols} \wedge 0_n \in B_{sols}$	$\mathcal{L}_n \in (A,B)_{sols}$ iff $\mathcal{L}_n \in A_{sols} \vee$ $1_i \mid 1'_i \mid 2_i \mid 2'_i \in A_{sols} \wedge \mathcal{L}_n \in B_{sols} \vee$ $1' \mid 2'_n \in A_{sols} \wedge 0_n \in B_{sols}$
$1_n \in (A,B)_{sols}$ iff $1_n \in A_{sols} \wedge 1_n \in B_{sols} \vee$ $2_n \in A_{sols} \wedge 0_n \in B_{sols} \wedge 1_n \in B_{sols}$	$1' \notin (A,B)_{sols}$ iff $1_i \mid 1'_i \mid 2_i \mid 2'_i \in A_{sols} \wedge 1' \notin B_{sols} \vee$ $1' \notin A_{sols} \wedge 1_n \in B_{sols} \vee$ $2' \notin A_{sols} \wedge 1_n \in B_{sols} \wedge 0_n \in B_{sols} \vee$ $2_i \mid 2'_i \in A_{sols} \wedge 1_n \in B_{sols} \wedge \mathcal{L}_n \in B_{sols}$
$2_n \in (A,B)_{sols}$ iff $2_n \in A_{sols} \wedge 1_n \in B_{sols} \vee$ $1_n \mid 2_n \in A_{sols} \wedge 2_n \in B_{sols}$	$2' \notin (A,B)_{sols}$ iff $1_i \mid 1'_i \mid 2_i \mid 2'_i \in A_{sols} \wedge 2' \notin B_{sols} \vee$ $1' \mid 2'_n \in A_{sols} \wedge 2_n \in B_{sols} \vee$ $2' \notin A_{sols} \wedge 1_n \in B_{sols} \vee$ $2' \notin A_{sols} \wedge 1_n \mid 2_n \in B_{sols} \wedge 0_n \in B_{sols} \vee$ $2_i \in A_{sols} \wedge 1_n \mid 2_n \in B_{sols} \wedge \mathcal{L}_n \mid 1'_n \mid 2'_n \in B_{sols}$
$0_c \in (A,B)_{sols}$ iff $0_c \in A_{sols} \vee$ $1_n \mid 2_n \mid 1'_i \mid 2'_i \in A_{sols} \wedge 0_c \in B_{sols} \vee$ $1_c \mid 2_c \in A_{sols} \wedge 0_i \in B_{sols}$	$\mathcal{L}_c \in (A,B)_{sols}$ iff $\mathcal{L}_c \in A_{sols} \vee$ $1_i \mid 1'_i \mid 2_i \mid 2'_i \in A_{sols} \wedge \mathcal{L}_c \in B_{sols} \vee$ $1_c \mid 1'_c \mid 2_c \mid 2'_c \in A_{sols} \wedge \mathcal{L}_n \in B_{sols} \vee$ $1' \mid 2'_c \in A_{sols} \wedge 0_n \in B_{sols}$
$1_c \in (A,B)_{sols}$ iff $1_c \in A_{sols} \wedge 1_n \in B_{sols} \vee$ $1_i \mid 1'_i \mid 2_i \mid 2'_i \in A_{sols} \wedge 1_c \in B_{sols} \vee$ $2_i \mid 2'_i \in A_{sols} \wedge 0_n \in B_{sols} \wedge 1_c \in B_{sols} \vee$ $2_c \in A_{sols} \wedge 0_n \in B_{sols} \wedge 1_n \in B_{sols}$	$1' \notin (A,B)_{sols}$ iff $1_i \mid 1'_i \mid 2_i \mid 2'_i \in A_{sols} \wedge 1' \notin B_{sols} \vee$ $1_c \mid 1'_c \mid 2_c \mid 2'_c \in A_{sols} \wedge 1' \notin B_{sols} \vee$ $1' \notin A_{sols} \wedge 1_n \in B_{sols} \vee$ $2' \notin A_{sols} \wedge 1_n \in B_{sols} \wedge 0_n \in B_{sols} \vee$ $2_i \mid 2'_i \in A_{sols} \wedge 1_n \in B_{sols} \wedge \mathcal{L}_c \in B_{sols} \vee$ $2_c \mid 2'_c \in A_{sols} \wedge 1_n \in B_{sols} \wedge \mathcal{L}_n \in B_{sols}$
$2_c \in (A,B)_{sols}$ iff $2_c \in A_{sols} \wedge 1_n \in B_{sols} \vee$ $1_c \mid 2_c \in A_{sols} \wedge 2_n \in B_{sols} \vee$ $1_i \mid 1'_i \mid 2_i \mid 2'_i \in A_{sols} \wedge 2' \notin B_{sols}$	$2' \notin (A,B)_{sols}$ iff $1_i \mid 1'_i \mid 2_i \mid 2'_i \in A_{sols} \wedge 2_c \in B_{sols} \vee$ $1_c \mid 1'_c \mid 2_c \mid 2'_c \in A_{sols} \wedge 2' \notin B_{sols} \vee$ $1' \mid 2'_c \in A_{sols} \wedge 2_n \in B_{sols} \vee$ $2' \notin A_{sols} \wedge 1_n \in B_{sols} \vee$ $2_c \in A_{sols} \wedge 1_n \in B_{sols} \wedge 1' \notin B_{sols} \vee$ $2_i \in A_{sols} \wedge 1_n \in B_{sols} \wedge 1' \notin B_{sols} \vee$ $2' \notin A_{sols} \wedge 1_n \mid 2_n \in B_{sols} \wedge 0_n \in B_{sols} \vee$ $2_c \mid 2'_c \in A_{sols} \wedge 1_n \mid 2_n \in B_{sols} \wedge \mathcal{L}_n \in B_{sols} \vee$ $2_i \mid 2'_i \in A_{sols} \wedge 1_n \mid 2_n \in B_{sols} \wedge \mathcal{L}_c \in B_{sols}$

Some typical arguments showing how the rules in the table have been constructed will be given. For example, as can be seen from the table, the conjunct may loop without ever performing a cut (that is, $\lightning_n \in (A,B)_{sols}$) in three cases.

The first case is when the first conjunct loops directly without performing a cut ($\lightning_n \in A_{sols}$).

The second case is when the first disjunct succeeds at least once ($1_i|1'_i|2_i|2'_i \in A_{sols}$), and the second disjunct loops without performing a cut ($\lightning_n \in B_{sols}$). Note that in this case the first disjunct may be of the form $1_c|1'_c|2_c|2'_c$ as the cut may be performed on backtracking, and not necessarily when the first solution is produced.

The third case is when the first conjunct succeeds at least once and then loops or generates infinitely many solutions ($1'_n|2'_n \in A_{sols}$) and the second disjunct fails ($0_n \in B_{sols}$) forcing the first disjunct to loop or backtrack indefinitely.

The reader may convince himself that the other rules in the table are correct by similar arguments.

If-then-else

For if-then-else-constructs, the test-part is assumed not to perform any cut[4]. The functions for computing the solution set are therefore essentially unchanged:

$$
\begin{aligned}
&(\text{Test -> Then; Else})_{sols} = \\
&\quad (\text{if } \lightning_n \in \text{Test}_{sols} \text{ then } \{\lightning_n\} \text{ else } \{\}) \cup \\
&\quad (\text{if } 0_n \in \text{Test}_{sols} \text{ then Else}_{sols} \text{ else } \{\}) \cup \\
&\quad (\text{if } 1_n|1'_n|2_n|2'_n \in \text{Test}_{sols} \text{ then Then}_{sols} \text{ else } \{\})
\end{aligned}
$$

Backtracking if-then-else

$$
\begin{aligned}
&\text{if(Test,Then,Else)}_{sols} = \\
&\quad (\text{if } \lightning_n \in \text{Test}_{sols} \text{ then } \{\lightning_n\} \text{ else } \{\}) \cup \\
&\quad (\text{if } 0_n \in \text{Test}_{sols} \text{ then Else}_{sols} \text{ else } \{\}) \cup \\
&\quad (\text{if } 1_n|1'_n|2_n|2'_n \in \text{Test}_{sols} \text{ then (Test,Then)}_{sols} \text{ else } \{\})
\end{aligned}
$$

[4] The “proper” semantics of a cut in the Test-part of an if-then-else construct is sometimes debated. Some suggestions have been:

- it is simply forbidden, and the program may crash if it is used anyway.
- it will cut outside the Test like an ordinary cut, and in particular remove the Else-part.
- it will have no effect outside the Test, but within the Test.

All three behaviors have been observed in the same Prolog system (earlier versions of SICStus) for compiled, emulated and interpreted code respectively. In later versions of SICStus, a check is made to ensure that a cut does not occur within the Test-part, and this will be assumed in the following analysis.

Recursive calls

Recursive predicates are treated analogously to the cut-free case, with a similar justification. For each new fix-point iteration, all "cut" elements are replaced by their cut-free equivalences as cuts do not propagate through procedure calls.

Example

Let us analyze a variant of `member/2` having a cut.

```
fmember(A,[A|T]) :- !.
fmember(A,[H|T]) :- fmember(A,T).
```

The transformation into one clause produces

```
fmember(X1,X2):- fmember(X1,X2)=fmember(A,[A|T]),!;
        fmember(X1,X2)=fmember(A,[H|T]), fmember(A,T).
```

We are looking for the solution set for `fmember/2` which is the solution set of its body.

fmember_{sols} =

(`fmember(X1,X2)=fmember(A,[A|T]),!`

`; fmember(X1,X2)=fmember(A,[H|T]), fmember(A,T)`)$_{sols}$ =

$f_{;}(f,(\{0_n,1_n\}, \{1_c\}), f,(\{0_n,1_n\}, \text{fmember}_{sols}))$

Starting the iteration with $\text{fmember}_{sols} = \{\mathcal{L}_n\}$ gives

$f_{;}(f,(\{0_n,1_n\}, \{1_c\}), f,(\{0_n,1_n\}, \{\mathcal{L}_n\})) =$

$f_{;}(\{0_n,1_c\}, \{0_n,\mathcal{L}_n\}) = \{0_n,\mathcal{L}_n,1_c\}$

A new iteration is started with the cut-free equivalent $\{0_n,\mathcal{L}_n,1_n\}$:

$f_{;}(f,(\{0_n,1_n\}, \{1_c\}), f,(\{0_n,1_n\}, \{0_n,\mathcal{L}_n,1_n\})) =$

$f_{;}(\{0_n,1_c\}, \{0_n,\mathcal{L}_n,1_n\}) = \{0_n,\mathcal{L}_n,1_n,1_c\}$

which, after replacement of 1_c by its cut-free equivalent 1_n, becomes $\{0_n,\mathcal{L}_n,1_n\}$ and a fix-point has been found.

Assuming that the Prolog system is capable of handling cyclic structures, this result is optimal in the sense that there are concrete instances of the goal that was analyzed which exhibit the three behaviors derived. That is, the three cases are

0_n: for `fmember(1,[]))`,
$\mathcal{L}_n$: for `fmember(1,[0,0,…])`,
where an infinite list of zeros can be created by `X=[0|X]`
1_n: for `fmember(1,[1])`

If the Prolog system incorporates an occur check in the unification, cyclic structures will never be created and the looping behavior will not occur. Although the analysis in this case is not optimal, it still correct. In general it is only guaranteed that the analysis is correct and not that it is optimal, so this result is not very surprising.

Use of solution sets

As already mentioned, there are two properties concerning the number of solutions of a goal that are of interest in the partial evaluator and they are defined as follows

$$\text{successful}(G) \equiv 0_n \notin G_{sols} \wedge 0_c \notin G_{sols}$$

$$\text{deterministic}(G) \equiv 2_n \notin G_{sols} \wedge 2' \notin G_{sols} \wedge 2_c \notin G_{sols} \wedge 2' \notin G_{sols}$$

These tests are particularly important in the partial evaluation of cut, but they are also used for `findall/3` and `bagof/3`.

3.2 Predicates with cut

It is very important to know whether a generated predicate contains a cut, as this prevents unfolding of calls to the predicate. It is a very simple syntactic test to check this, but some care has to be taken for some of the built-in meta-logical predicates. A cut in a `call/1`, `setof/3`, `bagof/3`, or a `findall/3` has a scope which is limited to the meta-predicate, and is not counted as a cut in this predicate. In contrast, a cut imbedded in `( –> ; )` or `if/3` must be counted, as its scope is the whole generated predicate. A small database in the partial evaluator contains information about the meta-predicates and what scope they give to a cut in one of their arguments.

```
% mc means meta-argument where cut is significant
% m means meta-argument where cut is not significant
% n means non-meta-argument
pred_property((m->mc;mc)).
pred_property(if(m,mc,mc)).
pred_property(call(m)).
pred_property(findall(n,m,n)).
% etc.
```

3.3 Purity of predicates

To efficiently handle partial evaluation of full Prolog, the built-in, user-defined and generated predicates must be classified according to their “purity”. A few predicates are so intimately dependent on the original code that they cannot be reasonably handled at all. For example, the predicate `ancestors/1` returns the current goal-stack as a list.

The predicates that are handled are classified into three groups: *side-effect*, *propagation sensitive* and *logical*. This classification is similar to Lakhotia’s in [Lakh89a].

A predicate P is considered to belong to the group *side-effect* if the relation P,`false`≤`false` does not hold (as defined in section 1.6). This means that for non-looping goals P in some program where ”P,`false`” is replaced by `false`, the new program behaves differently. As expected, `write/1`, `read/1`, `assert/1`, `retract/1` and “`!`” all belong to this group. The

predicates `var/1`, `nonvar/1`, `=../2` and `=/2` do not belong to this group as there are no programs that behave differently when this replacement is done.

How should we classify the predicate `loop`, defined by the clause `loop:-loop`.? Clearly `loop,false` behaves differently than `false` but `loop` is not considered a side-effect predicate. This goes back to the definition of the relation G≤G' between goals, which essentially allows G' to be anything if G loops. Similarly, the built-in predicate `repeat/0` is not a side effect predicate, although `repeat,false` never terminates.

A predicate P not having a side-effect is *propagation sensitive* if the relation P,X=*t*≤X=*t*,P does not hold for some variable X and term *t*. For instance, `X=3,var(X)` is clearly not the same as `var(X),X=3`. A very large group of built-in predicates belong to this category such as `var/1`, `nonvar/1`, `==/2`, `\==/2`, `arg/3` and `functor/3`.

The remaining predicates are *logical*. Here we find all the purely logical predicates such as `=/2`, `=../2`, `repeat/0`, `loop/0` (as defined above) and `</2`.

The following table summarizes these properties of predicates.

	P,`false` ≤ `false`	P,X=t ≤ X=t,P
logical	yes	yes
propagation sensitive	yes	no
side-effect	no	yes/no

The partial evaluator will use the classification of predicates to move a failure or a unification as far to the left as possible.

If possible, a predicate should be classified as logical, as the partial evaluator generates much better code in that case. The predicate `<` actually violates the test for being logical as `X<3,X=1` will fail, generating an error message, whereas `X=1,X<3` succeeds. Nevertheless, the predicate is classified as logical, as this difference is noticeable only for an incorrect program generating an error message, and as mentioned in section 1.6, it is assumed that the program given to the partial evaluator is correct. A similar argument is possible for `=../2`.

Unfortunately, `arg/3` cannot be classified as logical since `arg(X,f(a),Y),X=1` fails without an error message, whereas `X=1,arg(X,f(a),Y)` succeeds. From a partial evaluation point of view it would have been preferable if `arg/3` had written an error message if the first argument is not instantiated. It would probably also help the user find an error at an early stage.

As a general optimization, a propagation sensitive predicate becomes logical when it is ground, i.e. when it does not contain any variables. This is easily seen from the definition of logical.

When in doubt how to classify a predicate, it is always safe to classify it as “side-effect”. In particular, unknown user predicates are always considered side-effect predicates. This makes it

possible to run the partial evaluator on one file at a time, thereby reducing the complexity of the partial evaluation procedure. The resulting program generated by the partial evaluator may not be as efficient, but it will always be correct, i.e. the resulting program will produce at least the same results as the original program.

3.3.1 Classification of generated predicates

All generated predicates are classified on the basis of the classification of the predicates they call.

A generated predicate containing a call to a predicate that has a side-effect (other than cut) is always classified as having a side-effect itself. Otherwise, if the predicate contains a call to a propagation sensitive predicate or a cut it will be classified as propagation sensitive. Finally, if the predicate only calls logical predicates, it is logical itself.

It may seem odd that a predicate containing a cut, which is considered to have a side-effect, can be classified as propagation sensitive. The reason is that the scope of the cut does not reach outside the predicate. The cut however generally makes the predicate propagation sensitive, as in

```
p(a) :- !.
p(X).
```

With this definition, `b=X,p(X)` succeeds and `p(X),b=X` fails .

There is a case where a predicate containing cut may even be logical: only tests (predicates which do not bind anything) precede the cut, as in

```
p(X,Y) :- X<3, !, q(Y).
p(X,Y) :- X>=3, r(Y).
```

The current partial evaluator does not incorporate this refinement.

3.3.2 Preserving loops

Normally Mixtus will transform a looping predicate followed by a failure just to a failure. The user may set the flag *preserve_loops* which prevents this transformation. As the loop is a desirable side-effect to be kept, all predicates p which can loop or backtrack indefinitely (i.e. $\mathcal{L}_n|1'_n|2'_n \in p_{sols}$) are considered to be side-effect predicates, even if they do not call any side-effect predicates.

4 Partial evaluation of full Prolog

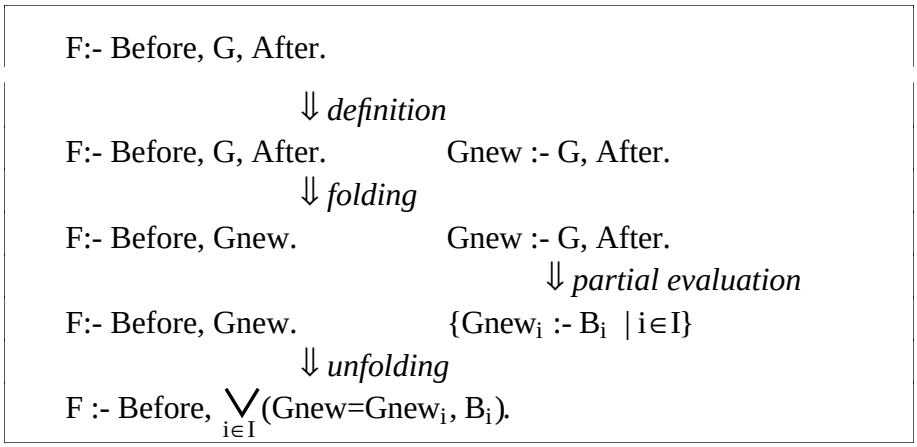
In partial evaluation for full Prolog, the conditions for folding and loop prevention are identical to those for pure Prolog. The unfolding transformations are here modified to facilitate propagation of bindings, while retaining full Prolog semantics. Partial evaluation of a number of built-in predicates, cut in particular, is discussed.

4.1 The basic mechanism for full Prolog

4.1.1 Making every goal in a clause the first

Recall that there are two unfolding transformations for Prolog: one where the unfolded goal is first in the clause, and one where it isn't. Only the former transformation performs any propagation of bindings. Below, a transformation is introduced which will guarantee that every goal temporarily becomes the first, and thereby enables propagation of bindings. This is done by generating a new clause where the goal is the first in the body and perform partial evaluation on that clause. To maintain the semantics of cut the clauses produced by partial evaluation must however be unfolded back again.

Another advantage of this transformation is a uniform treatment of unfolding for built-in and user-defined predicates. During partial evaluation some built-in "evaluable" predicates are executed, and this execution might be non-deterministic. This transformation will ensure that such an execution is handled properly, and that the solutions in the resulting program will be returned in the correct order. The following schema gives an overview of the transformations involved and is explained in the next paragraph.



Making the goal G *first in a clause for partial evaluation*

Given a goal G which is to be partially evaluated in a clause “F:- Before, G, After” the clause is replaced by “F :- Before,Gnew” where Gnew is a call to a new temporary predicate defined by a single clause “Gnew:-G,After”. This transformation may be seen as a “definition” followed by a folding transformation. The clause head Gnew has an arbitrary predicate name, and the arguments are the variables in “G,After”. As a refinement, using variable trimming as described in section 2.4.2, variables not occurring in “F:-Before” do not have to be included in Gnew.

Partial evaluation may then safely continue with the goal G in the newly generated clause “Gnew:-G,After” as it is the first goal. In this process, no problem arises if G is non-deterministic or propagates variables left, as it is the first goal in the clause.

The final result of partial evaluation of “Gnew:-G,After” is a set of new clauses, “$\{Gnew_i :- B_i \mid i \in I\}$”.

When the partial evaluator is recursively invoked, the goal-stack is not extended with the clause “Gnew:-G,After”. This will ensure that folding will not apply, that is, no recursive predicate for Gnew will be formed, and thus unfolding can always be applied. To maintain the semantics of potential cuts in “G,After”, it is always necessary to perform unfolding for the generated predicate Gnew. Of course, as before, when G is partially evaluated, a new recursive predicate may be generated.

In [LeSa88], where a partial evaluator for full Prolog is presented, a similar technique seems to be used for creating a predicate. The predicate is however only unfolded if it is defined by a single clause.

Some optimizations are always carried out to improve performance. If the disjunction is empty it is replaced by `false`. If the “Before” goal is a side-effect goal, this explicit `false` goal must be kept as it is. Otherwise, the “Before” part is a logical or propagation sensitive goal, as defined in section 3.3, and the empty set of clauses is returned. If “Before” was a looping predicate, this

transformation may transform a looping program into a non-looping program, thereby making a difference in semantics between the original program and the new one, as discussed in 1.6.

Example: aliasing

It is tempting to incorrectly let the clause

```
p(X,Y) :- var(X), Y=2.
```

be "partially evaluated" into

```
p(X,2) :- var(X).
```

as `Y` does not occur in a non-logical goal. Mixtus will however not let the binding of `Y` pass left over `var(X)`. A call which aliases `X` and `Y`, such as `p(Z,Z)`, would make the two clauses above behave differently.

4.1.1.1 Reevaluation to facilitate right-propagation

One important optimization is performed when unfolding for full Prolog. The goals formed by unfolding are reevaluated to facilitate propagation of bindings to the right. In pure Prolog this is not necessary, since the bindings propagate freely left and right.

An example: Let `p/1` be defined by `p(X):-var(X),X=3`. Unfolding of `p(X)` in `q(X):-p(X),r(X)` gives

```
q(X) :- var(X), X=3, r(X).
```

For full Prolog a reevaluation after unfolding is made which propagates the binding `X=3`

```
q(X) :- var(X), X=3, r(3).
```

This example shows that the reevaluation is essential as instead of just having `r(X)` we now have `r(3)`, which is more instantiated, thereby facilitating further partial evaluation.

4.1.1.2 Problem: generated recursive predicates are not unfolded

Mixtus will not unfold if a recursive predicate has been generated. This has been found to be a very good strategy, but there are cases where unfolding also of recursive predicates is desirable to allow the right-propagation of bindings. In this example `q(X)` is recursive, so it is not unfolded, although an unfolding would have found the contradiction `X=1` and `X=2`.

```
p(X):-q(X), X=2.
q(X):-(r(X) -> q(X); s(X)), X=1.
```

Unfolding is particularly desirable for trivial predicates, i.e. predicates defined with just one clause. Unfortunately, unfolding a recursive predicate will break the loop prevention mechanism of Mixtus. An alternative solution to this problem is to allow propagation of bindings from goals

which are not unfolded. Fortunately, situations similar to the one above seem to be uncommon, and Mixtus does not incorporate any special strategy for handling them.

4.1.1.3 Unfolding and generalized restart

For predicates produced by the generalized restart mechanism (as described in section 2.7) one precaution has to be taken to avoid loops. The code for these predicates has been produced with a more general instance of the call than the one in the original program in order to increase the probability of folding. If folding did occur, there is no problem, since there is no unfolding and thereby no recomputation will be made. The problem is when no folding occurs, a rare but possible situation. When the call to the predicate is unfolded, the goals encountered are not stored in the database of already partially evaluated goals, and a full partial evaluation will be performed for these goals. This might lead to further generalized restarts and unfoldings, which can make the partial evaluator loop. Mixtus prevents these loops by never unfolding predicates generated by a generalized restart.

4.1.2 Left propagation of bindings

For a logical goal we may propagate the bindings left. As the goal then becomes further instantiated, the partial evaluator is restarted on that goal.

For example, `p(X,Y,Z):-append(X,Y,Z),X=[1,2]` may then be partially evaluated into `p([1,2],Y,[1,2|Y])`.

Some care however has to be taken when reevaluating goals: we must ensure that the partial evaluator does not return to the same goal indefinitely with more and more instantiated arguments. This can easily be arranged by maintaining a list of all retried goals. In Mixtus each goal may be reevaluated at most *maxpropleft* times, where *maxpropleft* is a Mixtus parameter.

4.1.2.1 Using anti-unification to extract common bindings

As shown above, a call Gnew to a set of clauses $Gnew_i$:-B_i can be transformed into a disjunction "$\vee_i$ (Gnew=$Gnew_i$, B_i)". By applying anti-unification to the heads of the clauses, we can extract all common bindings. Let $Gnew_{anti}$ be the anti-unification of all $Gnew_i$. We can then get the following transformation instead

$$Gnew=Gnew_{anti}, \vee_i (Gnew_{anti}=Gnew_i, B_i)$$

As before, the unifications are simplified as much as possible.

Using this transformation means that some bindings only have to be made once, and not for each disjunct. But more important is the fact that left propagation may now be applied.

An example

An example will illustrate the idea. The predicate `member/2`, used below, is defined in appendix B. In the clause

```
h(A) :- member(A,L), p(L).
```

the call `member(A,L)` cannot be further partially evaluated, so partial evaluation continues with the second goal `p(L)` which is defined by

```
p([1]).
p([2]).
```

If anti-unification is not used, `p(L)` will be replaced by a disjunction

```
h(A) :- member(A,L), (p2(L)=p2([1]); p2(L)=p2([2])).
```

and the unifications can be simplified into

```
h(A) :- member(A,L), (L=[1]; L=[2]).
```

but no further optimizations are possible.

If anti-unification is used instead, we get

```
h(A) :- member(A,L), p2(L)=p2([X]),
        ((p2([X])=p2([1]); p2([X])=p2([2])).
```

which simplified becomes

```
h(A) :- member(A,L), L=[X], (X=1; X=2).
```

Now it is possible to use left-propagation of the single binding

```
h(A) :- member(A,[X]), (X=1; X=2).
```

Partial evaluation is restarted on `member(A,[X])`, and that goal can be eliminated by setting `A=X`.

```
h(A) :- (A=1;A=2).
```

Finally the disjunction, now the first goal, is unfolded.

```
h(1).
h(2).
```

4.1.3 Handling cuts

The cut is a very important built-in predicate that must be fully supported in any partial evaluator for full Prolog. The most important usage of cut is perhaps the implementation of negation as failure (see section 4.2.1.1). The first observation we make is that we cannot unfold a predicate containing a cut after partial evaluation, as the scope of the cut would then be changed[5]. Thus we now have three conditions that prevent unfolding: that the predicate is recursive, created

[5] Some predicate calls may be unfolded even if the predicate called contains a cut. This is possible if the call occurs in the last clause, and all calls preceding it in the clause are deterministic. It is also possible to unfold if the call is preceded by a cut and all goals between the cut and the call are deterministic. This generalizes an observation made by [Owen89]. For simplicity, Mixtus 0.3 never tries to unfold predicates containing cuts.

by a generalized restart, or contains cut. Since unfolding is necessary to propagate bindings further, it is crucial that partial evaluation is able to remove as many cuts as possible.

Also notice that in the procedure for temporarily generating new predicates shown above, these new predicates are never kept but always unfolded back using a disjunction. This will ensure that the scope of a cut remains unchanged. If the disjunction formed just consists of a single disjunct and a cut has been executed in the corresponding clause, the disjunct itself is handled as if it were a cut. Possibly that cut may also be executed.

Venken in [Venk84, Venk88] suggests that a mark should be used that limits the scope of the cut when a predicate containing a cut is unfolded. The main disadvantage of that suggestion is that most Prolog systems do not have a mechanism to limit the scope of cut.

The problem of how to handle cut is here split up into three parts: how to cut, when to cut, and when to remove cut.

4.1.3.1 How to cut

Conceptually, this is very simple: just discard the clauses after the clause with the cut that is to be executed.

4.1.3.2 When to cut

A cut may be executed during partial evaluation only if we are sure to reach the cut for all calls to this clause. This means that it must not be possible for the head unification to fail and all goals preceding the cut must not fail in this clause. The first condition is satisfied if all arguments of the head are distinct variables, since all possible instantiations of calls to the predicate will succeed in the head unification. Below, it is shown how this condition can be relaxed. The second condition, that the goals preceding the cut must not fail, is checked using the module for determinacy analysis described in section 3.1.

These conditions for executing cut generalize the conditions given in [Bugl89, Lakh90, Owen89] as they do not have a module for determinacy analysis.

4.1.3.3 When to remove cut

A cut may be removed if it is totally unnecessary, that is, it occurs in the last clause, and all calls preceding it in the clause do not produce any choice-points. The same module for determinacy analysis is also used for the latter analysis. A cut in the first part of a disjunction cannot be removed, but possibly a cut in the second part. Cuts in both the “then”-part and the “else”-part of an “if-then-else”-construct may be removed.

Further improvements may be possible if a cut has been removed in the last clause. The partial evaluator is therefore invoked again on the clause, producing a set of clauses. If that set is empty, the last clause disappears, the clause that preceded it becomes last, and the procedure for removing a cut is restarted again to handle this new last clause.

There are other cuts that could be removed: cuts not in the last clause, but in a clause where there already is a cut. The second of two cuts separated only by calls to deterministic predicates may safely be removed. The current partial evaluator does not incorporate this refinement as it cannot be crucial for making a predicate totally free from cuts, which is a necessary condition for unfolding. Also, such situations are very rare, so the improvement would not have been worthwhile.

An example

Recall that for each goal that is not the first in a clause, a new predicate is temporarily created as shown in section 4.1.1. When the newly created predicate is partially evaluated, a cut may be executed there even though it could not have been executed in the original clause. As the following example will show, a cut within a disjunction may trim away the rest of the disjuncts without trimming away the rest of the clauses. Let the query be `p(7,Z)` and the program

```
p(X,Y) :- nonvar(Y), (r(X), !, Y=0; Y=1).
p(X,X).
r(7).
```

When partially evaluating the temporarily generated predicate corresponding to the disjunction above, it is found that the cut may be executed. However, when looking at the whole clause, it is not certain that the cut will be reached as `nonvar(Y)` may fail. The resulting query is `p2(Z)` and the returned program is

```
p2(Y) :- nonvar(Y), !, Y=0.
p2(7).
```

Another example

Consider the following predicate, meant to compute the maximum of two integers.

```
max(X,Y,Y) :- X<Y, !.
max(X,Y,X).
```

Partial evaluation of the query `max(2,1,A)` will as expected return a new query `max2(A)` defined by the clause

```
max2(2).
```

But if the query `max(1,2,A)` is given instead, `max2(A)` will have the following definition

```
max2(2) :- !.
max2(1).
```

This is perhaps somewhat surprising, but an inspection of `max/3` reveals that this is the best the partial evaluator can do; the call `max(1,2,1)` should succeed according to the definition! One

might argue that the code for `max/3` is incorrect[6], but many real Prolog programs are written in this way, with the tacit assumption that some variables are always unbound when the predicate is called.

4.1.4 Keeping track of unbound variables: input variables

To generate efficient code, the partial evaluator must sometimes use information about which variables are unbound. Experience has shown that it is much easier to keep track of which variables may be bound; the other variables must be unbound. The variables that may be bound are called the “input variables”.

This classification of variables is done dynamically as the partial evaluator executes, and a list is kept of which of the currently encountered variables are input variables. Initially, all variables in the top-level query are normally input variables. All variables in a term unified with an input variable are also classified as input variables. Hence, some variables in the head of a clause may be classified as input variables. For a goal in a clause, the input variables are all the input variables in the head of the clause plus all the variables in the preceding goals in the clause. Note that all non-input variables in that goal are guaranteed to be unbound; they cannot even be bound to another variable.

The input variables are mainly used to relax the conditions for executing a cut. As will be shown below, the input variables can also be used to delay some unifications. Finally, the input variables can improve the treatment of some built-in predicates such as `var/1`, `copy_term/2` and `findall/3`.

4.1.5 Relaxing the conditions for executing a cut

When we start to partially evaluate a clause, the input variables in the head of the clause are identified. Let us call these variables the “head input variables”. Recall that the first condition for executing a cut was that all the arguments in the head must be distinct variables, as the unification might fail otherwise. This may now be relaxed to the condition that all head input variables must be distinct variables. The other head variables are guaranteed to be unbound at the time of the call, and a later unification performed in the body of the clause can therefore never fail. As only the head input variables may possibly cause a unification failure, the condition that the variables must be different is restricted to these variables. Bugliesi and Russo also have a similar method of allowing the non-input variables to become bound, and still perform a cut [Bugl89].

---

[6] The “correct” code is of course

```
max(X,Y,Z) :- X<Y, !, Y=Z.
max(X,Y,X).
```

To get a correct treatment of folding, the goal-stack has to be extended to include the input variables of each goal. Folding cannot be applied unless the input variables of a goal are a subset of the input variables of the goal on the goal-stack. The generalized restart mechanism therefore also has to be extended to guarantee that folding eventually will be possible. The simplest way to do this is to let all variables in the newly generated goal be classified as input variables.

In the example above for `max/3`, it was unclear whether the third argument could be instantiated. In the following clause, it is quite clear that the third argument of `max/3` is not an input argument, and in partial evaluation the query `twice_max(1,2,X)` can be fully evaluated to `X=4`.

```
twice_max(A,B,D) :- max(A,B,C), D is 2*C.
```

4.1.6 Delaying output unifications

Using the knowledge about the head input variables it is sometimes possible to move some data structures in the head into the body of the generated clause. This might be advantageous as the predicate may then fail before that structure is created. For example, assume that the second argument in the head of the following clause does not correspond to a head input variable

```
p(X,f(7,X)) :- X<3.
```

It will then be transformed into

```
p(X,Y) :- X<3, Y=f(7,X).
```

The rules for the transformation are as follows. Each head argument t_i not corresponding to a head input variable is replaced by a new variable X_i. An explicit unification $X_i=t_i$ is moved into the body past all deterministic calls. It will not be moved past a non-deterministic call as there is then a danger of having the unification executed several times. Also, the explicit unification must not be moved beyond the last call, unless it is a built-in predicate. The reason for this is that tail-recursion optimization might otherwise be lost.

This optimization, named “delayed output unification”, has been found particularly useful when a Definite Clause Grammar is used to produce a list, as the tests are done before the structures are created. Mixtus has a parameter named *delayoutput* for turning on this optimization.

4.2 Built-in predicates for full Prolog

The simplest method for handling built-in predicates has already been mentioned in section 2.2.3: conditions are given in a database that indicate when it is safe to evaluate a predicate. Some non-logical predicates can be handled this way:

```
evaluable(atom(X)) :- nonvar(X).
evaluable(integer(X)) :- nonvar(X).
evaluable(var(X)) :- nonvar(X).
```

Even better results may be obtained if knowledge about the "input variables" (`Ivars`) is used. Let the second argument of `evaluablel/2` be the list of input variables. The conditions for evaluation of `var/1` may be improved into

```
evaluablel(var(X),Ivars) :- nonvar(X).
evaluablel(var(X),Ivars) :- \+ (member(V,Ivars), V==X).
```

If the simple method above is not appropriate then there are two main ways for supporting built-in predicates in Mixtus:

- The predicate can be defined in Prolog, and the partial evaluator will be given that code.
- The predicate is handled by special code.

Examples will be given in the following of these two methods, with emphasis on the latter.

4.2.1 Built-in predicates defined in Prolog

A distinction must be made between predicates that can be defined in Prolog, and those that in a specific Prolog implementation actually are defined in Prolog. For performance reasons, i.e. execution speed and memory consumption, most Prolog-definable predicates are usually nevertheless handled by specialized code. In a partial evaluator, on the other hand, the main criterion for using a definition in Prolog is that it can be successfully partially evaluated. This in turn seems to be related to the complexity of the definition; a simple definition is more easily handled by the partial evaluator. A more complex definition may "hide" important properties of the predicate, which the partial evaluator may not be able to reveal. Making the behavior of a built-in predicate explicit is called reification [Lakh90].

4.2.1.1 Negation, `\+`

The predicate `\+/1` which implements negation as failure can be defined as

```
\+ X :- X, !, false.
\+ X.
```

Fortunately, this definition is simple enough to be handled successfully by Mixtus. Thus there is no need for special treatment of negation as failure in Mixtus.

An example

Let `p/2` be defined by

```
p(A,L) :- \+ member(A,L).
```

The query `p(X,[1,2,3])` is then partially evaluated into the new query `p2(X)` having the definition

```
p2(1) :- !, false.
p2(2) :- !, false.
p2(3) :- !, false.
p2(_).
```

As each element in the list `[1,2,3]` now corresponds to a separate clause, indexing has been made possible.

The semantics of negation as failure are not changed, because Mixtus preserves the full procedural semantics of the program. A weaker condition for preserving the semantics of negation as failure is that the finite failure set be maintained. Seki [Seki89] shows what modifications to the unfold/fold rules for pure Prolog [Tama84] are necessary to accomplish this. The partial evaluator by Prestwich [Pres90] is based on Seki's rules and is capable of handling negation as failure correctly.

4.2.1.2 'C'/3

The predicate `'C'/3` is usually not called directly, but calls to it are produced when a Definite Clause Grammar is converted into Prolog. The definition of `'C'/3` is

```
'C'([A|B],A,B).
```

4.2.2 If-then-else

The if-then-else-construct is built from `->/2` and `;/2`, that is "(test-part –> then–part; else–part)". Syntactically this is a disjunction, whose first disjunct is `->/2`. For the partial evaluator, this overloaded usage of `;/2` is a complication, because it destroys otherwise natural properties of `;/2` such as associativity.

To avoid code explosion, a goal following an if-then-else construct is not moved into the construct. That is, "(Test->Then;Else),G" is not transformed into "(Test->Then,G; Else,G)". Although applying this transformation would enable propagation of bindings into G, the size of the code generated may explode due to the duplication of G, especially if G also invokes an if-then-else-construct. Avoiding this transformation is quite similar to the rule that unfolding is not applied to generated predicates containing a cut since the arrow (->) may be considered as a cut with a limited scope.

If, for some reason, the user of Mixtus anyway wants to move a goal following the if-then-else construct into the construct, this can be accomplished by setting the parameter "maxlevel" to a value larger than zero, which is the default. The value of the parameter determines how many times this transformation is allowed in a single clause.

In SICStus Prolog, the test-part is not allowed to contain any cut, whereas the then- and else-parts may well contain cuts.

Partial evaluation of if-then-else is done by *transforming* it to a form better suited for partial evaluation, after which the *partial evaluation* is performed, and finally it is *retransformed* into if-then-else form. This three-step procedure will continue to be marked in italics below. Two examples will illustrate the approach before the general method is presented.

An example

By "replacing" `->` with `'!if'` the composite goal

```
(X=[A]; X=[A,B]) -> rev(X,Y), p(Y);
X=Y.
```

is *transformed* into

```
(X=[A]; X=[A,B]), '!if', rev(X,Y), p(Y);
X=Y
```

This when *partially evaluated*, having `'!if'` treated as a cut, becomes

```
X=[A], '!if', Y=[A], p([A]);
X=[A,B], '!if', Y=[B,A], p([B,A]);
X=Y
```

A *retransformation*, as described below, will replace `'!if'` and finally produce

```
X=[A] -> Y=[A], p([A]);
X=[A,B] -> Y=[B,A], p([B,A]);
X=Y
```

Notice that the bindings from the tests have been propagated into the then-part, although the test was non-deterministic.

Another example

A slight modification of the above example will trigger a more complex retransformation using a flag variable. The goal

```
nonvar(X), (X=[A]; X=[A,B]) ->
    rev(X,Y), p(Y)
;   X=Y
```

is first *transformed* into

```
nonvar(X), (X=[A]; X=[A,B]), '!if',
    rev(X,Y), p(Y)
;   X=Y
```

As `nonvar(X)` is propagation sensitive, *partial evaluation* will not propagate the bindings of `X` left, `nonvar(X)` is followed by a disjunction:

```
nonvar(X), (X=[A], '!if', Y=[A], p([A]);
            X=[A,B], '!if', Y=[B,A], p([B,A]))
;   X=Y
```

When *retransforming* it is not possible to "replace" `'!if'` by `->` as the scope of the test would not incorporate `nonvar(X)`. This problem is solved by using a flag `F` that will be set to indicate which case was taken.

```
(nonvar(X), (X=[A], F=1;
             X=[A,B], F=2)) ->  (F=1 -> Y=[A], p([A]);
                                 F=2 -> Y=[B,A], p([B,A]);
                                 true)

;   X=Y
```

Although the above expression may look very complex, the transformation can lead to a very efficient program, as the bindings from the non-deterministic test-parts are propagated to different then-parts, thereby enabling further partial evaluation.

4.2.2.1 How to handle if-then-else in general

Below, the general rules for the transformation and retransformation will be given, but first it is necessary to define what is meant by the goals *following* a cut, as this term is used below. The goals G following a cut in a composite expression C are exactly the goals that could be executed after the cut has been executed.

The *transformation* is very simple: "`(`test-part `->` then-part`;` else-part)" is transformed into "`(`test-part`,'!if',`then-part`;` else-part)".

Then *partial evaluation* of the transformed expression is performed, where `'!if'` is executed when an ordinary cut is.

In this process, `'!if'` may be duplicated (if the test-part is non-deterministic) or possibly eliminated (if it fails). Unlike an ordinary cut, `'!if'` is never explicitly removed during partial evaluation, because each occurrence signals a potential way of succeeding in the test, and this information is used in the retransformation.

The *retransformation* of T is done as follows. Let the result of the partial evaluation of the if-then-else construct be T.

If T is a disjunction, each disjunct T_i is retransformed into T_i' separately and the result is a new if-then-else construct T' whose form is $T\ '_1;\ldots;T\ '_n$ If T is not a disjunction, then n is 1.

Each single disjunct T_i is retransformed as follows. If T_i does not contain any `'!if'` then T_i' is identical to T_i. If T_i just contains one `'!if'`, let the sequence of goals *following* `'!if'` be

called A. Let T_i with the indicated occurrence of A and `'!if'` removed be written as $T_i\langle A\rangle$. The new disjunct T_i' then is

$T_i\langle$true$\rangle$ `->` A

If T_i contains more than one `'!if'`, the retransformation is more difficult. Let each `'!if'` and its following goals be called A_j. This is written as $T_i\langle A_1,\ldots A_n\rangle$. The retransformed disjunct T_i' then is

$T_i\langle$`X=1`,...,`X=n`$\rangle$ -> (`X=1 ->` A_1;...;`X=n ->`A_n; `true`)

The Prolog variable `X` works as a flag, telling which one of the `'!if'` that was taken.

The above retransformation may be slightly enhanced by the following optimizations. If all A_j are identical, the transformation above becomes

$T_i\langle$`true`,...,`true`$\rangle$ ->A_1

In the final disjunction formed T' (=T_1';...;T_n'), if T_n' is of the form (A->B), it is replaced by (A,B) provided that A cannot create any choice-points. If A can produce choice-points, T_n' is not changed, but the call `false` is added as T_{n+1}' so that an if-then-construct (i.e. without else) is not formed.

If T_1' in T' is not of the form (A `->`B), then partial evaluation is restarted on T', as there are further possibilities for improvement, this being a disjunction and no longer an if-then-else-construct.

4.2.2.2 The awkward syntax of if-then-else

In SICStus Prolog it is possible to have a goal of just the form “Test `->`Then”. This means exactly the same thing as “Test `->`Then; `false`”[7]. Goals of the form “Test `->`Then” present certain difficulties, for example the associativity of the disjunction operator “;” is destroyed as

```
(p -> q); (r ; s)
```

is not the same as

```
((p ->q); r); s
```

This problem is avoided if goals of the form “Test `->`Then” are immediately transformed into “Test `->`Then; `false`”.

Unfortunately this transformation does not completely solve the following problem. There are situations where “`true,` Goal” is not the same as “Goal”, due to the syntax of if-then-else. The expression “`true,` (Test `->`Then);Else” does not mean the same thing as “(Test `->`Then);Else”. The former expression is an ordinary disjunction whereas the latter is an if-then-else-construct.

[7] An often suggested and more natural semantics for “Test `->`Then” would be “Test `->`Then; `true`”. This change would however not solve the syntactic problems of the if-then-else-construct.

Thanks to the transformation of "Test `->`Then" described above Mixtus will never confuse the two expressions, but it may perform transformations that lead indirectly to confusion. For example, a goal of the form "`true, G; B`" is partially evaluated into "`G; B`". Subsequent partial evaluation may lead to `G` becoming instantiated to "Test `->`Then", thereby effectively converting the disjunction into an if-then-else-construct[8]. At present, Mixtus does not detect this situation, thereby potentially generating incorrect code. This does however seem never to happen in practice. One reason for this is that most variable goals `G` are, as shown below, transformed into `call(G)`, thereby preventing the situation described from occurring. The price seems too high, in execution performance and in the complexity of the transformations, to extend Mixtus to handle all these situations correctly. Instead the user is cautioned to use the construct "Test `->`Then" with care.

The most straightforward way to "correct" the syntax is to introduce a ternary predicate `ifthenelse/3`. Alternatively, the if-then-else construct could be written with a vertical bar (|) instead of a semi-colon, which would eliminate the double use of the semi-colon, which is the cause of all these syntactic problems.

4.2.3 Meta-calls

The potential performance improvements due to partial evaluation are particularly evident in meta-programming. It is common for such programs to use meta-predicates such as `call/1` and `findall/1`, and these predicates must be fully supported by the partial evaluator. In this section we show how `call/1` and a variable as a goal are handled. Other meta-predicates are handled in the following sections.

4.2.3.1 Call/1

A goal of the form `call(`G`)` is handled in much the same way as a composite goal G, that is, the partial evaluator will return a disjunction G'. Since a cut does not propagate outside the call, some cuts may be removed as described in section 4.1.4.3 above.

If a cut remains in G', the result is `call(`G'`)` since the scope of the cut must not be changed. The call must also be kept if G' contains a variable goal, as that goal may potentially be bound to a cut. If there is no cut or variable goal, G' is handled exactly like any composite goal and partially evaluated again together with the goals that follow it to facilitate propagation of bindings.

4.2.3.2 A variable as a goal

A Prolog program may have a variable as a goal as in

[8] As shown in the next section, most (but not all) variable goals `G` are fortunately converted into call(`G`), thereby often preventing this confusion.

```
p(A,X) :- q(A), X.
```

A problem arises when `X` is bound to a term containing a cut. For a Prolog interpreter the most natural thing is that the scope of that cut is the entire predicate, but for a Prolog compiler much better code can be generated if it is known that `X` can not contain a cut. The solution adopted by SICStus and several other Prolog systems is to constrain the scope of a possible cut by transforming the above clause when read in to

```
p(A,X) :- q(A), call(X).
```

The general rule is that all variable goals `X` in a conjunction, disjunction, or if-then-else-construct are changed into `call(X)` when a clause is read in.

Unfortunately for the partial evaluator, arguments of other meta-predicates such as `findall/3`, `freeze/2` and `call/1` are not transformed in this way. So `X` in

```
p(A,X) :- q(A), call((A=1,X; A=2)).
```

is not changed into `call(X)`.

Thus the partial evaluator must maintain a distinction between a variable goal `X` and `call(X)`.

If the program obtained from partial evaluation contains a call to an unbound variable, the whole original program is potentially needed. In these cases, a warning is issued by Mixtus.

4.2.4 Other built-in predicates

Several common built-in predicates need special treatment to be handled efficiently.

4.2.4.1 Arithmetic predicates

The binary infix predicates `is/2`, `=\=/2`, `=:=/2`, `</2`, `>/2`, `=</2` and `>=/2` compute or compare the value of arithmetic expressions, where such an expression may be composed of the following term constructors:

`+`, `-`, `*`, `/`, `//`, `mod`, `integer`, `float`, `/\`, `\/`, `^`, `\`, `<<`, `>>`, and `[X]`

The partial evaluator first tries to simplify the expressions by recursively simplifying the subexpressions and then the whole expression. Typical simplification rules are

$x * 1 \rightarrow x$

$x * 0.0 \rightarrow 0.0$

$x + 0.0 \rightarrow x$

More advanced methods for simplification are possible, but this method seems adequate for most purposes. If the simplified result becomes a number, the test predicate can be executed by the

partial evaluator. Otherwise, the result of partial evaluation is the predicate with its simplified arguments.

SICStus Prolog handles both integers and floating point numbers for all operations, and this causes some problems. The type of the result may depend on the type of the arguments to the operation, and if one of the arguments is unknown the resulting type cannot be computed. For the operations +, – and * the result is a float only if one of the arguments is a float. For example, the expression `X*0` cannot be simplified into `0`, as the result is `0.0` if `X` is a float. The user may optionally turn on an optimization parameter (*arithopt*) enabling this transformation anyway, if it is guaranteed that the type of the result is of no importance. Adding some type analysis to Mixtus could possibly improve the generated code without turning on this optimization switch.

One more optimization is made for `is/2`. If the first argument of `is/2` is a void variable, i.e. a variable with a single occurrence in the clause, the predicate call may be removed altogether.

4.2.4.2 copy_term/2

The call `copy_term(A,B)` is partially evaluated as follows. First it is checked whether A and B are unifiable. If this unification fails, the result will be a failure.

Otherwise, without unifying A and B, the distinct input variables in A are identified, and named $I_1,\ldots,I_n$. A copy of A is created called A2. The corresponding variables in A2 are named $I'_1,\ldots,I'_n$During partial evaluation we cannot know the run-time value of $I_1,\ldots,I_n$, so they must be copied, and the residue of `copy_term(A,B)` is

`copy_term(t(`$I_1,\ldots,I_n$`), t(`$I'_1,\ldots,I'_n$`)), A2=B`

Notice that it is not possible to split up the above call into a sequence of calls

`copy_term(`I_1`,`I'_1`),…,copy_term(`I_n`,`I'_n`), A2=B`

as this would not correctly handle the possible aliasing that might occur between the variables I_i at run-time.

This scheme can be improved as follows. Let the substitution resulting from unifying A2 with B be θ. If `t(`$I_1,\ldots,I_n$`)` and `t(`$I'_1,\ldots,I'_n$`)`θ are identical (this happens for the call `copy_term(X,X)` for example), the residue can be removed altogether. Furthermore, if n=1, the term constructor `t` can be removed, since its only function was to group several variables in the same term.

It is very important to handle `copy_term/2` carefully because partial evaluation of `findall/3`, `bagof/3` and `setof/3` will generate calls to `copy_term/2`, as will be shown below.

4.2.4.3 Predicates exploiting the standard total ordering of terms

A standard total ordering of all Prolog terms can be defined as follows [Carl88]:

- Variables, in a standard order (roughly, oldest first – the order is *not* related to the names of variables).
- Integers, in numeric order.
- Floats, in numeric order.
- Atoms, in alphabetical (i.e. ASCII) order.
- Compound terms, ordered first by arity, then by the name of the principal functor, then by the arguments (in left-to-right order).

The predicates exploiting this standard total ordering are

```
@</2, @=</2, @>/2, @>=/2, ==/2, \==/2, compare/3, sort/2,
keysort/2, setof/3
```

For instance, when two different unbound variables `X` and `Y` are compared, either `X@<Y` or `Y@<X` is true, depending on the storage address of the variables. In SICStus Prolog the variables are globalized (put on the heap) so that the comparison will return the same result as long as the variables are unbound. Nevertheless, the storage address depends on implementation details and is very hard to predict, and many simple program transformations will change the storage address. For instance,

```
?- X=X, Y=Y, X@<Y.
```

succeeds, whereas

```
?- Y=Y, X=X, X@<Y.
```

fails in SICStus Prolog. Our partial evaluator may well reorder variables and thus invert the result of a comparison between two unbound variables. This may seem a totally unacceptable result of a program transformation, but in most programs where a term comparison is made, it is only used to create an *arbitrary* ordering of terms. Thus, these programs will work equally well, regardless of where the variables are allocated. Assuming that an arbitrary ordering of terms is acceptable, we can define partial evaluation of the predicates dependent on the standard total ordering as shown below.

The tests `@</2, @=</2, @>/2, @>=/2, ==/2` *and* `\==/2`

A comparison “s op s'” (where op is one of `@</2`, `@=</2`, `@>/2`, `@>=/2`, `==/2` or `\==/2`) is partially evaluated as follows.

There are two auxiliary variables: l and l', which are initialized to empty lists. At the end of the processing, “$t(l_1,\ldots,l_n)$ op $t(l'_1,\ldots,l'_n)$” will be equivalent to “s op s'”. The terms s and s' are scanned in a way that preserves the total ordering, i.e. if their functors are identical, the arguments are scanned left to right. Pairs of identical subterms are always skipped, regardless of whether they are variables, atoms or compound terms. If a subterm of s is found to be a variable, it will then be appended to l and the corresponding subterm of s' will be appended to l'. Similarly

if a subterm of s' is found to be a variable. The comparison will then continue with the other subterms, until two subterms which are guaranteed to be different for all instances are encountered. That is, none of the above cases is applicable: the two subterms are not identical, do not share the same principal functor and none of them is a variable. In that case the comparison ends by appending these subterms to the lists l and l' respectively.

As a first result of partially evaluating "s op s'" we get "t(l_1,…,l_n) op t(l'_1,…,l'_n)". For instance, using the above algorithm we get the following results:

```
g(7,X,3)@<g(7,Y,4) becomes t(X,3)@<t(Y,4)
[1,2,3,4] @=< [X,Y,Z,4] becomes t(1,2,3)@=<t(X,Y,Z)
f(g(X,1),2,h(3))==f(g(X,1),Y,h(8)) becomes t(2,3)==t(Y,8)
```

Although correct, for `==/2` and `\==/2` this result is clearly unsatisfactory. It can be improved if neither l_n nor l'_n is a variable. In that case the result of partial evaluation is the value of "l_n op l'_n". For example, `t(2,3)==t(Y,8)` can be further simplified into `false`.

For the other comparisons some slight improvements have been incorporated in Mixtus. The test `t(X,3)@<t(Y,4)` can be simplified into `t(X)@=<t(Y)`, because `3@<4`. The table below summarizes the simplifications for `@<` and `@=<` that can be performed if neither l_n nor l'_n is a variable. The comparisons `@>` and `@>=` are treated similarly.

op	l_n compared to l'_n	op_{new}
@<	@<	@=<
	@>	@<
@=<	@<	@=<
	@>	@<

Situations when "`t(`l_1,…,l_n`)` op `t(`l'_1,…,l'_n`)`" can be replaced by "`t(`l_1,…,l_{n-1}`)` op_{new} `t(`l'_1,…,l'_{n-1}`)`"

If n=1 then further improvements are possible as the functor t is not needed and the residue is simply "l_1 op l'_1". For n=0 the residue is "`t` op `t`", which is further simplified into `true` for `@=<`, `@>=` and `==`, and into `false` for `@<`, `@>` and `\==`.

Finally, as only input variables can be bound during execution, the above algorithm is refined into considering just these as "variables". Non-input variables can then be compared at partial evaluation time, like constants and compound terms.

4.2.4.4 compare/3

A call to `compare(Op,Term1,Term2)` is partially evaluated as follows. If the first argument is bound to `=`, `<` or `>` then the call corresponds to comparing the remaining two arguments with `==`, `@<` and `@>` respectively. If `Op` is unbound then `Term1` and `Term2` are

compared as described in section 4.2.4.3. The result is a pair of lists, l and l', both of length n. If n=1 and neither l_1 nor l'_1 is a variable, the value of `Op` is given by comparing l_1 and l'_1. If n=0 then `Op` is `=`.

Otherwise the residue call becomes `compare(Op,t(`l_1`,...,`l_n`),t(`l'_1`,...,`l'_n`))`.

4.2.4.5 sort/2 and keysort/2

To handle `sort/2` and `keysort/2` the concept *comparable terms* is introduced. Two terms s and s' are *comparable* if exactly one of (s==s')θ, (s@<s')θ or (s@>s')θ holds for all substitutions θ of input variables.

A simple way to to find out if two terms s and s' are comparable is to partially evaluate `compare(Op,`s,s'`)`. If the residue is `true` or `false`, the terms are comparable.

A call `sort(L1,L2)` is evaluable if `L1` is a list where each pair of elements are comparable. Similarly `keysort(L1,L2)` is evaluable if `L1` is a list of items of the form `Key-Value` where for each pair of `Key` values they are comparable.

The definition of `setof/3` which uses `sort/1` is shown below. `bagof/3` turns out to be dependent on `keysort/2` and is also discussed below.

4.2.4.6 findall/3

A call `findall(X,Goal,List)` binds `List` to the list of instances of `X` for which the `Goal` succeeds. The order of the elements in the list should be the same as the order in which the proofs for `Goal` have been found. In most Prolog implementations of `findall/3`, the instances of `X` are recorded by asserting them and finally collecting them into the list `List`. Since Mixtus will always leave all `assert`s and `retract`s as they are, not much would be gained by partially evaluating the Prolog code for `findall/3`. Instead a special procedure will be employed using pseudo-asserts, which have simpler properties.

1. First the composite goal `(Goal,' PSEUDO ASSERT'(X))` will be partially evaluated. The added goal `'PSEUDO ASSERT'(X)` will be treated as an ordinary assert, i.e. it will never be executed or eliminated but may be further instantiated. The result of this partial evaluation is `Goal2`.

2. Then the calls to `'PSEUDO ASSERT'/1` will be extracted from `Goal2` so that the various instances of `X` can be collected in the list `List`. In this process a new goal `Goal3` is formed. It may however happen that the extraction fails. In that case the result of partial evaluation is `findall(X,Goalpe,List)`, where `Goalpe` is the result of partial evaluation of `Goal`. The extraction process is a bit complex and will be discussed in detail below.

3. `Goal3` is further optimized by a partial evaluation, producing goal `Goal4` which is the final residue. However, if `Goal4` contains a cut, its scope must be limited and the final residue becomes `call(Goal4)`.

An example

Before describing the extraction process, an example of partial evaluation is given. In the call `findall(X,(X=1;X=2),List)`, we identify `Goal` as `(X=1;X=2)`. Partial evaluation of

```
(X=1;X=2), 'PSEUDO ASSERT'(X)
```

will make `Goal2` equal to

```
X=1,'PSEUDO ASSERT'(1); X=2,'PSEUDO ASSERT'(2)
```

We will return below to the exact form of the goal `Goal3` produced by the extraction process, as it is a bit complicated. The form of `Goal4`, which is the result of partial evaluation of `Goal3`, depends on whether `X` is an input variable or not. For simplicity, in this example we will only consider the cases where `X` is an unbound variable at partial evaluation time.

If `X` is a non-input variable, we know that it cannot be bound during execution and the unifications in `Goal2` cannot fail. The residue in this case becomes

```
List=[1,2]
```

On the other hand, if `X` is an input variable, the unifications with `X` may fail, thereby removing an element from List. If the unifications succeed, they must not bind `X` permanently. A final residue which has these properties is

```
copy_term(X,X1), copy_term(X,X2),
(X1=1 -> List=[1|L2]; List=L2),
(X2=2 -> L2=[2]; L2=[])
```

Although this code looks a bit complex, it executes about ten times faster than the original `findall`-call.

In [Lakh90] partial evaluation of `findall/3` is discussed, but the call to `findall/3` is never totally removed. As seen from the above example, keeping `findall/3` may induce a large overhead.

Extraction of `'PSEUDO ASSERT'/1`:
Version 1: Ground, successful and deterministic goals

A transformation system is used to describe the extraction of `'PSEUDO ASSERT'/1` for constructing the `List` and `Goal3`.

The transformation that is to be “proven” is

$$\texttt{Goal2} \ \Rightarrow_{\texttt{List}=[]} \ \texttt{Goal3} \qquad \text{[v.1]}$$

where `Goal2` is given and `Goal3` is to be found. We first introduce a simplified transformation system by assuming that `Goal2` does not contain any variables at all. The system has one axiom and two rules.

$$\texttt{'PSEUDO ASSERT'(X)} \ \Rightarrow_{\texttt{L}=\texttt{L1}} \ \texttt{L=[X|L1]} \qquad \text{[PA v.1]}$$

That is, a goal is returned which adds the solution `X` to the list.

$$\frac{\texttt{one\_solution}(G_1) \qquad G_2 \ {}_{L}\Rightarrow_{L1}\ G_2'}{(G_1,G_2) \ {}_{L}\Rightarrow_{L1}\ (G_1,G_2')} \qquad \text{[conj. v.1]}$$

The test `one_solution(`G_1`)` is satisfied if the determinacy analysis finds that the goal is both *successful* and *deterministic*. The goal has to be *deterministic* as otherwise the solutions returned by G_2 must be duplicated once for each solution of G_1. Normally this restriction is not very severe since G_1 cannot be an explicit disjunction; partial evaluation always transforms "(A;B),C" into "(A,C); (B,C)". So all interesting cases where G_1 has a known finite number of solutions can be handled by the transformations. The condition that G_1 must be *successful* will guarantee that the solution recorded by G_2 will actually be reached. In the next version of the transformation system, this condition will be removed from the rules, and checked at run-time instead.

$$\frac{G_1 \ {}_{L}\Rightarrow_{L1}\ G_1' \qquad G_2 \ {}_{L1}\Rightarrow_{L2}\ G_2'}{(G_1;G_2) \ {}_{L}\Rightarrow_{L2}\ (G_1',G_2')} \qquad \text{[disj. v.1]}$$

This rule will convert a disjunction into a conjunction, so that all goals will be executed without backtracking. The lists are maintained so that `L` will contain the sum of solutions found in G_1 and G_2 plus the solutions in `L2`.

More rules can be added to support other constructs in Prolog, but these are the only rules implemented in Mixtus.

An example

Below, a "derivation" for

```
write(1),'PSEUDO ASSERT'(1); write(2),'PSEUDO ASSERT'(2)
```

is shown. In order to make the derivation fit the page, some abbreviations have been used.

$$\frac{\dfrac{\text{one\_sol(wr(1))} \quad \text{'PA'(1)} \ {}_{L}\Rightarrow_{L1}\ \text{L=[1|L1]}}{\text{wr(1), 'PA'(1)} \ {}_{L1}\Rightarrow \text{wr(1), L=[1|L1]}} \qquad \dfrac{\text{one\_sol(wr(2))} \quad \text{'PA'(2)} \ {}_{L1}\Rightarrow_{[]}\ \text{L=[2|[]]}}{\text{wr(2), 'PA'(2)} \ {}_{L1}\Rightarrow_{[]}\ \text{wr(2), L=[2|[]]}}}{\text{(wr(1),'PA'(1); wr(2),'PA'(2))} \ {}_{L}\Rightarrow_{[]}\ \text{wr(1), L=[1|L1], wr(2), L1=[2|[]]}}$$

That is, the result of the extraction is

```
write(1),L=[1|L1],write(2),L1=[2|[]]
```

Version 2: Non-ground, successful and deterministic goals

The restriction that all goals must be ground can be lifted if the goals in the different or-branches are copied so that bindings of variables in one branch do not affect the other branches. The transformation of disjunctions is therefore changed so that `copy_term/2` is applied on one

of the disjunctive goals. The variable `L` which collects the number of solutions also occurs in the copied goal but must not be copied. This is accomplished by unifying `L` with its copy as shown below. This is also done for `L1`.

$$\frac{G_1 \ {}_{L}\Rightarrow_{L1} \ G_1' \qquad G_2 \ {}_{L1}\Rightarrow_{L2} \ G_2'}{(G_1;G_2) \ {}_{L}\Rightarrow_{L2} (\texttt{copy\_term(t(}G_1\texttt{',L,L1),t(}G_1^c\texttt{,L,L1)),}G_1^c\texttt{,}G_2\texttt{')}} \quad \text{[disj. v.2]}$$

Subsequent partial evaluation of the produced goal will usually simplify the above expression considerably, and frequently remove `copy_term` altogether. Thus a large part of the burden of handling `findall/3` efficiently is moved to the treatment of `copy_term/2`, which checks for non-input variables etc.

In a disjunction only the first subgoal is copied. To ensure that also the top-level goal is copied an artificial disjunctive `false` is added to the top-level transformation

$$\texttt{(Goal2; false)} \ {}_{\texttt{List}}\Rightarrow_{\texttt{[]}} \ \texttt{Goal3} \qquad \text{[v.2]}$$

and a rule for `false` is added

$$\texttt{false} \ {}_{L}\Rightarrow_{L1} \ \texttt{L=L1} \qquad \text{[false v.2]}$$

Version 3: Non-ground and deterministic goals

Finally the restriction to successful goals will be removed. This is done by letting the transformation system return a pair a goals $\langle G^t,G^f\rangle$, so that G^t is called if all goals in this branch have succeeded and G^f otherwise. The axiom for `'PSEUDO ASSERT'(X)` can therefore both return the goal that adds `X` to the list and the goal that doesn't.

$$\texttt{'PSEUDO ASSERT'(X)} \ {}_{L}\Rightarrow_{L1} \ \langle\texttt{L=[X|L1],L=L1}\rangle \qquad \text{[PA v.2]}$$

The rule for a conjunction is changed so that the static test for success is moved into a run-time test, using the if-then-else construct.

$$\frac{\texttt{deterministic(}G_1\texttt{)} \qquad G_2 \ {}_{L}\Rightarrow_{L1} \ \langle G_2^t,G_2^f\rangle}{(G_1,G_2) \ {}_{L}\Rightarrow_{L1} \ \langle(G_1\texttt{-> }G_2^t\texttt{; }G_2^f\texttt{),}G_2^f\rangle} \quad \text{[conj. v.3]}$$

The special case in which G_1 is successful is handled by subsequent partial evaluation of the produced goal.

The disjunction rule is extended to handle the pair of goals.

$$\frac{G_1 \ {}_{L}\Rightarrow_{L1} \ \langle G_1^t,G_1^f\rangle \qquad G_2 \ {}_{L1}\Rightarrow_{L2} \ \langle G_2^t,G_2^f\rangle}{(G_1;G_2) \ {}_{L}\Rightarrow_{L2} \ \langle\texttt{copy\_term(t(}G_1^t\texttt{,L,L1),t(}G_1^c\texttt{,L,L1)),}G_1^c\texttt{,}G_2^t\texttt{),(}G_1^f\texttt{,}G_2^f\texttt{)}\rangle} \quad \text{[disj. v.3]}$$

At the top-level we are only interested in the result when `Goal2` is executed, and the second part of the pair returned is not used.

$$\texttt{(Goal2; false)}\ {}_{\texttt{List}}\Rightarrow_{\texttt{[]}}\ \langle \texttt{Goal3}, \_ \rangle \qquad \text{[false v.3]}$$

Non-deterministic goals

The above scheme could be extended to handle non-deterministic goals as well. This has not been considered worthwhile as these goals have to be handled by a call to `findall` to collect all solutions.

For example, partial evaluation of `findall(X,(p(X);q(X)),L)` would be partially evaluated into

```
findall(X,p(X),Lp), findall(X,q(X),Lq), append(Lp,Lq,L)
```

Besides the obvious inefficiency of calling `findall/3` twice instead of once, there is also the overhead of the `append`. If there had been a standard version of `findall/3` that returned d-lists instead, this solution could be contemplated as the overhead of the `append/3` operation would then disappear.

Returning to the example

We are now able to show the exact form of `Goal3` in the example above where the stages for partial evaluation of `findall(X,(X=1;X=2),List)` were shown. The form of `Goal3` produced by the extraction rules is rather complex:

```
(copy_term(
    t((copy_term(
        t((X=1->List=L1,L2=L3,L1=[1|L3];List=L2),List,L2),
        t(L4,L5,L6)),L4,(X=2->L6=L7,L8=L9,L7=[2|L9];L6=L9)),
      L5,L8),
    t(L10,L11,[]))
```

When this expression is partially evaluated again, its final form depends mainly on the treatment of `copy_term`, which in turn depends mainly on which variables are input variables. As already shown, the final expression may be as simple as `List=[1,2]`.

Another example

Original program:

```
p(L) :- findall(X,(r(X),write(X)),L).
r(2).
r(1).
```

Query given to the partial evaluator: `p(L)`.
Program produced by Mixtus:

```
p(L) :- p2(L).
p2(L) :- write(2), write(1), L=[2,1].
```

This code can be produced as `write/1` is deterministic (used in the extraction) and successful (used when the tests are simplified). The unification `L=[2,1]` cannot be moved to the head as `write/1` is a side-effect predicate.

4.2.4.7 bagof/3

The following definition of `bagof/3` is used[9]

```
bagof(X,Goal,L) :-
        free_variables(X^Goal,Freevars),
        findall(Freevars-X,Goal,List),
        keysort(List,Listsorted),
        group_pairs(Listsorted,Listgrouped),
        member(Freevars-L,Listgrouped).
```

First all the “free” variables are located and stored in the list `Freevars`. The free variables are those that are not “bound”, and the “bound” variables are essentially those in `X` or in the left part of an explicit quantification V^G in the `Goal`. The solutions from `bagof/2` are grouped so that for each value (modulo variable renaming) of the `Freevars` the corresponding values of `X` are returned in `L`. This is accomplished by the call to `findall/3`, which collects the pairs `Freevars-X` in `List`. Then `keysort/2` is used to sort the list with respect to the `Freevars` so that pairs with the same value (modulo renaming) of `Freevars` become adjacent. Then `group_pairs/2` groups into a list those `X` values which have the same value of `Freevars`. Finally `member/2` is called to non-deterministically select the values.

This definition together with definitions for `free_variables/2`, `group_pairs/2` and `member/2` could be used for partial evaluation of `bagof/3`. But the definition is too complex and hides some important properties of `bagof/3`, in particular those related to the treatment of input variables and free variables. So specialized code is used instead, as described below.

1. First the free variables of `X^Goal` are extracted and put in `Freevars`. Of these variables, the input variables are put in `Inputfreevars`. The variables in `X^Goal` that are not free variables are put in `Boundvars`, and similarly the input bound variables are put in `Inputboundvars`. We now demand that either `Inputfreevars` or `Inputboundvars` is empty. If not, there might be an aliasing between variables in the two sets at run-time and a bound variable may become a free variable. As the following steps assume that we know at partial evaluation time which variables are free, the `bagof/3` goal will not be modified if this is not the case.

[9] Our definition of `bagof/3` is a simplified but semantically equivalent version of code written by R.A. O’Keefe. The implementation of `bagof` in SICStus is also based on his code.

2. Then the composite goal `(Goal,'PSEUDO ASSERT'(Freevars-X))` is partially evaluated, producing `Goal2`.

3. Thereafter `Goal2` is scanned extracting all the different instantiations of `Freevars` produced, putting them into a list `Freevarslist`. All the goals in this list must be comparable[10], otherwise the call to `keysort/2` (as shown above in the definition of `bagof/3`) cannot be partially evaluated. If they are not comparable, `bagof(X,Goal2adjust,L)` is returned as a residue. Normally `Goal2adjust` is identical to `Goal2`, but if `Goal2` contains some variables that did not occur in the original `Goal`, these variables (`Newvars`) must be made "bound". The "existential" quantifier "^" of Prolog is used[11], so that `Goal2adjust` becomes `Newvars^Goal2`.

4. A call to `bagof` can be non-deterministic, returning one list for each variable variant of `Freelist`. In the partial evaluator this is implemented so that a list of lists, `LL`, is constructed, with the same length as `Freevarslist`. Each sub-list of `LL` contains the solutions returned for the corresponding element of the `Freevarslist`. Just as for `findall/3`, a transformation system is used for LL and the Goal3 that will compute LL. Only two transformations have to be changed

$$\texttt{Goal2} \ \underset{\texttt{LL}}{\Rightarrow}_{\texttt{[]}} \ \texttt{Goal3}$$

and

$$\texttt{'PSEUDO ASSERT'(Freevars-X)} \ \underset{\texttt{LL}}{\Rightarrow}_{\texttt{LL1}} \ \texttt{addsol(Freevars,X,LL,LL1)}$$

`addsol(Freevars,X,LL,LL1)` returns the pair of goals $\langle \texttt{LL}_i\texttt{=[X|LL1}_i\texttt{]}, \texttt{LL}_i\texttt{=LL1}_i \rangle$, where `Freevars` is a variable variant of the i:th the element of `Freevarslist`, and where $\texttt{LL}_i$ and $\texttt{LL1}_i$ are the i:th elements of `LL` and `LL1` respectively. The other elements of `LL` and `LL1` are set to be identical.

5. To ensure that the various solutions lists are produced non-deterministically, the goal has to be a disjunction. The test `L\==[]` ensures that only non-empty lists are returned. This is in accordance with the definition of `bagof/3`.

$$\texttt{Goal3,} \ \vee_i \ \texttt{(Freevars=Freevarslist}_i\texttt{,L=LL}_i\texttt{),} \ \texttt{L\==[]}$$

This composite goal is partially evaluated producing the final residue `Goal4`.

If `Goal3` contains a cut, its scope is limited by replacing `Goal3` by `call(Goal3)`. If `Goal3` is logical, the disjunction may be put first. This will often result in slightly better code.

---

[10] As the test for comparability becomes sharper if the notion of input variables is taken into consideration, it is best to test for comparability while extracting the `Freevars`, since the context of each instance of `Freevars` is known then.

[11] In [Lakh90] the same method is employed to adjust for the extra variables.

Examples

Some examples are given below which show how dramatically different code may result from small changes in the input program. For all examples below the query given to Mixtus is `t(V1,…Vn)` where `V1` to `Vn` are variables. The arity n of `t/n` varies from example to example.

```
t(X,Y,L) :- bagof(X,append(X,Y,[1,2]),L).
```
[Ex.1]

where `append/3` is defined as usual. The result of partial evaluation is simply

```
t(X, Y, L) :-
   bagof(X, (X=[],Y=[1,2];X=[1],Y=[2];X=[1,2],Y=[]), L).
```
[PE.1]

that is, the `bagof`-call has not been removed and only the call to `append/3` has been unfolded. Mixtus cannot do more as there is a possibility of aliasing between `X` and `Y` which makes it unclear whether `Y` is a bound or free variable. This risk of aliasing is removed if `X` does not occur as an argument:

```
t(Y,L) :- bagof(X,append(X,Y,[1,2]),L).
```
[Ex.2]

Mixtus is now capable of removing `bagof/3` completely, but the code produced is rather complex:

```
t(Y, L) :-
         copy_term(Y, C),
         copy_term(C, D),
         (   D=[1,2] ->
             E=[[]]
         ;   E=[]
         ),
         copy_term(C, F),
         (   F=[2] ->
             G=[[1]]
         ;   G=[]
         ),
         (   C=[] ->
             H=[[1,2]]
         ;   H=[]
         ),
         (   Y=[],
             H=L,
             L\==[]
         ;   Y=[1,2],
             E=L,
             L\==[]
         ;   Y=[2],
             G=L,
             L\==[]
         ).
```
[PE.2]

This code can be considerably simplified if it can be assumed that `Y` is not bound, which is the case if the definition is:

```
t(L) :- bagof(X,append(X,Y,[1,2]),L).
```
[Ex.3]

The resulting code just becomes

```
t([[1,2]]).
t([[]]).
t([[1]]).
```
[PE.3]

This result may be a bit surprising if one hasn't noticed that `Y` is a free variable, and bagof will return a separate list of solutions for each set of values of the free variables. In this case `Y` can have the values `[]`, `[2]` and `[1,2]`. The non-deterministic results of bagof are returned with the possible values of `Y` sorted, which explains the strange order of the solutions above.

A more common use of bagof would be to declare `Y` as a bound variable, as in this example:

```
t(L) :- bagof(X,Y^append(X,Y,[1,2]),L).
```
[Ex.4]

The result of partial evaluation then simply is

```
t([[],[1],[1,2]]).
```
[PE.4]

4.2.4.8 setof/3

`setof/3` is partially evaluated as if it were defined by

```
setof(X,Goal,List) :- bagof(X,Goal,List2), sort(List2,List).
```

4.2.4.9 clause/2

The predicate `clause/2` is used to access the source code and converting it to a data structure. As shown in section 5.1, `clause/2` is very useful for writing meta-interpreters in Prolog. In SICStus Prolog 0.6 and earlier `clause/2` could only be applied to interpreted code, thereby creating a difference in behavior between interpreted and compiled code, something which is generally undesirable. In version 0.7 this has been changed so that `clause/2` only works for predicates that have been declared "dynamic". Such predicates are always stored in interpreted form, even if they appear in a file that is compiled.

Unfortunately, if `clause/2` is only applicable to dynamic code then it is impossible to partially evaluate `clause/2`; dynamic code can be changed arbitrarily at run-time. Mixtus is not capable of handling code that changes and will always refrain from unfolding calls to dynamic code. Therefore calls to `clause/2` can not be unfolded. Note that this is a basic semantic problem of `clause/2`, and almost any program transformation system will run into this problem when handling `clause/2`.

The only way out of this dilemma seems to be a change of the semantics of `clause/2`. We suggest that there should be a possibility of declaring predicates "visible", independent of whether they are "dynamic" or not. The semantics of `clause/2` would be changed so that it is

only applicable to visible predicates. An implementation of this is straightforward if the visible predicates are interpreted. It also seems natural to change the semantics of `listing/0` so that it lists the visible predicates.

4.2.5 Constraints

Fujita appears to have been the first to consider partial evaluation and constraints [Fuji87]. Lately, Smith and Hickey have continued these investigations in a number of publications [SmHi90, Smit90a, Smit90b, SmHi91]. In Mixtus, the emphasis is not on handling constraints, and in order not to complicate the system they only have a limited support. For several applications, as shown in the examples of partial evaluation, this support seems sufficient, however.

4.2.5.1 freeze/2 and wait-declarations

In SICStus Prolog a predicate `p` with arity `n` may have a wait-declaration of the form

```
:- wait p/n.
```

which means that a call to the predicate will be deferred until the first argument becomes a non-variable. Although any user-defined predicate may have a wait-declaration, it is sometimes convenient to use the built-in predicate `freeze/2`, which is defined as[12]

```
:- wait freeze/2.
freeze(X,G) :- call(G).
```

The built-in predicate `freeze/2` has the following definition:

`freeze(X,Goal)` will suspend `Goal` until `nonvar(X)`.

In SICStus Prolog the checking of `nonvar(X)` is made after each unification that binds X. This means that a unification may potentially start an execution of `Goal`. As `Goal` may contain calls to arbitrary Prolog predicates containing for instance side-effects, a unification in an otherwise logical predicate may trigger these effects, making the predicate behave as if it had side-effects. Thus the program transformations introduced so far may be incorrect if `freeze/2` is present in the code. Some examples:

```
p(X) :- write(X).
q(3) :- false.
```

The query `freeze(X,p(X)),q(X)` will write “3” and then fail. However, one of the program transformations would transform the call `q(X)` into `false`, as the predicate fails without any side-effects. The resulting code would in this case just fail, and never trigger `freeze/2`.

---

[12] Alternatively, `freeze/2` may be taken as the primitive, and any call $p(x_1,\ldots,x_n)$ to a wait-declared predicate is replaced by $freeze(x_1,p(x_1,\ldots,x_n))$.

There is also a problem when `freeze/2` contains a call to a propagation sensitive predicate:

```
p(t(Z)) :- var(Z).
q(t(Z)) :- Z=3.
```

The query `freeze(X,p(X)),q(X)` will succeed as the test `var(Z)` is performed before `Z=3`. If the clause for `q` is transformed into `q(t(3)).`, the query would instead fail[13].

The examples above show that the normal transformations are not applicable if `freeze/2` is applied to a goal containing side-effects or non-logical predicates. Unfortunately, there are also cases where even logical predicates may cause problems. For example, if `freeze/2` is applied to a non-deterministic logical predicate, the choice-points may be moved:

```
p(X).
p(X).
q(7) :- !.
```

The query `freeze(X,p(X)),q(X)` will succeed just once, since the choice-point in `p/1` is removed by the cut in `q/1`. A program transformation would remove the cut in `q`, as it just has a single clause, and the unification is considered deterministic. The query applied to the transformed program would succeed twice.

The above examples show that a major revision of the transformations in the partial evaluator would be necessary to fully support `freeze/2`. In many cases transformations crucial for generating an efficient program would have to be inhibited in order to cope with `freeze/2`.

Therefore, the present partial evaluator will just support `freeze/2` when it does not cause any problems, i.e. when the frozen goal is logical and deterministic. The system will check this, and alert the user if that condition is violated. However, the examples above are not typical and in most cases the transformed program is acceptable, even if the condition is violated.

4.2.5.2 freeze/1

A call to `freeze(Goal)`, which suspends until `Goal` is ground, has similar problems as `freeze(X,Goal)`, and the same restrictions are put on the frozen goal. There is one special case that can be handled, regardless of the type of frozen goal: if `X` is a non-variable then `freeze(X,Goal)` is equivalent to `call(Goal)`. Similarly, for `freeze(Goal)` if `Goal` is ground.

---

[13] This example is only valid for interpreted SICStus Prolog. When compiled `q(t(Z)) :- Z=3` is changed into `q(t(3))` as a part of the compilation transformations, thereby making a slight difference in semantics between interpreted and compiled code.

The clause `q(t(Z)) :- true, Z=3` is however not changed by compilation as `true` is not removed, making the argument valid also for compiled code.

4.2.5.3 frozen/2 and call_residue/2

`frozen/2` and `call_residue/2` will not cause any problems for transformations in the partial evaluator as they do not freeze any goals, but rather check for frozen goals.

The classification of `freeze(X,Goal)` is the same as the classification of `Goal`. For `freeze(Goal)`, we can do better: even if `Goal` is propagation sensitive, `freeze(Goal)` is logical. Whenever the `Goal` in `freeze(Goal)` is executed, we know that it must be ground as this is the condition for execution. But as a general optimization, a ground propagation sensitive goal is actually logical.

4.2.5.4 dif/2

The call `dif(A,B)` puts a constraint on `A` and `B` so that whenever they become identical in a branch of the execution, that branch will fail. In the Mixtus implementation a `dif/2` goal is always executed. This works very well as the basic predicates used in the implementation such as `findall/3` and `assert/1` fully support these constraints.

Pruning

Partial evaluation of a clause is always performed using the predicate `call_residue/2` which executes a goal and returns as a list all those constraints that were created during the execution of the goal. Only those constraints are considered to be part of the clause produced. In conjunction with this, a pruning is also performed on the constraints so that the following constraints are not included in the final clause:

- `dif(`A,B`)` where A or B is a term containing a variable not occurring anywhere else in the clause.

An occurrence in another dif-constraint is not counted as occurrence in this case. A constraint of that kind can never be violated, so it is unnecessary and can be removed. The “Independence of Inequations theorem” [Lass88] guarantees that occurrences of a variable just in other dif-constraints can not make the clause fail. In SICStus Prolog, however, all remaining constraints are printed at top-level, even if they cannot be violated. This pruning operation may therefore change the behavior of the program, but it seems hard to imagine a situation where the user actually finds it useful to know the constraints of anonymous variables.

Generalized restart and folding

In the mechanism for generalized restart, anti-unification is used to find a term that is more general than two other terms. This mechanism now has to be extended to cope with dif-constraints. For simplicity, the constraints are not taken into account in the anti-unification, but rather just dropped. This will still produce a more general term, as a term without constraints is always more general than the same term with constraints. Thus the produced generalization does

not have any constraints at all. A more refined anti-unification, taking the constraints into account, is also possible, but has not been incorporated into Mixtus.

Also the instance test used in the folding test must be modified to cope with dif-constraint. A procedure which checks that a term t_1 is an instance of t_2 works as follows. First t_1 and t_2 are unified. All variables in t_1 must remain distinct variables after the unification. Then the constraints of the variables in t_1 are examined. No new constraints must have been added to any variable due to the unification.

For simplicity, the constraints are just compared syntactically, so two constraints may in fact be semantically identical without this algorithm detecting it. This means that some instance-tests fail due to the incomplete test, although one term actually is an instance of the other. For Mixtus, this shortcoming is no problem, as instance-tests are used for folding and a missed opportunity to fold will just lead to a generalized restart, where the dif-constraints are eliminated.

The test for variable variants, used to avoid recomputations, must also be extended to handle dif-constraints. For this, we use the observation that two terms t_1 and t_2 are variable variants iff t_1 is an instance of t_2 and t_2 is an instance of t_1.

Smith has in [Smit90b] more thoroughly investigated what the concepts anti-unification, instance and variable variants mean for terms with dif-constraints.

Making the constraints visible

Constraints are not automatically visible when the final program is printed using the built-in predicates `write/1` or `portray_clause/1`. In Mixtus the constraints are reified using the built-in predicate `frozen/2` which for a variable returns the constraints on that variable. When a clause is printed, all new constraints for a goal will precede the goal, and all constraints in the head will come first in the body of the clause. Care is also taken to remove any duplicate constraints.

An example of partial evaluation of a program using constraints is given in 6.1.2.

4.2.6 Excluded built-in predicates

Although the title of this thesis claims that “full Prolog” is handled by the partial evaluator, a few built-in predicates in SICStus have been excluded. They all have in common that they are rather unusual and very unlogical, and a partial evaluator that could support them would be substantially more complex. Whenever we discuss built-in predicates in this text, these predicates are implicitly excluded.

4.2.6.1 Built-ins accessing the goal-stack of Prolog

Predicates belonging to this group are

`ancestors/1` and `subgoal_of/1`

These built-ins are not supported since the unfolding of the partial evaluator may remove some predicates and predicates are renamed.

One way of letting these predicates view the goal-stack of the original program, would be to extend all generated predicates with one extra argument which would maintain the goal-stack structure of the original program. Due to efficiency considerations, this has not been done in our system. Instead these built-ins will be kept as they are, and will access the goal-stack of the generated program.

4.2.6.2 Built-ins accessing the actual names of predicates

Since new predicate names are created for the generated program, any code that relies on the actual names of predicates will not work properly after partial evaluation. Thus any built-in predicate that accesses the Prolog code, such as `clause/2`, `predicate_property/2`, is potentially dangerous. But most uses of these predicates are harmless, as the actual names of the predicates are not relevant. It does not seem feasible for the partial evaluator to distinguish between the harmless and the dangerous cases, so the user is instead cautioned when using these predicates.

4.2.6.3 Limited support for setarg/3

In SICStus there is a predicate `setarg(N,F,A)` which *replaces* the `N`:th argument of the structure bound to F with `A`. On backtracking, the value of the argument is restored. For example

```
X=t(a,7), setarg(2,X,8), X=t(a,8).
```

will succeed! The variable X is used as a pointer referring to a specific occurrence of a structure. Throughout the partial evaluator forward propagation is assumed applicable, i.e. “X=t, e[X]” is exactly the same as “e[t]”. As `setarg/3` is the only predicate that violates this fundamental property[14], it is not supported directly in the partial evaluator.

However, there is a workaround by using a programming methodology. `setarg/3` is normally used on just a single or a few structures in a program. Remove all explicit occurrences of those functors by calls to an access predicate which is placed in a separate file which is not to be seen by the partial evaluator. This program transformation will guarantee that the “functor” argument of setarg will remain a variable.

The transformation applied to the example above would give

```
make_t(X,a,7), setarg(2,X,8), make_t(X,a,8).
```

with the access predicate for the structure defined as

---

[14] In Quintus Prolog 2.5 forward propagation is not applicable for the arithmetic predicates.
`X=1+2, Y is X` will fail with an error message, whereas `Y is 1+2` succeeds. In version 3.0 the two goals are however equivalent.

```
make_t(t(A,B),A,B).
```

The definition of the access predicate should not be available to the partial evaluator, but must of course be incorporated into the generated program. The best way to do this is to unfold all calls to the access predicate in the generated program, so there will be no overhead introduced by accessing the structure. The following execution times illustrate the effects of these transformations for a program containing `setarg/3`.

	relative execution time
original program	1.00
⇓ transformation to use access predicates	
transformed program	1.51
⇓ partial evaluation	
partially evaluated program	1.47
⇓ access predicates are unfolded	
final program	0.87

The final speed-up figure above is not very impressive as the program (a constraint solver) does not contain much static overhead. Note that without the final unfolding of the access predicate, partial evaluation cannot compensate for the introduction of the access predicates.

The current partial evaluator does not automatically perform the above extraction of the access predicates and their subsequent unfolding.

4.2.7 Cyclic structures

During unification a cyclic structure may be created, for example when `X` is unified with `f(X)`. There are essentially three approaches to this problem in Prolog implementations:

- Occur check. A unification which would create a cyclic structure fails. Unfortunately, doing an occur check for each binding would in general make the unification prohibitively inefficient. No practical Prolog systems incorporate an occur check as default for this reason.
- Ignore the problem. No occur check is made and cyclic structures may well be created. It is up to the programmer to ensure that if a cyclic structure is created, it will not cause any problems. Unification of `X` and `Y` when both are cyclic (eg. `X=f(X), Y=f(Y)`) may never terminate. This pragmatic approach is presently the normal solution in most Prolog implementations.
- Full cyclic unification. It is not only possible to extend ordinary unification to handle cyclic structures, but it can also be made in such a way that it has only a negligible

overhead compared with ordinary unification [Morr81, Hari84]. In fact, there are cases where unification of non-cyclic structures may be speeded up considerably by supporting cyclic structures. Full cyclic unification is the approach taken by SICStus Prolog. However, of the other built-in predicates in SICStus Prolog 0.7, only the term comparison predicates `compare/3`, `@</2`, `@=</2`, `@>/2`, `@>=/2`, `==/2`, `\==/2`, `sort/2`, `keysort/2` support cyclic structures.

4.2.7.1 Partial evaluation of cyclic structures

As the partial evaluator is meant for SICStus Prolog it must be able to cope with cyclic structures. Just as the partial evaluator may get into "infinite" loops that no intended instance of the program would go into, the partial evaluator may create cyclic structures that would not normally be created. Although this does not seem to happen in practice, cyclic structures must be supported to guarantee termination of the partial evaluation process.

Unfortunately, as just mentioned SICStus Prolog only supports cyclic structures for unification and term comparison. Many other built-in predicates such as `assert/1`, `findall/3` loop indefinitely if given a cyclic structure. As the partial evaluator relies heavily on these predicates, we have a problem here. Two solutions are proposed: either the cyclic structures are avoided, or they are handled.

Avoiding cyclic structures

One solution is to check each unification in the partial evaluator so that no cyclic structures will be created. If so, the unification is left as it is as a residue. This is the solution presently chosen.

Whenever a unification is encountered, the predicate `unify_cyclic(X,Y,Residue)` will be called. For example,

```
unify_cyclic(f(X,h(i(Y)),Z), f(g(X),Y,7), Residue)
```

will return

```
Z=7
Residue=(X=g(X), Y=h(i(Y)))
```

This solution means that the knowledge of the bindings of cyclic variables is not propagated to the rest of the code, which is far from satisfactory for programs which rely on cyclic structures.

Handling cyclic structures

Another much better solution of the problem of handling cyclic structures is not to avoid creating them, but to rewrite and extend the built-in predicates so that they fully support cyclic structures. These "cyclic" versions of the built-in predicates could be written entirely in Prolog, as there is a simple and natural way to let non-cyclic structures represent cyclic structures: Let a functor, say "ref" taking an integer argument be reserved for this purpose. The integer argument

of “ref” tells how many levels of functors up it refers to. The cyclic structure in the example above could then be represented by

```
f(g(ref(1)), h(i(ref(2))),7).
```

This approach remains to be investigated and implemented. Code not yet incorporated into Mixtus 0.3, for cyclic unification and anti-unification can be found in appendix A.

4.3 Printing and pruning

Pruning or “dead code elimination” is a standard technique to remove code that is not reachable. In Mixtus, the partial evaluation ends with printing only those predicates which are reachable from the top level query. If a meta-call with a variable as an argument occurs in the final program, it cannot be determined which predicates are actually called. In that case, a warning is issued to the user saying that potentially all original code is needed.

The final printing of the program will also include all op-declarations present in the original program. Each generated predicate is also preceded by a comment of the form

```
% Anew:-A.
```

where Anew and A are related as described in the basic procedure. If the predicate has been generated by a generalized restart, an instance of the above clause is printed. If applicable, a dynamic-declaration, multi-file-declaration or wait-declaration is also printed.

4.4 Preventing code explosion

Too much unfolding may cause code explosion so that the produced program becomes much too large. As already mentioned, unfolding is not applied to predicates that contain a cut or are recursive or are created by the generalized restart mechanism. Sometimes this may not be enough, and Mixtus has an adjustable maximum limit (the parameter maxnondeterm) to the number of clauses a predicate may have if unfolding is to be allowed. Normally this limit is set to 10, but for some programs (database-like applications) a higher value is better. Setting the limit to 1 ensures that indexing will not be lost as shown in the following example.

Most Prolog implementations just do indexing on the main functor of the first argument of the head of a clause[15]. This may counteract the advantages of unfolding as the indexing may be lost. Let this program be partially evaluated with respect to `q(A)`:

```
q(f(X)) :- f(X).   | f(1).   | g(a).
q(g(X)) :- g(X).   | f(2).   | g(b).
                   | f(3).   | g(c).
```

[15] That is, if the main functor of the first argument is known when the predicate is called, the right clause(s) will be taken immediately using a jump-table. This drastically improves performance especially if there are many clauses where indexing can be used to distinguish between the clauses.

Then a program just consisting of the following assertions could be produced.

```
q(f(1)). q(f(2)). q(f(3)). q(g(a)). q(g(b)). q(g(c)).
```

Thus indexing is lost for the inner structure (`X`) in making all the facts explicit. By preventing unfolding where there are multiple matching clauses it can be guaranteed that no indexing will be lost. But preventing unfolding will also prevent propagation of bindings. Thus it is not clear which choice is better, and the user is given the possibility of controlling whether unfolding should be done for non-deterministic predicates.

The example above also shows that for real Prolog implementations it may be unsatisfactory to use the number of inferences as a measurement of execution complexity.

Smith and Hickey [Smit90a] also discuss this issue, and name the unfolding strategy corresponding to *maxnondeterm*=1 “lazy unfolding”. “Eager unfolding” corresponds to *maxnondeterm*=∞ .

Another kind of code explosion may theoretically occur even if all predicates are nonrecursive and are defined by a single clause. If the same predicate is called (directly or indirectly) more than once in a clause body, there is a potential danger of code explosion. For example, let the initial query be `p(7)` and let all calls be unfolded in

```
p(X) :- p1(X), p1(X).
p1(X) :- p2(X), p2(X).
p2(X) :- p3(X), p3(X).
p3(X) :- p4(X), p4(X).
p4(X) :- write(X).
```

We then get 16 explicit calls to `write(7)`, which is perhaps acceptable, but it is very easy to extend the example so that the result can clearly be characterized as a code explosion. As ordinary Prolog programs do not seem to contain this kind of code, no measures have been taken in the partial evaluator to prevent this kind of behavior.

5 The implementation of Mixtus

In this chapter we will use the standard technique of deriving a partial evaluator from an interpreter. This was first done for Prolog by Komorowski [Komo81] and later also in Prolog by Venken [Venk84]. This chapter provides a separate path to understanding some of the basic mechanisms in Mixtus, and can be read more or less separately from the other chapters. This also means that there will be a slight overlap between this chapter and the rest of the text.

First some interpreters for Prolog will be shown. Then one of these will be enhanced to become a partial evaluator for Prolog which does not preserve the order of the solutions. A sequence of more and more refined partial evaluators is then presented, which ends with a version capable of handling full Prolog. For complexity reasons, the full Mixtus system will not be presented, so the implementation of cut and generalized restart will not be shown.

5.1 Some interpreters for Prolog

As a starting point for the partial evaluator, a simple meta-interpreter for Prolog is used.

```
solve((A,B)) :- !, solve(A), solve(B).
solve((A;B)) :- !,
        ( solve(A)
        ; solve(B)).
solve(A) :- predicate_property(A,built_in), !,
        call(A).
solve(A) :- clause(A,B), solve(B).
```

Interpreter 1

The built-in predicate `clause/2` will backtrack over all matching clauses and non-deterministically return different clause bodies. In the following, a deterministic variant of `clause/2` called `clause_disj/2` will be used, which combines all head unifications and bodies into a single disjunction. For instance, if `append/3` is defined as

```
append([],A,A).
append([B|C],D,[B|E]) :- append(C,D,E).
```

a call to `clause_disj(append(X,Y,Z),W)` will bind `W` to

```
( append(X,Y,Z)=append([],A,A)
; append(X,Y,Z)=append([B|C],D,[B|E]), append(C,D,E))
```

In section 3.1, more details of this transformation into a single clause are discussed. In appendix C, the code for this transformation is shown.

Interpreter 1 is appealing in its simplicity, but unfortunately not quite adequate for our purposes as the conjuncts in a conjunction are solved independently. The following continuation-based interpreter is more adequate[16]:

```
solve(A) :- s(A,true).

s_goal(G) :- clause_disj(G,Body), solve(Body).

s((A,B),C) :- !, s(A,(B,C)).
s((A;B),C) :- !,
        ( s(A,C)
        ; s(B,C)).
s(A,C) :- predicate_property(A,built_in), !,
        call(A), s_next(C).
s(A,C) :- s_goal(A), s_next(C).

s_next(true).
s_next((A,B)) :- s(A,B).
```

Interpreter 2: a “conjunction-list”

Interpreter 2 has the advantage of knowing the goals to be executed after a certain goal, which will be shown to be crucial to our partial evaluator. The second argument of `s/2` acts as a continuation, containing the not yet executed goals. This argument is always a “conjunction-list”, i.e. it has the same structure as a list but `','` is used instead of `'.'` and `true` is used instead of `[]`. A “conjunction-list” is thus either `true` or `(A,B)`, where `A` can be any term, but `B` must be a “conjunction-list”. The predicate `s_next/1` pops the top goal, if any, from the continuation, and continues the execution with that goal.

This interpreter reflects the Prolog execution on a suitably “low” level and is a good starting-point for the partial evaluator which is basically an enhancement of this interpreter. As an example of another enhancement, an interpreter that supports “cut” without using “ancestral” cuts

[16] The first partial evaluator for full Prolog by Venken [Venk84] was also continuation-based. An ordinary list works equally well as a "conjunction-list", and could be used instead if lists are particularly well supported by the Prolog system.

is shown below. This is the technique used by Mixtus to support cut. All execution of `s/4` takes place within the scope of the cut in `solve/1`. The predicate `s/4` has two new arguments: `Cut` and `After`. If a cut (i.e. "`!`") is encountered during execution `Cut` is set to `true` and `After` is set to contain all goals after the cut. The cut in `solve/1` will then be executed and remove all choice-points in `s/4`. Execution may then continue with the remaining goals in `After`. Notice that a single cut suffices to execute all cuts encountered in the program to be interpreted.

```
solve(G) :- s(G,true,Cut,After),
        ( Cut=true, !, solve(After)
        ; true).

s_goal(G) :- clause_disj(G,Body), solve(Body).

s(!,B,true,B) :- !.
s((A,B),C,Cut,After) :- !, s(A,(B,C),Cut,After).
s((A;B),C,Cut,After) :- !,
        ( s(A,C,Cut,After)
        ; s(B,C,Cut,After)).
s(A,B,Cut,After) :- predicate_property(A,built_in),!,
        call(A), s_next(B,Cut,After).
s(A,B,Cut,After) :- s_goal(A), s_next(B,Cut,After).

s_next(true,false,_).
s_next((A,B),Cut,After) :- s(A,B,Cut,After).
```

Interpreter 3: an interpreter that supports cut

This interpreter is similar to [OKee85] but is like [Lakh89b] able to handle arbitrary combinations of conjunctions and disjunctions in the body of the clauses. The interpreter in [Cava89] is capable of handling cut without having a "conjunction-list", but is considerably more complex. If the built-in predicate `reduce/1` is present, it is also possible to handle cut without the "conjunction-list" as shown in [Ster86b]. The predicate `reduce/1`, which cuts to a certain goal on the stack, is however only present in a few Prolog systems.

5.2 Partial evaluator 1: Generating residues

The first three partial evaluators presented here all use logical unfolding, not unfolding for Prolog. This means that the order of the solutions computed by the produced program need not

be the same as that computed by the original program. This will be corrected in the fourth partial evaluator which is capable of handling full Prolog.

The partial evaluator is structured as a specialized meta-interpreter. Some goals are not executed, and are left as a *residue*. These goals form the bodies of the clauses in the program produced by partial evaluation. The final partial evaluator will be "derived" through a sequence of gradually refined partial evaluators. As mentioned above, the first partial evaluator is derived from Interpreter 2:

```
p_goals(A,Residue) :- p(A,true,true,Residue).

p_goal(A,Rin,Rout) :- clause_disj(A,B),
        p(B,true,Rin,Rout).

p((A,B),C,Rin,Rout) :- !, p(A,(B,C),Rin,Rout).
p((A;B),C,Rin,Rout) :- !,
        ( p(A,C,Rin,Rout)
        ; p(B,C,Rin,Rout)).
p(A,C,Rin,Rout) :- predicate_property(A,built_in), !,
        (evaluable(A) ->
            call(A), p_next(C,Rin,Rout)
        ;   entergoal(A,C,Rin,Rout)
        ).
p(A,C,Rin,Rout) :-
        (should_unfold(A) ->
            p_goal(A,Rin,Rout2),
            p_next(C,Rout2,Rout)
        ;   entergoal(A,C,Rin,Rout)
        ).

p_next(true,Rin,Rin).
p_next((A,B),Rin,Rout) :- p(A,B,Rin,Rout).

entergoal(A,C,Rin,Rout) :- p_next(C,(Rin,A),Rout).
```

Partial evaluator 1: generating residues, derived from Interpreter 2

The top level predicate `p_goals(A,Residue)` takes a goal `A` and non-deterministically returns different residues corresponding to the parts of the program that cannot be evaluated by the partial evaluator. These residues are used to construct the resulting program, as will be shown

below. The two last arguments (usually called `Rin` and `Rout`) of `p/4`, `p_goal/3`, `p_next/3` and `entergoal/2` are used for the input residue and the output residue respectively. Initially, the input residue is set to `true`.

5.2.1 Controlling the computation

Two predicates control which goals should be computed during partial evaluation: `should_unfold/1` (for user-defined predicates) and `evaluable/1` (for built-in predicates).

5.2.1.1 User-defined predicates: `should_unfold/1`

In this simple version of the partial evaluator, it is assumed that the user supplies the definition of `should_unfold/1`. As an example, consider the following program and unfold declaration:

```
append([], L, L).
append([H|T], L, [H|R]) :- append(T, L, R).

should_unfold(append(X,_,_)) :- nonvar(X).
```

Calling `p_goals(append([A,B,C],D,E),Residue)` returns

```
E=[A,B,C|D]
Residue=true
```

The resulting program produced by partial evaluation then becomes

```
append([A,B,C],D,[A,B,C|D]):- true.
```

Another example: calling `p_goals(append([A,B|C],D,E),Residue)` will return

```
E=[A,B|F]
Residue=(true,append(C,D,F))
```

and the resulting program is

```
append([A,B|C],D,[A,B|F]) :- true,append(C,D,F).
```

In the latter example, the residue contains a call to the original `append/3` definition as it could not be unfolded when its first argument was unbound due to the definition of `should_unfold/1`. In the next section, it will be shown how to create new predicate names for all generated predicates, so that there will be no confusion between the old program and the new.

The residue above also contains a superfluous call to `true`, which will be removed if the definition of `p_goals/2` is replaced by

```
p_goals(A,Residue) :-
        p(A,true,true,Rout), cleanup_conj(Rout,Residue).
```

where `cleanup_conj/2` removes unnecessary calls to `true`.

5.2.1.2 Built-in predicates: `evaluable/1`

The partial evaluation of built-in predicates is controlled by the predicate `evaluable/1`, which succeeds if the built-in predicate can be safely evaluated during partial evaluation. Some typical definitions are

```
evaluable(true).
evaluable(X=Y).
evaluable(var(X)) :- nonvar(X).
```

The predicates `true/1` and `=/2` may always be evaluated, whereas `var/1` is only evaluable if its argument is non-variable.

5.3 Partial evaluator 2: Generating clauses

In general, the partial evaluator just presented may non-deterministically return several residues. This makes it more convenient to use mixed computation instead of partial evaluation, where a new goal Q' is created from the distinct variables of the original goal Q. Each of the residues then corresponds to a clause for Q' in the generated program. A new predicate `peval_goal/3` is introduced which returns the new goal Q' (in `Anew`) and its defining clauses (`Cls`). For example,

```
peval_goal(member(X,[1,2]), Anew, Cls)
```

will return

```
Anew = memberx(X)
Cls = [(memberx(1):-true), (memberx(2):-true)]
```

provided `member/2` is defined as usual

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

should_unfold(member(_,X)) :- nonvar(X).
```

Another example for `member/2`:

```
peval_goal(member(X,[X,Y,Y|Z]), Anew, Cls)
```

will return

```
Anew = memberx(X,Y,Z)
Cls =  [(memberx(A,B,C) :- true),
        (memberx(A,A,C) :- true),
        (memberx(A,A,C) :- true),
        (memberx(A,B,C) :- member(A,C))]
```

The partial evaluator will automatically construct a new predicate name which is returned in `Anew`. The predicate call `reparametrization(A,Anew)` will given a goal `A` return a new predicate name in `Anew` which is neither the name of a built-in nor of a generated predicate. The arguments of the structure returned in `Anew` are exactly the distinct variables of `A`.

For example, calling `reparametrization(pred(term(X,Y,3),Y),Anew)` could return `Anew = predterm(X,Y).` The new relation `Anew` is defined as

```
Anew :- A.
```

which in the example above becomes

```
predterm(X,Y) :- pred(term(X,Y,3),Y).
```

As can be seen from the code below, the built-in predicate `findall/3` is used to collect all residues[17]. To ensure that each call to a user-defined predicate is fully explored before it is used, `findall/3` is also used to collect all residues which form the defining clauses (`Cls`). By exploring each predicate fully before unfolding, we open up the possibility of automatic unfolding as will be shown in the next section.

The actual unfolding occurs when `member((Anew :- B2), Cls)` is used to non-deterministically choose the generated clauses. The different bodies `B2` will thereby be entered as residues. A typical situation just before calling `member/2` might be `A=p(X,3)`, `Anew=px(X)` and

```
Cls=[(px(a) :- true), (px(Y) :- q(Y))].
```

Calling `member/2` will first generate `B2=true` (with `X=a`) and on backtracking `B2=q(X)`. Notice how the connection between the variables in the original goal `A` and the newly generated goal `B2` is established by letting `Anew` be unified with the heads of the clauses in `Cls`.

[17] As `findall/3` is fundamental for the partial evaluator but is not a standard predicate its definition is given here:

`findall`(*Template*,*Goal*,*List*)

For each successful execution of *Goal*, the corresponding instances of *Template* are copied and stored into the list *List*. The order of the elements in *List* correspond to the order of the solutions of *Goal*. If *Goal* fails immediately, *List* becomes the empty list.

```
peval_goal(A,Anew,Cls) :-
        reparametrization(A,Anew),
        clause_disj(A,B),
        findall((Anew :- Residue),
                p_goals(B,Residue),
                Cls).

p_goals(A,Residue) :-
        p(A,true,true,Rout), cleanup_conj(Rout,Residue).

p((A,B),C,Rin,Rout) :- !, p(A,(B,C),Rin,Rout).
p((A;B),C,Rin,Rout) :- !,
        ( p(A,C,Rin,Rout)
        ; p(B,C,Rin,Rout)).
p(A,C,Rin,Rout) :- predicate_property(A,built_in), !,
        (evaluable(A) ->
            call(A), p_next(C,Rin,Rout)
        ;   entergoal(A,C,Rin,Rout)).
p(A,C,Rin,Rout) :-
        (should_unfold(A) ->
            peval_goal(A,Anew,Cls),
            member((Anew :- B2), Cls),
            entergoal(B2,C,Rin,Rout)
        ;   entergoal(A,C,Rin,Rout)).

p_next(true,Rin,Rin).
p_next((A,B),Rin,Rout) :- p(A,B,Rin,Rout).

entergoal(A,C,Rin,Rout) :- p_next(C,(Rin,A),Rout).
```

Partial evaluator 2: generating clauses

5.4 Partial evaluator 3: Automatic unfolding and folding

One disadvantage of the previously presented partial evaluators is the need for unfold declarations. The next partial evaluator virtually eliminates that need by introducing fold transformations. It works as follows:

A stack of the goals (called `GS`) encountered is maintained, much in the way it is handled in an ordinary Prolog interpreter. Each entry in the stack is of the form `gs(A,Anew,Isrec)`, where `A` and `Anew` are related as described above and have all of their variables in common. In contrast to the goal stack of a Prolog interpreter, a *copy* has been made of `A` and `Anew` in order to prevent further instantiation. Thus the goal-stack contains copies of all ongoing partial evaluations. Recall that `Anew` is defined by

```
Anew :- A.
```

This means that whenever an instance of `A` is found during partial evaluation, it can be replaced by the corresponding instance of `Anew`. This check for possible foldings is done for each user-defined predicate by calling `can_fold(A,GS,Anew)`. A successful call will also instantiate the third component, `Isrec` ("is recursive"), to `true` as `A` must then be directly or indirectly recursive. The source code of `can_fold/3` can be found in appendix C.

As can be seen in the code below, besides collecting the residues in `peval_goal/5`, all goal-stacks (`GS`) are collected by a call to `findall/3`. The only reason for doing this is to unify all corresponding `Isrec`s in `extract/3` so that if any solution has set an `Isrec` to `true`, it will be set to `true` in the goal-stack to be used in the rest of the computation. There are more efficient ways of unifying all `Isrec`s without unifying the whole goal-stack, but this solution is chosen here for its simplicity.

If the check for folding fails, a call to `loop_prevention(A,GS)` is made. The call will succeed if a possible loop is detected and the goal `A` will simply be put into the residue, being left completely unevaluated. Different versions of loop prevention are presented in section 2.5, and no source code will be given here.

If neither a folding is possible nor a potential loop has been detected, the partial evaluator is called again with `peval_goal(A,GS,Anew,Cls,IsRec)`. It is the value returned by `IsRec` that determines if the newly generated predicate should be unfolded. If `IsRec` is `true` then no unfolding will occur, and by calling `save_predicate(Cls)` the generated predicate will be asserted in a database. If `IsRec` is not true, unfolding will occur by having `member/2` non-deterministically pick one of the clauses in `Cls`.

```
pe(A) :- reset_all,
         peval_goal(A,[],Anew,Cls,_IsRec),
         write_program(A,Anew,Cls).

peval_goal(A,GS,Anew,Cls,IsRec) :-
        reparametrization(A,Anew), clause_disj(A,B),
        copy_term(gs(A,Anew),gs(A2,Anew2)),
        NewGS = [gs(A2,Anew2,IsRec)|GS],
        findall(pair((Anew :- Residue),NewGS),
                p_goals(B,NewGS,Residue),
                L),
        extract(L,Cls,NewGS).

p_goals(A,GS,Residue) :- p(A,true,GS,true,Rout),
                         cleanup_conj(Rout,Residue).
extract([],[],_).
extract([pair(Cl,GS)|R],[Cl|ClR],GS) :-
        extract(R,ClR,GS).

p((A,B),C,GS,Rin,Rout) :- !, p(A,(B,C),GS,Rin,Rout).
p((A;B),C,GS,Rin,Rout) :- !,
        (p(A,C,GS,Rin,Rout); p(B,C,GS,Rin,Rout)).
p(A,C,GS,Rin,Rout):- predicate_property(A,built_in),!,
        (evaluable(A) ->
            call(A), p_next(C,GS,Rin,Rout)
        ;   entergoal(A,C,GS,Rin,Rout)).
p(A,C,GS,Rin,Rout) :-
        (can_fold(A,GS,Anew) ->
            entergoal(Anew,C,GS,Rin,Rout)
        ;loop_detection(A,GS) ->
            entergoal(A,C,GS,Rin,Rout)
        ;   peval_goal(A,GS,Anew,Cls,IsRec),
            (IsRec\==true ->
                member((Anew :- B2), Cls),
                entergoal(B2,C,GS,Rin,Rout)
            ;   save_predicate(Cls),
                entergoal(Anew,C,GS,Rin,Rout))
        ).
p_next(true,_,Rin,Rin).
p_next((A,B),GS,Rin,Rout) :- p(A,B,GS,Rin,Rout).
```

```
entergoal(A,C,GS,Rin,Rout):-p_next(C,GS,(Rin,A),Rout).
```

Partial evaluator 3: automatic unfolding and folding

As can be seen from the code above, a new top-level predicate `pe/1` has been introduced. It will first reset the database containing the stored predicates, then perform a partial evaluation and finally print the produced new query and the generated program by calling `write_program/3`. For convenience, the original (`A`) and the new query (`Anew`) are presented as a clause "A :- Anew.".

For example, `pe(append([A,B,C],D,E))` will print

```
append([A,B,C],D,E) :- appendx(A,B,C,D,E). % A:-Anew
appendx(A,B,C,D,[A,B,C|D]).                % def of Anew
```

Another example: calling `pe(append([A,B|C],D,E))` will print

```
append([A,B|C],D,E) :- append(A,B,C,D,E).  % A:-Anew
append(A,B,C,D,[A,B|E]) :- appendx(C,D,E). % def of Anew
appendx([],A,A). % def of recursive generated predicate
appendx([A|B],C,[A|D]) :- appendx(B,C,D).
```

Two predicates are generated above: `append/5` with just a single clause, and `appendx/3`, which happens to be identical to the original `append/3`. As can be seen from the example above, `write_program/3` prints all predicates called directly or indirectly from the new query. The source code of `reset_all/0`, `save_predicate/1` and `write_program/3` is not given here, as the structure of these predicates is so simple.

Of course there is no guarantee that `can_fold/3` will eventually succeed for all programs and questions, but somewhat surprisingly this happens almost always in ordinary Prolog programs. However, without adding an extra loop check as an emergency break, this partial evaluator is not guaranteed to terminate.

5.5 Partial evaluator 4: Full Prolog

In this section a partial evaluator for full Prolog, i.e. including non-logical predicates such as `var/1`, `write/1` but without cut, is presented. In contrast to the partial evaluator for pure Prolog, the order of the solutions will not be changed by this variant.

To be able to cope with the increased complexity needed for supporting full Prolog, the partial evaluator is slightly restructured. The top-level predicates are however almost unchanged:

```
pe(A) :- reset_all,
         peval_goal(A,[],Anew,Cls,_IsRec),
         write_program(A,Anew,Cls).

peval_goal(A,GS,Anew,Cls,IsRec) :-
        reparametrization(A,Anew),
        clause_disj(A,B),
        copy_term(gs(A,Anew),gs(A2,Anew2)),
        NewGS = [gs(A2,Anew2,IsRec)|GS],
        findall(pair((Anew :- Residue),NewGS),
                p_goals(B,NewGS,Residue),
                L),
        extract(L,Cls,NewGS).

p_goals(A,GS,Rout) :-
        p(A,state(true,GS,true,Res)),
        cleanup_conj(Res,Rout).

extract([],[],_).
extract([pair(Cl,GS)|R],[Cl|ClR],GS) :- extract(R,ClR,GS).
```

Partial evaluator 4, part 1 of 9: Top-level part for full Prolog

As the arguments `C`, `GS`, `Rin` and `Rout` are seldom accessed, they are stored into a single structure `state(C,GS,Rin,Rout)`. This enables us to remove those four arguments from several predicates and instead just have one called `State`.

The predicate `p` will classify a goal encountered into one of five groups: meta-predicates, built-in predicates, user-defined predicates, generated predicates and unknown predicates. As an optimization it is also checked whether the goal has already been partially evaluated (by `is_already_pevaled/2` which inspects the asserted data base of generated predicates). If so, the new goal is entered into the residue. Although not strictly necessary, this optimization is crucial for making the partial evaluator able to handle larger programs, by avoiding unnecessary recomputations.

```
p(A,State) :- is_meta(A), !, peval_meta(A,State).
```

```
p(A,State) :- predicate_property(A,built_in), !,
        peval_builtin(A,State).
p(A,State) :- is_already_pevaled(A,Anew), !,
        entergoal(Anew,State).
p(A,State) :- predicate_property(A,(dynamic)),!,
        peval_user(A,State).
p(A,State) :- is_generated(A,Aorig),!,
        p(Aorig,State).
p(A,State) :- peval_unknown(A,State).
```

Partial evaluator 4, part 2 of 9: Classifying goals

We will now look at the five cases mentioned above starting with the *meta-predicates*. In contrast to the other built-in predicates, these predicates take Prolog goals as arguments. Only the treatment of disjunction and conjunction is shown below.

```
is_meta((_,_)).
is_meta((_;_)).

peval_meta((A,B),state(C,GS,Rin,Rout)) :- !,
        p(A,state((B,C),GS,Rin,Rout)).
peval_meta((A;B),State) :- !,
        ( p(A,State)
        ; p(B,State)).
```

Partial evaluator 4, part 3 of 9: Handling meta-predicates

The *built-in* predicates are handled exactly as before:

```
peval_builtin(A,State) :- evaluable(A), !,
        call(A), p_next(State).
peval_builtin(A,State) :- entergoal(A,State).

p_next(state(true,_GS,Rin,Rin)) :- !.
p_next(state((A,C),GS,Rin,Rout)) :-
        p(A,state(C,GS,Rin,Rout)).
```

Partial evaluator 4, part 4 of 9: Handling built-in predicates

The *user-defined* predicates are handled almost as before. Here all dynamic predicates are considered user-defined, as `clause/2` only works for dynamic predicates. The selector predicate `goalstack/2` is used for accessing the goal-stack in the `State` variable.

```
peval_user(A,State) :- can_fold(A,State,Anew), !,
        entergoal(Anew,State).
peval_user(A,State) :- goalstack(State,GS),
        loop_detection(A,GS), !,
        entergoal(A,State).
peval_user(A,State) :-
        goalstack(State,GS),
        peval_goal(A,GS,Anew,Cls,IsRec),
        (IsRec\==true ->
            member((Anew :- B2),Cls),
            entergoal(B2,State)
        ;   save_predicate(A,Anew,Cls),
            entergoal(Anew,State)
        ).

goalstack(state(_,GS,_,_),GS).
```

Partial evaluator 4, part 5 of 9: Handling user-defined predicates

The *unknown* predicates are predicates for which we lack any information. Normally they are predicates defined in a separate file not yet loaded. In that case the worst is assumed, i.e. they are treated exactly like predicates having a side-effect.

```
peval_unknown(A,state(C,GS,Rin,Rout)) :-
        peval_after(side,A,C,GS,Rin,Rout).
```

Partial evaluator 4, part 6 of 9: Handling unknown predicates

5.5.1 Handling non-logical predicates

So far, not much has been changed in the program compared to the partial evaluator for pure Prolog. The real difference comes in how new goals are entered into the residue by `entergoal/2` where the classification of the predicate is crucial. Here this classification is done by calling the predicate c`lassify_goals/2`. How this is done is shown in section 3.3.

```
entergoal(Goal,state(C,GS,Rin,Rout)) :-
        classify_goals(Goal,Class),
        peval_after(Class,Goal,C,GS,Rin,Rout).
```

Partial evaluator 4, part 7 of 9: Entering a goal as a residue and continuing

The `Class` may either be `side` (for side-effect predicates), `sensitive` (for propagation sensitive predicates) or `logical`, and the predicate `peval_after/6` has three clauses corresponding to those cases.

```
peval_after(side,Goal,C,GS,Rin,(Rin,Goal,Cls_or)) :-
        peval_goals(C,GS,Goalnew,Cls),
        to_disjunction(Cls,Goalnew,Cls_or).

peval_after(sensitive,Goal,C,GS,Rin,(Rin,Goal,Cls_or)) :-
        peval_after(side,Goal,C,GS,Rin,(Rin,Goal,Cls_or)),
        Cls_or \== false.

peval_after(logical,Goal,C,GS,Rin,Rout) :-
        peval_after(sensitive,Goal,C,GS,Rin,
                    (Rin,Goal,Cls_or)),
        (propagate_left(Cls_or,Cls_rest) ->
            (not_too_many_propagate_left(Goal) ->
                p(Goal,state((Cls_rest,true),GS,Rin,Rout))
            ;   Rout = (Rin,Goal,Cls_rest)
            )
        ;   Rout = (Rin,Goal,Cls_or)
        ).
```

Partial evaluator 4, part 8 of 9: Improvements for “logical” and “sensitive” predicates

In the next three sections, the three clauses of `peval_after/3` shown above will be explained in detail, one by one.

Side-effect goals

For a *side-effect* goal, whatever follows the goal must not in any way effect the goal. That is, no failures or instantiations of variables following the goal must change the goal. This is implemented by the first clause of `peval_after/6` as shown above. To achieve a “border” between the goal (`Goal`) and what follows it (`C`), the partial evaluator (`peval_goals/4`) is called again with `C` as an argument. The predicate `peval_goals/4` is quite similar to

`peval_goal/5`, but is used for arbitrary composite goals rather than single goals. There is however no need to extend the goal stack (which is used for checking for recursive calls) and the argument `IsRec` is not needed.

```
peval_goals(A,GS,Goalnew,Cls) :-
        reparametrization(A,Goalnew),
        findall(pair((Goalnew :- Residue),GS),
                p_goals(A,GS,Residue),
                L),
        extract(L,Cls,GS).
```

Partial evaluator 4, part 9 of 9: Handling composite goals

The result is (as always from the partial evaluator) a new query (`Goalnew`) and new clauses (`Cls`). The new query could form the residue for `C`, but instead a disjunction is formed by calling `to_disjunction(Goalnew, Cls, Cls_or)`(see appendix C). This is clearly advantageous when `Cls` is the empty list and `Cls_or` becomes `false`, or when `Cls` just consists of a single clause and `Cls_or` essentially becomes the body of that clause. If `Cls` contains several clauses, then `Cls_or` becomes a disjunction.

Under no circumstances may any argument in `Goalnew` become bound, even to an other argument in `Goalnew`, as this binding could propagate left past the side-effect goal.

Examples

The examples elaborated below will illustrate how a call to

`entergoal(Goal,state(C,GS,Rin,Rout))`

is handled. In all examples let `GS=[]`, `Rin=[]` and `Goal` be the side-effect goal `write(X)`.

First let `C` be `X=3`. Then a partial evaluation of `X=3` is performed by a call to `peval_goals(C,GS,Goalnew,Cls)` which returns `Goalnew = eq(X)` and `Cls = [(eq(3):-true)]`. The call `to_disjunction(Cls,Goalnew,Cls_or)` transforms this into `Cls_or = (X=3)`, so the final residue becomes `write(X),X=3`, i.e. the same as what we started with!

Similarly, if `C` is `(X=3,X=4)`, then `peval_goals/4` will return `Cls=[]`, and `to_disjunction/3` will return `Cls_or=false`. The residue in this case then will become `write(X),false`.

If `C` is `append(X,Y,[1,2])` then `peval_goals/4` will return `Goalnew = append2(A,B)` and

```
Cls =  [(append2([],[1,2]):- true),
        (append2([1],[2]) :- true),
        (append2([1,2],[]):- true)].
```

Calling `to_disjunction/3` will transform this into the disjunction `Cls_or = (A=[],B=[1,2]; A=[1],B=[2]; A=[1,2],B=[]).`

It might seem overly cautious not to allow any bindings to propagate left over the side-effect goal, even if the variables involved do not occur in that goal, but the following examples show that potential aliasing makes this necessary. The clause

```
p(X,Y,Z) :- write(X), Y=Z
```

cannot be transformed into

`p(X,Y,Y) :- write(X)`, as the query `p(A,A,3)` writes an unbound variable whereas `p(A,A,3)` writes "3". So potential aliasing makes it impossible in general to let variable bindings propagate left past non-logical goals.

Propagation sensitive goals

For *propagation sensitive* goals, an optimization is possible (which can be seen in `peval_after/6`): if the goal is followed by a `false`, the goal may fail. This follows directly from the definition of propagation sensitive goals `G`: we know that `G,false` is the same as `false`.

Logical goals

For *logical* goals, the above optimization is of course also valid, but more can be done. All logical goals `G` have the property that `G,X=t` is the same as `X=t,G`, so variable bindings may safely be propagated left past the goal. If the logical goal is followed by a disjunction (created by `to_disjunction/3`) we cannot propagate the bindings from the disjuncts. A call to the predicate `propagate_left(Cls_or, Cls_rest)` will succeed if `Cls_or` does not contain any disjunction and will in that case perform the leading unifications in `Cls_or` and return the rest in `Cls_rest`. To ensure that the partial evaluator will always terminate, a limit must be set on the number of left-propagations made for a specific goal. This limit is checked by calling the predicate `not_too_many_propagate_left/1`, which succeeds if the limit is not reached. If so, the partial evaluation is then restarted with the newly instantiated `Goal` followed by `Cls_or` (and `true` to form a conjunction-list). `Goal` and `Cls_or` may possibly contain calls to predicates generated during partial evaluation. That is why `is_generated(A,Aorig)` is called in `p/2` so that these calls will be retransformed to the original predicate names.

5.6 Other features of Mixtus

For space and complexity reasons, the source code for many other features of Mixtus will not be given here. The most important of these features are however described elsewhere in this text, as follows

- The mechanism of generalized restart is used to guarantee that folding always eventually applies (see section 2.7).

- Many built-in predicates, and cut in particular are supported by Mixtus (see chapter 4).

6 Examples of partial evaluation

This chapter gives some examples of what can be achieved by Mixtus. It is not a straightforward task to evaluate a program transformation system. For example, the run-time of the transformed program must be measured using a specific query, not the general query given to the partial evaluator. The run-times may then vary considerably for different specific queries, and it is not clear which specific query is more representative than the other. For this reason most examples here are given with the full input and output code so that the quality of the code generated can be studied. In the first sub-section seven test programs are presented and the performance of Mixtus is compared with five other partial evaluators.

In the following sections a small meta-interpreter, an interpreter for an assembler, an interpreter for definite clause grammar and a program using wait-declarations are presented.

6.1 Lam's test programs

The following test programs have been taken from John Lam's comparison between five different logic programming partial evaluators [Lam89]. The last two programs can be found in a paper by Lam and Kusalik [Lam90]. Although the programs are very short and simple, they illustrate several programming techniques. Each program is partially evaluated with respect to a query, and the code produced is then shown together with some comments.

6.1.1 Program 1: relative

Original program

```
relative(X, Y) :- ancestor(Z, X), ancestor(Z, Y).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
father(jap, carol).
father(jap, jonas).
father(jonas, maria).
mother(carol, paulina).
mother(carol, albertina).
mother(albertina, peter).
mother(maria, mary).
mother(maria, jose).
mother(mary, anna).
mother(mary, john).
```

Partial evaluation query

```
relative(john,Y).
```

Program produced by Mixtus

As shown in section 1.6, a predicate relating the original query to the new query is added to the produced program. The new query `relativejohn1(A)` is formed by extracting the variables of the original query `relative(john,A)`. In this case, the transformed program will use the indexing mechanism of SICStus Prolog, which only works on the first argument. Notice that the same name may occur several times, which is a consequence of the fact that Mixtus preserves the order of the solutions.

```
relative(john, A) :- relativejohn1(A).

relativejohn1(anna).
relativejohn1(john).
relativejohn1(carol).
relativejohn1(jonas).
relativejohn1(paulina).
relativejohn1(albertina).
relativejohn1(peter).
relativejohn1(maria).
relativejohn1(mary).
relativejohn1(jose).
relativejohn1(anna).
relativejohn1(john).
relativejohn1(maria).
relativejohn1(mary).
relativejohn1(jose).
relativejohn1(anna).
relativejohn1(john).
relativejohn1(mary).
relativejohn1(jose).
relativejohn1(anna).
relativejohn1(john).
```

6.1.2 Program 2: match

Original program

`match(P,T)` succeeds if the pattern P is found in T.

```
match(P,T) :- match1(P, T, P, T).

match1([], _, _, _).
match1([A|Ps], [B|Ts], P, [_|T]) :-
         A \== B, match1(P, T, P, T).
match1([A|Ps], [A|Ts], P, T) :- match1(Ps, Ts, P, T).
```

Partial evaluation query

```
match([a,a,b], T).
```

Program produced by Mixtus

```
match([a,a,b], A) :- 'match.a1'(A).

% 'match.a1'(A):-match([a,a,b],A)
'match.a1'(A) :- 'match1.a1'(A).

% 'match1.a1'(A):-match1([a,a,b],A,[a,a,b],A)
'match1.a1'([A|B]) :- a\==A, 'match1.a1'(B).
'match1.a1'([a,A|B]) :- a\==A, 'match1.a1'([A|B]).
'match1.a1'([a,a,A|B]) :- b\==A, 'match1.a1'([a,A|B]).
'match1.a1'([a,a,b|A]).
```

In the above program, Mixtus has added some comments indicating the original query corresponding to the generated predicate. The predicate `'match.a1'/1` which just consists of a single clause, can easily be removed by a post-processing stage.

Use of constraints to improve the code

The code produced above is approximately as efficient as that produced by the other partial evaluators investigated by Lam. However, the program `match` originally comes from Fujita [Fuji87], where constraints are used to produce a much better program. Mixtus can also be made to produce essentially the same program as Fujita does. First `\==/2` must be changed into the more logical `dif/2`. This will make it possible to put a constraint on a variable, which is crucial for pruning the search tree. An inspection of the program produced above shows that when `dif/2` is used it is sub-optimal to stop unfolding when an instance of a goal on the goal-stack is found. For example, the goal in the second clause of `'match1.a1'/1` is better unfolded as it only matches a single clause. The parameter "*fold_instance*" of Mixtus should therefore be unset, so that unfolding will not be stopped prematurely by folding. This means that folding will only be attempted for goals that are variable variants. Finally, to emulate the result of Fujita, only deterministic goals are allowed to be unfolded by setting the parameter maxnondeterm to 1. This will enable Mixtus to produce the following code.

```
match([a,a,b], A) :- 'match.a1'(A).

% 'match.a1'(A):-match([a,a,b],A)
'match.a1'(A) :- 'match1.a1'(A).

% 'match1.a1'(A):-match1([a,a,b],A,[a,a,b],A)
'match1.a1'([B|A]) :- dif(a, B), 'match1.a1'(A).
'match1.a1'([a|A]) :- 'match1.a2'(A).

% 'match1.a2'(A):-match1([a,b],A,[a,a,b],[a|A])
'match1.a2'([B|A]) :- dif(a, B), 'match1.a1'(A).
'match1.a2'([a|A]) :- 'match1.b1'(A).

% 'match1.b1'(A):-match1([b],A,[a,a,b],[a,a|A])
'match1.b1'([B|A]) :- dif(b, B), 'match1.a2'(B, A).
'match1.b1'([b|_]).

% 'match1.a2'(A,B):-match1([a,b],[A|B],[a,a,b],[a,A|B])
'match1.a2'(B, A) :- dif(a, B), 'match1.a1'(A).
'match1.a2'(a, A) :- 'match1.b1'(A).
```

As pointed out by Smith and Hickey [Smit90a], partial evaluation of `match/2` , where the dif-constraints are handled properly, corresponds to a derivation of the Knuth-Morris-Pratt [Knut77] string matcher.

6.1.3 Program 3: contains

Original program

String-matching: `contains(Symbol,Text)` succeeds if the `Text` contains the `Symbol`. (The arguments of `contains/2` are reversed in [Kurs88] and [Lam89].)

```
contains(Symbol,Text) :- con(Text, ([], Symbol)).

con(_, (_,[])).
con([C|Rtext], SymbInfo) :-
        new(C, SymbInfo, SymbInfoNew),
        con(Rtext, SymbInfoNew).

new(C,(Prefix, [C|RestPostfix]),(PrefixNew, RestPostfix)) :-
        append(Prefix, [C], PrefixNew).
new(C,(Prefix, [D|RestPostfix]),(PrefixNew, PostfixNew)) :-
        C \== D,
        append(Prefix, [C], H),
        append(PrefixNew, Rest, Prefix),
        append(_, PrefixNew, H),
        append(Rest, [D|RestPostfix], PostfixNew).

append([],A,A).
append([B|C],D,[B|E]) :- append(C,D,E).
```

Partial evaluation query

```
contains([a,a,b], T).
```

Program produced by Mixtus

```
contains(A, [a,a,b]) :- contains1(A).

% contains1(A):-contains(A,[a,a,b])
contains1(A) :- con1(A).

% con1(A):-con(A,([],[a,a,b]))
con1([a|A]) :- con2(A).
con1([A|B]) :- A\==a, con1(B).

% con2(A):-con(A,([a],[a,b]))
con2([a|A]) :- con3(A).
con2([A|B]) :- A\==a,(con1(B); A=a,con2(B)).

% con3(A):-con(A,([a,a],[b]))
con3([b|A]).
con3([A|B]) :- A\==b,(con1(B); A=a,con2(B); A=a,con3(B)).
```

Note that in the second clause of `con2/1`, `A\==a` followed by `A=a` cannot be simplified, because `A` might be a variable. If `\==/2` is replaced by the more logical `dif/2`, the underlined branch of the program will be removed as expected. When this is done, the code produced by Mixtus compares favorably with Kursawe's partial evaluator [Kurs88], from which the program `contains/2` was originally taken.

6.1.4 Program 4: transpose

Original program

A matrix stored as a list of lists is transposed by this program.

```
transpose(M, []) :- nullrows(M).
transpose(M1, [Row|M2]) :-
        makerow(M1, Row, M3),
        transpose(M3, M2).

makerow([], [], []).
makerow([[X|R1]|M1], [X|Row], [R1|M2]) :-
        makerow(M1, Row, M2).

nullrows([]).
nullrows([[]|M]) :- nullrows(M).
```

Partial evaluation query

```
transpose([[A,B,C,D,E,F,G,H,I],R2,R3],Res)
```

Program produced by Mixtus

Mixtus performs all possible computations and returns a unit clause.

```
transpose([[A,B,C,D,E,F,G,H,I],J,K], L) :-
         'transpose..1'(A, B, C, D, E, F, G, H, I, J, K, L).

'transpose..1'(A, B, C, D, E, F, G, H, I,
  [J,K,L,M,N,O,P,Q,R],
  [S,T,U,V,W,X,Y,Z,A1],
  [[A,J,S],[B,K,T],[C,L,U],[D,M,V],[E,N,W],
   [F,O,X],[G,P,Y],[H,Q,Z],[I,R,A1]]).
```

6.1.5 Program 5: ssupply

Original program

A data base with some access predicates is given.

```
% suppliers(s#, sname, status, city)
suppliers(s1, smith, 20, london).
suppliers(s2, jones, 10, paris).
suppliers(s3, blake, 30, paris).
suppliers(s4, clark, 20, london).
suppliers(s5, adams, 30, athens).

% parts(p#, pname, color, weight, city)
parts(p1, nut, red, 12, london).
parts(p2, bolt, green, 17, paris).
parts(p3, screw, blue, 17, rome).

%supply(s#, p#, qty)
supply(s1, p1, 300).
supply(s1, p2, 200).
supply(s1, p3, 400).
supply(s2, p1, 300).
supply(s2, p2, 400).
supply(s3, p1, 400).
supply(s4, p1, 200).
supply(s5, p1, 500).
supply(s5, p2, 400).

heavyweights(PN, PName, Color, Weight, PCity) :-
        parts(PN, PName, Color, Weight, PCity),
        Weight > 10.

london_heavyweights(PN, PName, Color, Weight) :-
        heavyweights(PN, PName, Color, Weight, PCity),
        PCity = london.
```

```
red_parts(PN, PName, Weight, PCity) :-
        parts(PN, PName, Color, Weight, PCity),
        Color = red.

london_red_parts(PN, PName, Weight) :-
        red_parts(PN, PName, Weight, PCity),
        PCity = london.

london_red_heavy_parts(PN, PName, Weight) :-
        london_red_parts(PN, PName, Weight),
        london_heavyweights(PNo, PName, Color, Weight),
        PNo = PN.

good_suppliers(SN, SName, Status, SCity) :-
        suppliers(SN, SName, Status, SCity),
        Status > 10.

best_suppliers(SN, SName, Status, SCity) :-
        good_suppliers(SN, SName, Status, SCity),
        good_cities(City),
        City = SCity.

good_cities(paris).
good_cities(london).
good_cities(hongkong).
good_cities(regina).
good_cities(saskatoon).

ssupply(SN, PN, Qty) :-
        london_red_heavy_parts(PN, PName, Weight),
        best_suppliers(SN, SName, Status, SCity),
        supply(SN, PN, Qty).
```

Partial evaluation query

```
ssupply(X,Y,Z).
```

`ssupply(X,Y,Z)` means that `X` is a supplier from London which supplies red heavy parts with part number `Y` in quantity `Z`.

Program produced by Mixtus

```
ssupply(A, B, C) :- ssupply1(A, B, C)

% ssupply1(A,B,C):-ssupply(A,B,C)
ssupply1(s1, p1, 300).
ssupply1(s3, p1, 400).
ssupply1(s4, p1, 200).
```

6.1.6 Program 6: depth

Original program

```
depth(true, 0).
depth((A, B), D) :- depth(A, D1), depth(B, D2), max(D1, D2, D).
depth(A, s(D1)) :- in_clause(A, B), depth(B, D1).

max(s(X), 0, s(X)).
max(0, s(X), s(X)).
max(s(X), s(Y), s(M)) :- max(X, Y, M).

in_clause(member(X, Xs),app(_, [X|_],Xs)).
in_clause(app([], L, L),true).
in_clause(app([X|L1], L2, [X|L3]), app(L1, L2, L3)).
```

Partial evaluation query

```
depth(member(X,[a,b,c,m,d,e,m,f,g,m,i,j]),D)
```

Program produced by Mixtus

```
depth(member(B,[a,b,c,m,d,e,m,f,g,m,i,j]), A) :-
        depthmember2(B, A).

% depthmember2(A,B):-depth(member(A,[a,b,c,m,d,e,m,f,g,m,i,j]),B)
depthmember2(a, s(s(0))).
depthmember2(A, s(s(B))) :-
        depthapp3(A, B).

% depthapp3(A,B):-depth(app(C,[A|D],[b,c,m,d,e,m,f,g,m,i,j]),B)
depthapp3(b, s(0)).
depthapp3(c, s(s(0))).
depthapp3(m, s(s(s(0)))).
depthapp3(d, s(s(s(s(0))))).
depthapp3(e, s(s(s(s(s(0)))))).
depthapp3(m, s(s(s(s(s(s(0))))))).
depthapp3(f, s(s(s(s(s(s(s(0)))))))).
depthapp3(g, s(s(s(s(s(s(s(s(0))))))))).
depthapp3(m, s(s(s(s(s(s(s(s(s(0)))))))))).
depthapp3(i, s(s(s(s(s(s(s(s(s(s(0))))))))))).
depthapp3(j, s(s(s(s(s(s(s(s(s(s(s(0)))))))))))).
```

The predicate `depthapp3/2` has not been unfolded above since the limit for unfolding of non-deterministic predicates has been reached. If the Mixtus parameter *maxnondeterm* had been set to a value higher than 10, the predicate would have been unfolded.

Another query to Mixtus

The capabilities of the partial evaluator are better illustrated if a less specific query is given to the partial evaluator: `depth(member(B,C), A)`. The resulting program then is

```
depth(member(B,C), A) :-
        depthmember1(B, C, A).

% depthmember1(A,B,C):-depth(member(A,B),C)
depthmember1(A, B, s(C)) :-
        depthapp1(A, B, C).

% depthapp1(A,B,C):-depth(app(D,[A|E],B),C)
depthapp1(A, [A|_], s(0)).
depthapp1(A, [_|B], s(C)) :-
        depthapp1(A, B, C).
```

Note that the last three clauses correspond to the clauses stored as data in `in_clause/2`.

6.1.7 Program 7: grammar

Original program

This program is a translation of a definite clause grammar for a restricted functional language.

```
expression(T,Q) --> v(T), quals(Q).
expression(T,[]) --> v(T).

v(T) --> ident(T).
v(T) --> numeric( T ).
v(T) --> builtin( T ).
v(T) --> bracket(T).
v(subscripted(T,S)) --> ident( T ), down, v(S), up.
v(T) --> leftparen, v(T), rightparen.

bracket(bracket(V,S,A)) --> leftbracket,
          v(V), semicolon,
          bracketlist(S), semicolon,
          bracketlist(A), rightbracket.
bracket(bracket(1,S,A)) --> leftbracket,
          bracketlist(S), semicolon,
          bracketlist(A), rightbracket.
bracket(bracket(1,apply(tally,A),A)) --> leftbracket,
          bracketlist(A), rightbracket.

bracketlist(subrange(T,R)) --> v(T), down, subrange( R ), up.
bracketlist(T) --> itemized(T).
bracketlist(N) --> v(N).

itemized([H|T]) --> v(H), comma, itemized(T).
itemized([T]) --> v(T).
```

```
subrange( subrange(L,U) ) --> v(L), dotdot, v(U).
subrange( subrange( open, open ) ) --> dotdot.

quals(mergequals(Q1,Q2)) --> qual(Q1), quals(Q2).
quals(Q) --> qual(Q).

qual(suchthat(L,R)) --> suchthat, v(L), equal, v(R).
qual(forall(T)) --> forall, subrange(T).
qual(when(T)) --> when, v(T).
qual(where(T)) --> where, isa(T).

isa( def( ID, VAL ) ) --> ident(ID), isa, v(VAL).

leftparen --> ['('].
rightparen --> [')'].
leftbracket --> ['['].
rightbracket --> [']'].
colon --> [':'].
semicolon --> [';'].
dotdot --> ['..'].
comma --> [','].
equal --> ['='].
where --> [where].
when --> [when].
isa --> [is].
suchthat --> [suchthat].
suchthat --> [such],[that].
forall --> [forall].
forall --> [for],[all].
down --> ['{'].
up --> ['}'].

builtin( true ) --> [true].
builtin(false) --> [false].
builtin(nil) --> [nil].
builtin('Int') --> ['Int'].
builtin('Nat') --> ['Nat'].
builtin('Bool') --> ['Bool'].

ident( D ) --> commonfunction( D ).
ident( D ) --> commonvariable( D ).
```

```
numeric( 0 ) --> [ 0 ].
numeric( 1 ) --> [ 1 ].
numeric( 2 ) --> [ 2 ].
numeric( 3 ) --> [ 3 ].
numeric( 4 ) --> [ 4 ].
numeric( 5 ) --> [ 5 ].
numeric( 6 ) --> [ 6 ].
numeric( 7 ) --> [ 7 ].
numeric( 8 ) --> [ 8 ].
numeric( 9 ) --> [ 9 ].

commonfunction( tally ) --> [tally].
commonfunction( shuffle ) --> [shuffle].
commonfunction( link ) --> [link].
commonfunction( cols ) --> [cols].
commonfunction( reshape ) --> [reshape].
commonfunction( div ) --> [div].
commonfunction( mod ) --> [mod].
commonfunction( index ) --> [index].
commonfunction( '-' ) --> ['-'].
commonfunction( '*' ) --> ['*'].
commonfunction( each ) --> [each].
commonvariable( a ) --> [a].
commonvariable( n ) --> [n].
commonvariable( f ) --> [f].
commonvariable( v ) --> [v].
commonvariable( x ) --> [x].
commonvariable( i ) --> [i].
commonvariable( shp ) --> [shp].
commonvariable( arg ) --> [arg].
commonvariable( res ) --> [res].

ac([Symbol|Rest],Symbol,Rest).
```

Partial evaluation query

```
expression(n,[],String,[])
```

Program produced by Mixtus

```
expression(n, [], A, []) :- expressionn1(A).

% expressionn1(A):-expression(n,[],A,[])
expressionn1([n]).
expressionn1(['(',n,')']).
expressionn1(['(','('|A]) :- vn2(A, []).

% vn2(A,[]):-v(n,A,[')',')'])
vn2([n,')',')'|A], A).
vn2(['('|B], A) :- vn2(B, [')'|A]).
```

Program produced with preserve_loops unset

The above program was produced with the Mixtus parameter *preserve_loops* unset, which is the default. That is, Mixtus was allowed to replace non-productive loops followed by failure with a failure, and to propagate bindings left thereby potentially replacing non-productive loops with a failure. For the other six test programs the setting of *preserve_loops* does not matter, but for this program it makes a difference. The following code is produced by Mixtus if *preserve_loops* is set.

```
expression(n, [], A, []) :- expressionn1(A).

% expressionn1(A):-expression(n,[],A,[])
expressionn1(A) :- vn1(A, _), fail.
expressionn1([n]).
expressionn1(['('|A]) :- vn1(A, B), B=[')'].

% vn1(A,B):-v(n,A,B)
vn1([n|A], A).
vn1(['('|B], A) :- vn1(B, C), C=[')'|A].
```

Program produced with preserve_loops set

Clearly this code is not as efficient as the code produced when *preserve_loops* was unset. The first clause of `expressionn1/1` will never succeed, but will only fail or loop forever, and is better removed totally. Left propagation of bindings will make the program tail-recursive.

If the query `expression(n,[],X,[])` is given to this program it will indeed loop forever, without producing any results. On the other hand, the program produced with *preserve_loops* unset will for that query correctly find exactly those instances of `X` that succeed:

```
X = [n] ? ;
X = ['(',n,')'] ? ;
X = ['(','(',n,')',')'] ? ;
etc.
```

6.1.8 Test results

The other partial evaluators in Lam's paper are Fujita's (termed *constraints* by Lam) [Fuji87], Kursawe's (*pure*) [Kurs88], *ProMix* by Lakhotia [Lakh89a, Lakh89b, Lakh89c], Levi and Sardu's (*mult*) [LeSa88] and Takeuchi's (*peval*) [Take86].

In comparison with the code produced by these partial evaluators, code produced by Mixtus performs well. Note that all code produced by Mixtus was made without any annotations and that Mixtus is careful to maintain full Prolog semantics. The following table shows the search tree size, i.e. the number of logical inferences performed, for some run-time queries. In section 6.2 an interpreter computing the search tree size is presented. All data except data for Mixtus come from [Lam90].

program	run-time query	Orig. prog. search tree size	Lam's figures best system(s)	Lam's figures search tree size	Mixtus search tree size
relative	`relative(john,peter)`	112	pure, ProMIX, peval	2	3
match	`match([a,a,b], [a,b,c,d,e,f,g,h,i,j,k, l,m,n,o,p,q,r,s,t,u, v,a,a,b,w,x,y,z])`	59	peval (using `\==`)	46 (speed-up 1.13)	56 (speed-up 1.31)
			constraints (using `dif/2`)	56 (speed-up 1.64)	57 (speed-up 1.56)
contains	`contains([a,a,b], [a,b,c,d,e,f,g,h,a,a, b,i,j,k,l,m,n,o,p,q,r, s,t,u,v,w,x,y,z])`	109	peval	9	6
transpose	`transpose([ [1,2,3,4,5,6,7,8,9], [2,3,4,5,6,7,8,9,10], [3,4,5,6,7,8,9,10,11]], Transpose)`	69	pure, ProMIX, mult, peval	2	3
ssupply	`ssupply(s3,p1,Quantity)`	25	pure, ProMIX, peval	2	3
depth	`depth(member(i,[a,b,c, m,d,e,m,f,g,m,i,j]),D)`	38	pure, mult, peval	2	4
grammar	`expression(n,[], ['(','(','(','(',n,')', ')',')',')'],[])`	160	peval	6	6

For several of the above programs, the search tree size is 3 for Mixtus, which is one more than the best of the other partial evaluators. This extra inference can be attributed to the top level clause relating the original predicate to the one produced, and could be removed at a post-processing stage of the code. Also note that there is just one extra inference per program, not per predicate.

The run-time figures are approximately proportional to the search tree size, and thus there is a considerable speed-up for most of the programs. There is one notable exception in the case of the program *match* as shown in the table. The program produced by the partial evaluator *constraints*

by Fujita needs more inferences but runs considerably faster than the program produced by *peval*. The speed-up figures are given within parentheses. Fujita's program is more deterministic and the structures unified are much smaller. As shown earlier, Mixtus is capable of producing a program very similar to Fujita's.

Prestwich [Pres90] has also used Lam's tests to benchmark his system. For the seven programs presented above he reports the following search tree sizes: 1, 51, 47, 1, 1, 1 and 6. That is, his system has a couple of inferences less than Mixtus for all programs except program 3 (*contains*), where it has a search tree that is eight times larger. The system is semi-automatic in the sense that a generalization function has to be declared.

6.2 Meta-interpreter for search tree size

The performance of the various partial evaluators in section 6.1.8 were compared using the "search tree size". Given a goal, the following interpreter will compute the search tree size.

```
searchsize(G,N) :-
        reset_counter,
        ss(G), !,
        read_counter(N).

ss(true) :- !, inc.
ss((A, B)) :- !, ss(A), ss(B).
ss(X) :- predicate_property(X,built_in), !, inc, call(X).
ss(A) :- clause(A, B), inc, ss(B).

reset_counter :- retractall(counter(_)), assert(counter(0)).
read_counter(N) :- retract(counter(N)).
inc :-  retract(counter(N)), N1 is N+1, assert(counter(N1)).
```

Using partial evaluation the interpretation overhead may be removed. Consider the following program.

```
nrev([], []).
nrev([X|Xs], R) :- nrev(Xs, R1), append(R1, [X], R).

append([], L, L).
append([H|T], L, [H|R]) :- append(T, L, R).
```

The result of partial evaluation of the query `searchsize(nrev(A,B),N)` is

```
searchsize(nrev(A,B),C) :- searchsizenrev1(A,B,C).

% searchsizenrev1(A,B,C):-searchsize(nrev(A,B),C)
searchsizenrev1(A, B, C) :-
        reset_counter,
        ssnrev1(A, B), !,
        read_counter(N).

% ssnrev1(A,B):-ss(nrev(A,B))
ssnrev1([], []) :- inc, inc.
ssnrev1([B|C], A) :- inc, ssnrev1(C, D), ssappend1(D, B, A).

% ssappend1(A,B,C):-ss(append(A,[B],C))
ssappend1([], A, [A]) :- inc, inc.
ssappend1([B|C], A, [B|D]) :- inc, ssappend1(C, A, D).
```

To improve the readability of the produced program, Mixtus was not given the definitions of `reset_counter/0`, `read_counter/1` and `inc/0` . When producing the above program, we ignored all difficulties concerning `clause/2` as discussed in section 4.2.4.9.

6.3 A simple interpreter for a simple assembler

The program given below is an interpreter in Prolog for an assembler-like language. Only a few instructions are implemented, which is sufficient to show the programming style. The instruction

```
add,src1,src2,dest
```

will add the contents of the cells at addresses `src1` and `src2` and put the result in address `dest`. The instructions `sub`, `mul`, `and` (logical bitwise and), and `shr` (shift right) are similar.

```
goto,lbl
```

will unconditionally jump to the label `lbl`, and

```
beq,src,lbl
```

jumps to `lbl` if the contents of the cell at address `src` is zero. The pseudo-instruction

```
label,lbl
```

indicates the location of the label `lbl`. Finally the instruction

```
print,src
```

prints the contents of the cell at address `src`.

The memory of the assembler-machine is implemented as a list of pairs. For example

```
[a-17, b-0, x-123]
```

means that the cell a has the contents 17. The predicate call `rd(X,XV,M)` will read the value of cell `X` from the memory `M` and return that value in `XV`. Similarly the predicate call `wr(Z,ZV,M,M2)` will take a memory `M` and replace the value of cell `Z` with the value `ZV` to obtain the new memory `M2`.

On the top-level the interpreter is called by `run(P,M)` where `P` is the program as a list of instructions and `M` is the initial state of the memory. The main predicate `i/3` of the interpreter is called with `i(P,P,M)`, where the first argument serves as a program counter with the next instruction to be executed first in the list, and the second argument always contains the whole program so that backward jumps can be made. The code has been written with simplicity rather than efficiency in mind. For example, the `goto`-instruction is executed by calling append non-deterministically to find the code following the label.

```
run(P,M) :- i(P,P,M).

i([],_,_).
i([add,X,Y,Z|R],P,M) :-rd(X,XV,M), rd(Y,YV,M), ZV is XV+YV,
                        wr(Z,ZV,M,M2), i(R,P,M2).
i([sub,X,Y,Z|R],P,M) :-rd(X,XV,M), rd(Y,YV,M), ZV is XV-YV,
                        wr(Z,ZV,M,M2), i(R,P,M2).
i([mul,X,Y,Z|R],P,M) :-rd(X,XV,M), rd(Y,YV,M), ZV is XV*YV,
                        wr(Z,ZV,M,M2), i(R,P,M2).
i([and,X,Y,Z|R],P,M) :-rd(X,XV,M), rd(Y,YV,M), ZV is XV/\YV,
                        wr(Z,ZV,M,M2), i(R,P,M2).
i([shr,X,Y,Z|R],P,M) :-rd(X,XV,M), rd(Y,YV,M), ZV is XV>>YV,
                        wr(Z,ZV,M,M2), i(R,P,M2).
i([label,_|R],P,M) :- i(R,P,M).
i([goto,L|_],P,M) :- append(_,[label,L|R],P), i(R,P,M).
i([print,X|R],P,M) :- rd(X,XV,M), write(XV), i(R,P,M).
i([beq,X,L|R],P,M) :- rd(X,V,M),
                      (V=0 -> i([goto,L|R],P,M);
                              i(R,P,M)).

wr(X,V,[X-_|Mem],[X-V|Mem]).
wr(X,V,[C|Mem],[C|Mem2]) :- wr(X,V,Mem,Mem2).

rd(X,V,Mem) :- member(X-V,Mem,_).

member(X,[X|R],R).
member(X,[Y|R],[Y|R2]) :- member(X,R,R2).

append([], L, L).
append([H|T], L, [H|R]) :- append(T, L, R).
```

An interpreter for an assembler-like language

An assembler program which prints a^b if x=1 and b≥0 is shown below. The initial values of the memory cells for a, b and x are given in a list which is an argument to the predicate

`a_exp_b/1`. Before calling `run(P,M)`, the whole program is put in the list `P`, and the initial memory configuration is put in `M`.

```
a_exp_b([A,B,X]) :-              % comments in C
          %X = 1,                 % x = 1
          P =
          [
          label,start,            % while (b != 0)
          beq,b,end,              %     {
          label,while_even,       %
          and,b,one,temp,         %     while ((b & 1) == 0)
          beq,temp,even,          %           {
          goto,not_even,
          label,even,
          shr,b,one,b,            %           b = b>>1;
          mul,a,a,a,              %           a = a*a
          goto,while_even,
          label,not_even,         %           }
          sub,b,one,b,            %     b = b-1
          mul,x,a,x,              %     x = x*a
          goto,start,             %     }
          label,end,
          print,x],               % print x
          Mem=[a-A,b-B,x-X,temp-_,one-1],
          run(P,Mem).
```

An assembler program which prints a^b *if* $x=1$ *and* $b \geq 0$

The result of partial evaluation of the query `a_exp_b(A)` with respect to the interpreter and the assembler program is shown below. The comments are generated by Mixtus and show the relation between the newly generated predicate and the predicate call that caused it to be generated. The very long comment for `' i.beq2'/3` shows that this predicate corresponds to the situation when the instruction "`beq,temp,even`" is just to be executed. That predicate is also deterministic and tail-recursive, which means that the recursive call will be compiled to just a jump to the beginning of the code. The code produced by a Prolog compiler should therefore be rather close the original assembler program!

```
a_exp_b(A) :-
        a_exp_b1(A).

% a_exp_b1(A):-a_exp_b(A)
a_exp_b1([A,B,C]) :-
        (   B=0 ->
            write(C)
        ;   D is B/\1,
            'i.beq2'(A, B, C, D)
        ).

% 'i.beq2'(A,B,C,D):-
% i([beq,temp,even,goto,not_even,label,even,shr,b,one,b,
% mul,a,a,a,goto,while_even,label,not_even,sub,b,one,b,
% mul,x,a,x,goto,start,label,end,print,x],
% [label,start,beq,b,end,label,while_even,and,b,one,temp,
% beq,temp,even,goto,not_even,label,even,shr,b,one,b,
% mul,a,a,a,goto,while_even,label,not_even,sub,b,one,b,
% mul,x,a,x,goto,start,label,end,print,x],
% [a-A,b-B,x-C,temp-D,one-1])
'i.beq2'(A, B, C, D) :-
        (   D=0 ->
            E is B>>1,
            F is A*A,
            G is E/\1,
            'i.beq2'(F, E, C, G)
        ;   H is B-1,
            I is C*A,
            (   H=0 ->
                write(I)
            ;   J is H/\1,
                'i.beq2'(A, H, I, J)
            )
        ).
```

Code 1: result from partial evaluation of the query `a_exp_b(X)`

The query `a_exp_b([3,19,1])` corresponds to computing 3^{19}. Using the interpreter and the assembler program directly this computation takes 21ms (on a SPARCstation 1), whereas using the program produced by partial evaluation it takes 0.89ms. Thus in this case partial evaluation will speed up the computation more that 20 times.

The assembler program above does not compute the value of a^b by multiplying by a b times, but rather by repeated squaring and multiplication. For example, a^5 is not computed by the direct product $a \cdot a \cdot a \cdot a \cdot a$ but rather by the expression $a(a^2)^2$. A classical exercise in partial evaluation is to compute a^b given that b=5 [Ersh80]. If Mixtus is given the query `a_exp_b([A,5,1])` the following program is produced.

```
a_exp_b([A,5,1]) :-
        'a_exp_b.1'(A).

% 'a_exp_b.1'(A):-a_exp_b([A,5,1])
'a_exp_b.1'(A) :-
        B is A,
        C is A*A,
        D is C*C,
        E is B*D,
        write(E).
```

Code 2: result from partial evaluation of the query `a_exp_b([A,5,1])`

There are two ways to produce the above program. Either by a one-step procedure where the query `a_exp_b([A,5,1])` is partially evaluated given the interpreter and the assembler program. Alternatively a two-step procedure is employed where the partial evaluator is called twice: first with the more general query `a_exp_b(X)` producing Code 1 above, and then using the query `a_exp_b([A,5,1])` on Code 1 to produce Code 2. It is a remarkable fact, as shown by the table below, that the two-step procedure is much more efficient. This is because the optimizations depending on b being 5 are carried out on a much simpler program (code 1) than the whole interpreter.

method	*query*	*program*	*result*	*partial evaluation time*
one-step	`a_exp_b([A,5,1])`	interpreter + assembler prog.	code 2	455s
two-step	`a_exp_b(X)`	interpreter + assembler prog.	code 1	167s
	`a_exp_b([A,5,1])`	code 1	code 2	2s

This appears to be an optimization technique that is frequently applicable. Given a query which is to be partially evaluated, if the program produced from partial evaluation by a more general query produces a much smaller program than the original one, it might be advantageous to use the two-step technique. In certain cases, even a multi-step technique may yield even better results. Mixtus does not attempt to automatize this optimization.

6.4 Definite Clause Grammar

Most Prolog systems, including SICStus, have built-in support for writing definite clause grammars [PeWa78], which is a convenient way to express grammar rules of the form

head `-->` *body*.

which means that *body* is a possible form for *head*. Usually these rules are considered “syntactic sugar” and are converted to ordinary clauses when read in. An alternative approach is to write an interpreter for these rules:

```
:- op(1200,xfx,--->). % used instead of --> to avoid confusion

dcg(G,_,_) :- var(G), !,
         write('error: variable given as a goal'), nl, fail.
dcg({G},A,B) :- !, G, A=B.
dcg((G1,G2),A,C) :- !, dcg(G1,A,B), dcg(G2,B,C).
dcg((G1;Else),A,C) :- (nonvar(G1), G1=(Test->Then)), !,
         (dcg(Test,A,B)->
             dcg(Then,B,C)
         ;   dcg(Else,A,C)).
dcg((G1;G2),A,B) :- !, (dcg(G1,A,B); dcg(G2,A,B)).
dcg(G,A,B) :- islist(G), !, append(G,B,A).
dcg(\+G,A,B) :- !, \+dcg(G,A,B).
dcg((G1->G2),A,C) :- !, (dcg(G1,A,B)->dcg(G2,B,C)).
dcg(G,A,B) :- (G ---> Body), dcg(Body,A,B).
dcg(G,A,B) :- G=..Glist, append(Glist,[A,B],G2list),
              G2=..G2list, G2.

append([], L, L).
append([H|T], L, [H|R]) :- append(T, L, R).

islist(X) :- X==[]; functor(X,F,2), F='.', arg(2,X,X2), islist(X2).
```

The above interpreter can be partially evaluated with respect to some rules. The following grammar for regular expressions will illustrate the method.

```
:- op(400,xf,'*'). % postfix
:- op(100,fx,`).   % prefix

regexpr((A;B)) ---> (regexpr(A); regexpr(B)).
regexpr((A,B)) ---> regexpr(A), regexpr(B).
regexpr(_*) ---> [].
regexpr(X*) ---> regexpr(X),regexpr(X*).
regexpr(`X) ---> [X].
regexpr(any) ---> [_].
```

These rules are to be understood as follows. A list matches a regular expression depending on the form of the expression:

- (A;B) matches if A or B matches
- (A,B) matches if A followed by B matches
- A* matches the empty list or a list where A matches one or several times in succession
- `X matches the terminal symbol X
- any matches any single terminal symbol

Mixtus will produce the following code for the query `dcg(regexpr(C), A, B)`.

```
dcg(regexpr(C), A, B) :- dcgregexpr1(C, A, B).

% dcgregexpr1(A,B,C):-dcg(regexpr(A),B,C)
dcgregexpr1((C;_), A, B) :-
        dcgregexpr1(C, A, B).
dcgregexpr1((_;C), A, B) :-
        dcgregexpr1(C, A, B).
dcgregexpr1((C,D), A, B) :-
        dcgregexpr1(C, A, E),
        dcgregexpr1(D, E, B).
dcgregexpr1(_*, A, A).
dcgregexpr1(C*, A, B) :-
        dcgregexpr1(C, A, D),
        dcgregexpr1(C*, D, B).
dcgregexpr1(`A, [A|B], B).
dcgregexpr1(any, [_|A], A).
```

A dummy definition of `regexpr/3` is needed to avoid expansion in the last clause of `dcg/3`. Otherwise calls would have been generated to the supposedly external predicate `regexpr/3`, which would have complicated the above code.

As the code produced by SICStus definite clause expansion mechanism is quite similar, partial evaluation here provides an alternative way of doing the transformation. Pereira and Shieber also have an interpreter for definite clause grammars which they partially evaluate and achieve similar results [PeSh87].

We are now able to go one step further, and partially evaluate the more specific query `dcg(regexpr((`1;`2,`3*)*), A, [])` which produces:

```
dcg(regexpr((`1;`2,`3*)*), A, []) :-
        'dcgregexpr*;`11'(A).

% 'dcgregexpr*;`11'(A):-dcg(regexpr((`1;`2,`3*)*),A,[])
'dcgregexpr*;`11'([]).
'dcgregexpr*;`11'([1|A]) :-
        'dcgregexpr*;`11'(A).
'dcgregexpr*;`11'([2|A]) :-
        'dcgregexpr*`31'(A, B),
        'dcgregexpr*;`11'(B).

% 'dcgregexpr*`31'(A,B):-dcg(regexpr(`3*),A,B)
'dcgregexpr*`31'(A, A).
'dcgregexpr*`31'([3|B], A) :-
        'dcgregexpr*`31'(B, A).
```

A variable as a goal

A variable may be a goal in the body of a definite clause grammar rule, for example

```
test(A) --> [before], A, [after].
```

SICStus expands this clause to

```
test(A, B, C) :-
        'C'(B, before, D),
        phrase(A, D, E),
        'C'(E, after, C).
```

During execution it is expected that this goal `A` is instantiated, and `phrase/3`, which is a built-in predicate, will handle its first argument according to the rules for a grammar body. Partial evaluation of the query `dcg(test(X),A,B)` which uses the DCG interpreter above will produce similar code.

```
dcg(test(C), A, B) :-
        dcgtest1(C, A, B).

% dcgtest1(A,B,C):-dcg(test(A),B,C)
dcgtest1(A, [before|C], B) :-
        dcg1(A, C, D),
        D=[after|B].
```

The code for `dcg1/3` is not shown as it is almost identical to the interpreter `dcg/3` above.

6.5 Wait-declarations

The following program mainly illustrates how wait-declarations are handled in Mixtus, but it also shows that cut can be executed and eliminated. The predicate `f/3`, as shown below, implements a monotonic function f which is used when the least fixed point is to be found in a set of equations. The details on how this is done can be found in [Sahl90a].

```
f(X,Y,V) :- f(X,Y,V,0,0,0).

f(X,Y,V,XInext,YInext,VI) :-
          t(XInext,YInext,VInext),
          increment_V(VI,VInext,V,Vnext),
          fx(X,Y,Vnext,XInext,YInext,VInext),
          fy(X,Y,Vnext,XInext,YInext,VInext).

fx(X,Y,V,XI,YI,VI) :- max(M), XI<M, t(M,M,Max), VI<Max, !,
          XI1 is XI+1, x_incremented(X,Y,V,XI1,YI,VI).
fx(_,_,_,_,_,_).

fy(X,Y,V,XI,YI,VI) :- max(M), YI<M, t(M,M,Max), VI<Max, !,
          YI1 is YI+1, y_incremented(Y,X,V,XI,YI1,VI).
fy(_,_,_,_,_,_).

:- wait x_incremented/6.
x_incremented(s(X),Y,V,XI,YI,VI) :- f(X,Y,V,XI,YI,VI).
:- wait y_incremented/6.
y_incremented(s(Y),X,V,XI,YI,VI) :- f(X,Y,V,XI,YI,VI).

increment_V(VInext,VInext,Vnext,Vnext) :- !.
increment_V(VI,VInext,s(V),Vnext) :-
          VI1 is VI+1, increment_V(VI1,VInext,V,Vnext).
```

That code calls the predicates `t/3` and `max/1` describing the monotonic function f:

```
t(0,0,0).  t(0,1,1).  t(0,2,1).
t(1,0,0).  t(1,1,1).  t(1,2,2).
t(2,0,2).  t(2,1,2).  t(2,2,2).
max(2).
```

The result of partial evaluation of the query `f(X,Y,Z)` is the following (where the predicates produced by Mixtus have been renamed for readability).

```
f(X, Y, V) :- fx10(X, Y, V), fy01(Y, X, V).
:- wait fx01/3.
fx10(s(X), Y, V) :- fx20(X, Y, V), fy11(Y, X, V).
:- wait fx20/3.
fx20(s(X), Y, s(s(V))).
:- wait fy11/3.
fy11(s(Y), X, s(V)) :- fx21(X, Y, V), fy12(Y, X, V).
:- wait fx21/3.
fx21(s(X), Y, s(V)).
:- wait fy12/3.
fy12(s(Y), X, s(V)).
:- wait fy10/3.
fy01(s(Y), X, s(V)) :- fx11(X, Y, V), fy02(Y, X, V).
:- wait fx11/3.
fx11(s(X), Y, V) :- fx21(X, Y, V), fy12(Y, X, V).
:- wait fy02/3.
fy02(s(Y), X, V) :- fx12(X, Y, V).
:- wait fx12/3.
fx12(s(X), Y, s(V)).
```

Partially evaluated program

Using `f/3`, which implements the monotonic function f, the least fixed point of the equations $x_1 = f(x_1,x_2)$, $x_2 = f(x_1,1)$ is found by

```
?- f(X1,X2,X1), f(X1,s(_),X2).
```

which returns

```
X1=s(_), X2=s(_)
```

that is, $x_1=1$ and $x_2=1$. The original program needs 3.4 ms to answer that query on a SPARCstation 1, whereas the partially evaluated program needs just 0.5 ms.

7 Conclusions

In this chapter the main contributions of the thesis are first summarized. A brief discussion of related work then follows. Then some properties of the program produced by the partial evaluator are discussed, and also the efficiency of the partial evaluator itself. A discussion of the suitability of applying partial evaluation to Prolog programs follows. Finally, some of the remaining problems and research issues are mentioned.

7.1 Contributions

The main contributions are the following. A new definition of partial evaluation for full Prolog is given, which is adapted to the non-logical features of Prolog. Less strict versions of the definition are presented which allow some non-productive loops to be removed, and it is argued that this relaxation is reasonable both from a procedural and a declarative point of view.

A partial evaluator for full Prolog, named Mixtus, is presented. The most distinctive feature of Mixtus is the automatic handling of unfolding, which is based on a loop prevention mechanism and a mechanism for generalized restart using anti-unification. A distinction is made between loop detection and loop prevention, and several algorithms for the latter are described and proven correct.

To improve the treatment of a number of built-in predicates, cut in particular, a module is presented which estimates the number of solutions a goal may produce. Precise definitions are given of what is meant by logical, propagation sensitive and side-effect predicates. Based on these definitions further optimizations are allowed while retaining the full Prolog semantics.

Attention is also given to the built-in predicates, cut in particular, where the concept of input variables increases the opportunities for a cut to be performed during partial evaluation. In the if-then-else construct, the test may be non-deterministic and yet the bindings generated there may be propagated to the then-part. Negation as failure is fully supported, since it is based on the cut. The built-in predicates `findall/3`, `bagof/3` and `setof/3` are handled properly even if some potential solutions fail due to “bound” variables being instantiated at run-time. Mixtus has limited support for frozen goals and dif-constraints. Programs that create cyclic structures are handled correctly, but not efficiently, since Mixtus avoids creating cyclic structures. In appendix A, an algorithm for doing anti-unification on cyclic structures is presented. In chapter 5 an elementary partial evaluator is developed from a meta-interpreter. The full partial evaluator, named Mixtus, has been made available for research and educational purposes.

7.2 Related work

We don't view Prolog primarily as a logic programming language but as a procedural language, so side-effects and the order of the solutions produced by a program are of central importance. Thus, we cannot base our methodology on the declarative semantics and can neither use the unfolding transformations as described in [Tama84] nor the definition of partial deduction in [Lloy87, Komo89]. Instead we provide alternative definitions, suitable for full Prolog.

We share this view of Prolog with a number of researchers, but unfortunately a number of publications have been unclear as to whether they are focused on full Prolog or rather on the declarative semantics. Venken was the first to publish a paper where the non-logical features of Prolog were discussed in more detail [Venk84]. One of the main issues in handling full Prolog is the treatment of cut, and a number of publications have discussed this issue [Venk88, LeSa88, Bugl89, Lakh90, Owen89]. We now believe that the treatment of cut in partial evaluation is all but solved, as there are very few instances in Mixtus where it still causes problems.

Partial evaluation is probably most widely used for removing the parsing overhead in interpreters, and successful examples of such use are discussed in [Lakh89a, Lakh89b, Lakh90, LeSa88, Owen89, Ster86a, Take86]. van Harmelen discusses the limitations of partial evaluation, in particular in relation to meta-programming in [Harm89], where he asserts that the interpreter must not depend on too much dynamic information for the partial evaluation to be successful. We have arrived at a similar conclusion as Mixtus when given insufficient information just returns the program to be partially evaluated unchanged.

Partial evaluation and Prolog with constraints is another interesting research issue. Mixtus can handle dif-constraints reasonably, but a more elaborate framework for handling constraints has been laid out by Smith and Hickey [SmHi90, SmHi91]. It seems partial evaluation and constraints mix very well.

In Mixtus, the decisions when to unfold are based on criteria for loop prevention and folding. An alternative approach is to analyze the program, and to see whether it is suitable and safe, with respect to termination, to unfold. Fujita suggests the use of abstract interpretation for such an analysis [Fuji88a, Mell86].

To be able to use all the Futamura projections [Futa82, Fuji88b], self-applicability of a partial evaluator is necessary. Self-applicable partial evaluators for pure Prolog have been published [Fuji88b, Full88]. By keeping the partial evaluators very small, although non-trivial, self-applicability is achieved. Termination of the partial evaluation procedure is not guaranteed in [Full88], whereas in [Fuji88b] extensive annotations are required by the user. Self-applicability of Mixtus is discussed in section 7.4.3.

As it is not immediately clear what makes one partial evaluator better than another, some sort of comparison is needed. Lam and Kusalik have conducted such comparisons [Lam89, Lam90] where a number of partial evaluators are compared [Fuji87, Kurs88, Lakh89a, Lakh89b,

Lakh89c, LeSa88, Take86]. We claim that Mixtus, although capable of handling full Prolog, performs well compared to those systems as shown in section 6.1.8.

7.3 Properties of the program produced

In this section the most important properties of the program produced are discussed, i.e. its correctness (the program does the same as the original program), performance (it does it faster) and size (it is not too large).

7.3.1 Correctness of the transformations

It is of fundamental importance that the transformations performed by the partial evaluator are correct, i.e. the resulting program must satisfy the conditions given in the definition of partial evaluation. Correctness has not been formally shown for all transformations presented, but rather it is hoped that the reader is convinced of the correctness informally. For a practical system it is also important that the implementation is correct. However, it does not seem feasible to prove the correctness of the implementation, so the standard fallback method is used: verification. Mixtus has been tested on a large number of programs and these have been found by inspection and execution to be correctly transformed.

7.3.2 Performance improvement

To show that the program transformations actually improve performance would be much harder than to show correctness. First a model of the execution must be formed which includes estimates of the complexity of the operations. Then it must be shown that each transformation does not cause the performance to deteriorate. The main problem here is to choose and construct an appropriate model.

One method is simply to count the number of procedure calls (or inferences). Unfolding for Prolog, in contrast to logical unfolding, always decreases the number of inferences in the resulting program. The disjunction introduced by unfolding for Prolog has, however, a similar overhead as a procedure call in SICStus Prolog. The other transformations, folding and definition, keep the number of inferences constant. However, one extra inference is added when the top-level predicate transforming mixed computation into partial evaluation is introduced. Thus it is guaranteed that the number of inferences is increased by at most one in a program produced by Mixtus. But just counting the number of inferences does not model the actual execution cost very well. For example, the program `match/2` in section 6.1.2 is partially evaluated in two ways, and it is not the version with the fewest number of inferences that executes fastest.

A more complex model would be very cumbersome to handle. To be perfect, the model should incorporate all the details of the implementation, which would make it far too complicated to handle formally.

Instead an informal approach has been taken for Mixtus, where for each transformation introduced it is argued that it is "reasonable". In practice it is not necessary that all transformations are beneficial or neutral; for a large program it is the combined result of all transformations applied that is of interest. By applying Mixtus to a large number of programs it has been found that performance practically never deteriorates, and there are seldom any simple transformations that could further improve performance. Of course there are still several cases where the system could be further significantly improved.

For interpreters, which is the most suitable class of programs to be partially evaluated, it seems that Mixtus usually does what is expected. That is, the parsing layer of interpretation is removed.

7.3.3 Size of the generated program

Mixtus does not contain any global control which prevents the generated program from growing unwieldy in size. Instead there are some heuristics used locally. For example, a code explosion may occur when a conjunction of calls to non-deterministic predicates are expanded, and all possible combinations are generated. This situation is prevented, as Mixtus has a parameter *maxnondeterm*, which limits the number of clauses a predicate may have if it is to be allowed to be unfolded.

Other kinds of code explosion are also possible. Each different predicate call pattern generates a new predicate specialized for that pattern. It is not difficult to construct programs which contain a very large number of call patterns. Partial evaluation of such programs may produce very large programs.

Another problem with abnormal size of programs is that the limits of either the implementation or the computer may be reached.

SICStus Prolog, like most other implementations, has a limit on the number of variables in a clause (max 255), arguments of a compound term (max 255), and number of clauses of a predicate (65535). For a handwritten program it is highly unlikely that any of these limits will be reached, but for an automatically generated program this may happen more easily.

For a small computer, the size of the new program may be larger than it can handle. For a computer with virtual memory, a large program may run slower if it does not fit into the physical memory. For a computer with cache memory, the original program may have been small enough to fit the cache, whereas the newly produced program may be too large. As the main reason for doing partial evaluation is to speed up execution, and a large program size may counteract that effort, this might be a problem.

Normally, however, the size of the generated program is not a big problem. Typically ordinary Prolog programs will grow by a factor 3–4. Partial evaluation of interpreters applied to source programs will often produce resulting programs much smaller than the interpreter.

7.4 Efficiency of the partial evaluator

The space and time efficiency of Mixtus is crucial for making it a practical tool. We have not found the memory consumption to be much of a problem, so we will here only discuss the execution time of Mixtus. A distinction in the discussion below is made between predictable and acceptable execution time. Auto-projection is also discussed as a method for reducing the execution time.

7.4.1 Predictable execution time of the partial evaluator

Although it is guaranteed that the partial evaluation will terminate, for practical purposes it must also terminate within a reasonable time limit. But it is of equal practical importance that the produced program is sufficiently efficient, and it takes more time to produce a better program. A compromise has to be made, and for Mixtus the emphasis has been put on producing efficient code. Although usually acceptable, the partial evaluation time is much longer than the time to compile a program.

If partial evaluation is ever to become a natural part of the program development cycle (edit-compile-test), the programmer must be able to predict the partial evaluation time. The programmer may set some general parameters to speed up the partial evaluation process, at the expense of the quality of the code. For example, Mixtus has a parameter named *quickpe* which simplifies the work of the partial evaluator. When the parameter is set, if folding is not possible for the first recursive call of a predicate, a generalized restart will be immediately triggered. This means that all goals will be unfolded at most once, and the partial evaluation will be done much quicker. In some cases, such a simplified partial evaluation might be sufficient, but it will generally fail to remove the overhead of an interpreter.

One way to limit the execution time of Mixtus would be to incorporate a mechanism which adjusts those parameters during execution if a predetermined time-limit is approached. This idea remains to be investigated.

Of course the partial evaluation time varies with the size of the program to be analyzed, but the time is even more dependent on the kind of program that is partially evaluated. The execution time of the partial evaluator is therefore very hard to predict. This is not very surprising as the partial evaluator can also be used as an ordinary Prolog program interpreter, and the execution time of a Prolog program cannot be predicted in a simple way. Indeed, it is not even decidable whether the Prolog program will terminate. Although it is guaranteed that Mixtus will always terminate, the upper bounds for execution time are far too high to be of practical interest. Normally, however, the execution time of Mixtus seems roughly proportional to the length of the program to be partially evaluated.

7.4.2 Acceptable execution time

Mixtus is implemented as a meta-interpreter which uses the logical variables of Prolog and backtracking. As this method is comparatively close to ordinary Prolog execution, the method is fairly efficient, especially with regard to memory consumption. However, Mixtus is several orders of magnitude slower than SICStus when executing Prolog programs, as could be expected for a meta-interpreter. Without powerful workstations and efficient Prolog implementations, work on partial evaluation for Prolog might not have been practical. One general optimization has been found to be invaluable for large programs: as mentioned in section 2.8, if a goal has been partially evaluated, the result is saved, and just a lookup is needed to partially evaluate the same goal again.

A very elegant way to improve performance of a partial evaluator is to apply it to itself. For this, it is necessary that it is self-applicable.

7.4.3 Selfapplicability of Mixtus

If a partial evaluator is an auto-projector, i.e. self-applicable, it is possible to automatically create a compiler given an interpreter or to create a general compiler-compiler. These are called the Futamura projections [Futa82, Fuji88b]. Since many partial evaluators are written in the language that they analyze, they are technically self-applicable. In practice this is not sufficient; the partial evaluator must also be able to handle in a satisfactory way the subset of the language it is written in. The fundamental problem for achieving self-applicability is that when the partial evaluator becomes more powerful and complex it also becomes much harder to partially evaluate.

In principle Mixtus is self-applicable, since no constructs are used that it cannot handle correctly. However, there is one important built-in predicate that is not efficiently handled and which is central to the execution of Mixtus: `assert/1`. No call to `assert/1` in the program to be partially evaluated is ever executed by Mixtus. This means that the central mechanisms, such as storing generated code, will not be executed by Mixtus, and the quality of the code will be very poor.

One way to efficiently support self-application is of course to extend Mixtus to handle `assert/1` efficiently. It is however unclear how this should be done, and remains a subject for further study how to accomplish this.

Another more easily accessible way to support self-application is to rewrite Mixtus so that `assert/1` is not used at all. The only essential use of `assert/1` is to communicate results from one non-deterministic branch to another. One approach to abolishing `assert/1` would be never to rely on the built-in backtracking of Prolog, but to code the backtracking explicitly. A ground representation of variables would then possibly also be needed, as well as explicit code for all unifications. The partial evaluator would then execute slower than Mixtus, and besides the comparative simplicity of the present approach, this is one reason why efficient self-applicability

has not been attempted. On the other hand, the gains from self-applicability might outweigh the disadvantages, so this remains an interesting path for future research. All methods used in Mixtus for loop prevention, generalized restart, treatment of cut etc. could be carried over without change to an auto-projector.

7.5 Suitability of Prolog for partial evaluation

Prolog is very well suited to partial evaluation since partial structures, i.e. terms containing Prolog variables, are supported directly by the language. In some cases ordinary Prolog execution corresponds to partial evaluation in other languages. For example, the partially instantiated goal `member(f(X),[g(X),Y])` will result in `Y=f(X)`. Another important feature, which simplifies program transformation, is the single assignment property. This means that the value of a variable once assigned never changes, except for backtracking. Another important consequence of this is that two variables once unified, can never become different again.

Mixtus is careful to maintain full Prolog semantics and the program produced will never change the order of the solutions. This restriction makes several "logical" optimizations impossible, such as deleting multiple solutions. Yet, the examples in chapter 6 show that very few opportunities for optimization are actually lost by keeping full Prolog semantics. In one respect a compromise has been made that seems important: Mixtus allows some non-productive loops to be removed.

In full Prolog there are a few features that are very cumbersome for partial evaluation. The if-then-else-construct has a syntax poorly suited to program transformation, as mentioned in section 4.2.2.2. The `clause/2` predicate has a semantics that makes it impossible to unfold (see section 4.2.4.9). Although `dif/2` fits very well into the partial evaluation framework, `freeze/2` seems to be very hard to handle in general as the frozen predicate may be woken up at any unification. The treatment of the predicates `findall/3`, `bagof/3` and `setof/3` becomes very complex, mainly due to the lack of restricted scope for quantified variables. Although Mixtus accepts side-effect predicates such as `assert/1` and `retract/1`, these are never executed, but rather kept as they are.

7.6 Future work and research problems

Several possible improvements have not (yet?) been incorporated into Mixtus. As already mentioned, Mixtus never performs folding on composite goals. Important transformations, such as loop merging, are therefore not possible. At present, it is unclear whether these transformations are possible to automatize and whether they are feasible for full Prolog.

Even if a goal is not unfolded, it could be arranged that some bindings are propagated. The "common feature extraction" of Nakagawa [Naka87] and "constraint lifting" [SmHi90] are such optimizations, which are not performed by Mixtus, and could possibly be added. For full Prolog,

we have noticed a need for propagating bindings only to the right of the call for some predicates that have not been unfolded.

Other simple transformations have been contemplated, but have not been implemented, as they do not seem so important:

- Removal of unnecessary arguments in predicates. The method described in [Sahl90a] could be used for this.
- Unfolding of trivial predicates, i.e. predicates with just a single clause and a single goal in the body. These may occur as unfolding is prevented for recursive predicates.
- Common subterm elimination. A generalization of this is to avoid recreating a structure repeatedly in a loop (see section 2.4.4).

A partial evaluator can always be improved. The emphasis of Mixtus is that it is *automatic* and for *full* Prolog. Some remaining problems in these two main issues are discussed below.

Full

It is a very special challenge to carry out partial evaluation on an existing language such as Prolog, and not on a language created specifically for the partial evaluator, which has often been the case. On the whole, Prolog seems well suited to partial evaluation, but as discussed above there are some weak points in the design of the language with respect to partial evaluation. The main predicates, which cannot be simplified by Mixtus are `assert/1` and `retract/1`. It is a question for future research whether it is feasible to execute them by a partial evaluator.

Automatic

If possible, Mixtus will always fold instead of unfold. This choice is a part of the mechanism for guaranteeing termination of execution and is not easily changed. In some cases, as mentioned in section 2.9 unfolding is preferable to folding. It remains to characterize these cases, and extend the mechanisms of Mixtus so that termination is still guaranteed. Abstract interpretation could possibly be used to analyze the program to find sharper criteria for folding and unfolding.

The main unsolved (and unsolvable) problem in Mixtus is to create a perfect loop prevention mechanism. Our experiments, however, indicate that the simple tests in this thesis usually are quite adequate.

References

This list of references only includes books and papers that are directly referenced in the text. For an excellent annotated bibliography on partial evaluation see [Sest88].

Abra89 Abrahamson, H. and Rogers, M. H., *Meta-Programming in Logic Programming*, MIT Press, 1989

AlKa90 Ali, K.A.M. and Karlsson R., The Muse Or-Parallel Prolog Model and its Performance, in Proceedings of the 1990 North American Conference on Logic Programming, MIT Press, pp. 757–776, 1990

Beck76 Beckman, L., Haraldson, A., Oskarsson, Ö. and Sandewall, E., A Partial Evaluator, and its Use as a Programming Tool, *Artificial Intelligence Journal 7*, pp. 319–357, 1976

Benk89 Benkerimi, K. and Lloyd, J.W. A Procedure for the Partial Evaluation of Logic Programs, Tech. Rept. TR-89-04, Computer Science Department, University of Bristol, May 1989

BeWe90 Berlin, A. and Weise, D., Compiling Scientific Code Using Partial Evaluation, IEEE Computer, December 1990

Bjør88 *Partial Evaluation and Mixed Computation*, proceedings from workshop in Gammel Avernæs, Denmark, eds. Bjørner, D., Ershov, A. P., and Jones, N. D., North-Holland, October 1987

Bol90 Bol, R.N., Towards More Efficient Loop Checks, in Proceedings of the 1990 North American Conference on Logic Programming, MIT Press, pp. 465–479, 1990

Bond90 Bondorf, A., Automatic Autoprojection of Higher Order Recursive Equations, ESOP'90. 3rd European Symposium on Programming, Copenhagen, Denmark, (Lecture Notes in Computer Science, vol. 432, Springer-Verlag), May 1990

BuDa77 Burstall, R.M. and Darlington, J., A Transformation System for Developing Recursive Programs, *Journal of ACM*, 24, 1977

Bugl89 Bugliesi, M. and Russo, F., Partial Evaluation in Prolog: Some Improvements about Cut, in Proceedings of the 1989 North American Conference on Logic Programming, MIT Press

Carl88 Carlsson, M. and Widén, J., SICStus Prolog User's Manual, Research report R88007, SICS, Sweden, 1988

Cava89 Cavalieri, M., Lamma, E., Mello, P. and Natali, A., Meta-Programming in Prolog Through Direct Introspection: A Comparison with Meta-Interpretation Techniques, in [Abra89] pp. 399–415, 1989

ClMe84 Clocksin, W.F. and Mellish, C.S., *Programming in Prolog*, 2nd Edition, Springer-Verlag, New York, 1984

Colm73 Colmerauer, A., Kanoui, H., Roussel, P. and Pasero, R., Un système de communication homme-machine en français, Groupe de recherche en Intelligence Artificielle, Université d'Aix-Marseille, Luminy, 1973

Ersh77 Ershov, A., On the Partial Computation Principle, *Information Processing Letters*, 6(2):38–41, April 1977

Ersh88a Ershov, A. and Jones, N. D., Two Characterizations of Partial Evaluation and Mixed Computation, in [Bjør88], pp. xv–xxi, 1988

Ersh88b Ershov, A., Opening Key-Note Speech, in [Bjør88], pp. xxiii–xxix, 1988

Fuch88 *New Generation Computing*, 6 (1988), Special issue: selected papers from the Workshop on Partial Evaluation and Mixed Computation, ed. K. Fuchi, Ohmsha. Ltd/Springer-Verlag, 1988

Fuji87 Fujita,H., An Algorithm for Partial Evaluation with Constraints, ICOT Technical Memorandum: TM-0367, 1987

Fuji88a Fujita, H., Abstract Interpretation and Partial Evaluation of Prolog Programs, Technical Memorandum TM-0484, ICOT, 1988

Fuji88b Fujita, H. and Furukawa, K., A Self-Applicative Partial Evaluator and Its Use in Incremental Compilation. in [Fuch88], 1988

Full88 Fuller, D.A. and Abramsky, S., Mixed Computation of Prolog Programs. in [Fuch88], 1988

Futa71 Futamura, Y., Partial Evaluation of Computation Process — An approach to a Compiler-Compiler, in *Systems, Computers, Controls*, 2(5):45–50, 1971

Futa82 Futamura, Y., Partial Computation of Programs. in *RIMS Symposia on Software Science and Engineering*, Kyoto, 1982

Hara77 Haraldsson, A., A Program Manipulation System Based on Partial Evaluation, Linköping Studies in Science and Technology Dissertations No. 14, Linköping University, 1977

Hari84 Haridi, S. and Sahlin, D., Efficient implementation of unification of cyclic structures, in *Implementations of Prolog*, ed. J.A. Campbell, Ellis Horwood, 1984

Harm89 van Harmelen, F., The Limitations of Partial Evaluation, in *Logic-Based Knowledge Representation*, eds. Jackson, P., Reichgelt, H., and van Harmelen, F., MIT Press, 1989

Jone89 Jones, N.D., Sestoft, P. and Søndergaard, H., Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation, *Lisp and Symbolic Computation*, 2(1):9–50, 1989

Kahn82 Kahn, K.M., A Partial Evaluator of Lisp Programs Written in Prolog, First International Logic Programming Conference, Marseille, France, pp 19–25, 1982

Kahn84 Kahn, K.M. and Carlsson, M., The Compilation of Prolog Programs without the Use of a Prolog Compiler, in Proceedings of the international conference on fifth generation computer systems, Tokyo, ICOT, 1984

Klee52 Kleene, S.C., *Introduction to Metamathematics*, D. van Nostrand, Princeton, New Jersey, 1952

Knut77 Knuth, D.E., Morris, J.H., and Pratt, V.R., Fast Pattern Matching in Strings, *SIAM Journal on Computing Vol. 6*, No. 2, pp. 323–350, June 1977

Komo81 Komorowski, H.J., A specification of an abstract Prolog machine and its application to partial evaluation, PhD thesis, No. 69, Software Systems Research Center, Linköping University, 1981

Komo89 Komorowski, H.J., Synthesis of Programs in the Framework of Partial Deduction, Ser.A, No. 81, Departments of Computer Science & Mathematics, Åbo Akademi, Finland, 1989

Kond88 Kondoh, S. and Chikayama T., Macro Processing in Prolog, Technical Report TR-410, ICOT, Japan

Kowa74 Kowalski, R., Predicate Logic as Programming Language, Proc. Information Processing 1974, pp. 569–574, Elsevier North-Holland Inc., 1974

Kurs88 Kursawe, P., Pure Partial Evaluation and Instantiation, in [Bjør88] pp. 283–298, 1988

Lakh88 Lakhotia, A. and Sterling, L., Composing Logic Programs with Clausal Join, in [Fuch88], 1988

Lakh89a Lakhotia, A. and Sterling, L., Development of a Prolog Partial Evaluation System. Tech. Rept. Department of Computer Engineering and Science, Case Western Reserve University, 1989

Lakh89b Lakhotia, A. ProMiX: a Prolog Partial Evaluation System. Tech. Rept. CES-89-05, Department of Computer Engineering and Science, Case Western Reserve University, 1989

Lakh89c Lakhotia, A., A workbench for developing logic programs by stepwise enhancement, Technical Report TR-89-163a, Center for Automation and Intelligent Systems Research, Case Western Reserve University, 1989

Lakh90 Lakhotia, A. and Sterling, L., ProMiX: a Prolog Partial Evaluation System, in [Ster90], 1990

Lam89 Lam, J., Control Structures in Partial Evaluation of Pure Prolog, M.Sc. Thesis, Department of Computational Science, University of Saskatchewan, Canada, 1989

Lam90 Lam, J. and Kusalik, A., A Partial Evaluation of Partial Evaluators for Pure Prolog, Research Report 90-9, Department of Computational Science, University of Saskatchewan, Canada, 1990

Lass88 Lassez, J.-L. and Maher, M.J. and Marriot, K., Unification Revisited, in *Foundations of Deductive Databases and Logic Programming*, Ed. Minker, J., Morgan Kaufmann Publishers, Inc., 1988

LeSa88 Levi, G. and Sardu, G., Partial Evaluation of Metaprograms in a "Multiple Worlds" Logic Language, in [Fuch88], 1988

Lloy87 Lloyd, J.W. and Shepherdson, J.C., Partial Evaluation in Logic Programming, Tech. Rept. CS-87-09, Computer Science Department, University of Bristol, December 1987

Lomb64 Lombardi, L.A. and Raphael, B., LISP as the Language for an Incremental Computer. in *The Programming Language Lisp: Its Operation and Applications*, E.C. Berkeley and D.G. Bobrow, eds., pp. 204–219, MIT Press, Cambridge, Massachusetts, 1964

Mell86 Mellish, C.S., Abstract interpretation of Prolog programs, in the proceedings of the Third International Conference on Logic Programming, Springer-Verlag, 1986

Moge86 Mogensen, T., The application of partial evaluation to ray-tracing, Master's thesis, DIKU, University of Copenhagen, Denmark, 1986

Morr81 Morris, F.L., On List Structures and Their Use in the Programming of Unification, School of Computer and Information Science, Syracuse University, 1981

Naka87 Nakagawa, H., A Prolog Program Transformation System, in *Program Specification and Transformation*, ed. L.G.L.T. Meertens, North-Holland, 1987

NiMa90 Nilsson, U. and Małuszyński, J., *Logic, Programming and Prolog*, John Wiley & Sons, 1990

OKee85 O'Keefe, R.A., On the treatment of cuts in Prolog source-level tools, in *Symposium on Logic Programming*, IEEE, 1985

Owen89 Owen, S., Issues in the Partial Evaluation of Meta-Interpreters, in *Meta-Programming in Logic Programming*, The MIT Press, pp 319–339

PeSh87 Pereira, F.C.N. and Shieber, S.M., *Prolog and natural-language analysis*, Center for the Study of Language and Information, Leland Stanford Junior University, 1987

PeWa78 Pereira, F.C.N. and Warren, D.H.D., Definite Clause Grammars Compared with Augmented Transition Networks, Department of Artificial Intelligence Research Report No. 58, University of Edinburgh, October 1978

Plot70 Plotkin, G.D. A note on inductive generalisation, in *Machine Intelligence 5*, eds. Meltzer, B. and Michie, D., American Elsevier, New York 1970, pp. 153–163, 1970

Pres90 Prestwich, S., Partial Evaluation by Unfold/Fold with Generalisation, ECRC, München, November 1990

Reid89 Reid, Glenn, Adobe, personal communication, Nov. 1989

Reyn70 Reynolds, J.C., Transformational Systems and the Algebraic Structure of Atomic Formulas, in *Machine Intelligence 5*, eds. Meltzer, B. and Michie, D., pp. 135–152, American Elsevier, New York 1970

Robi65 Robinson, J.A., A machine-oriented logic based on the resolution principle, *Journal of the ACM* 12, pp. 23–41, 1965

Ross89 Ross, P., *Advanced Prolog, techniques and examples*, Addison-Wesley, 1989

Sahl90a Sahlin, D., Finding the Least Fixed Point Using Wait-Declarations in Prolog, in proceedings of PLILP'90, Linköping, Sweden, Lecture Notes in Computer Science 456, Springer-Verlag, pp. 151–158, 1990

Sahl90b Sahlin, D., The Mixtus Approach to Automatic Partial Evaluation of Full Prolog, in proceedings of the North American Conference on Logic Programming, MIT Press, 1990

Sahl91 Sahlin, D., Determinacy Analysis for Full Prolog, in proceedings of the ACM/IFIP Symposium on Partial Evaluation and Semantics Based Program Manipulation, ACM Press, 1991, forthcoming

Seki89 Seki, H., Unfold/Fold Transformation of Stratified Programs, Proceedings of the Sixth International Conference on Logic Programming, pp. 554–568, The MIT Press, 1989

Sest88 Sestoft, P. and Zamulin, A., Annotated Bibliography on Partial Evaluation and Mixed Computation, in [Fuch88] and [Bjør88], 1988

SmHi90 Smith, D. and Hickey, T., Partial Evaluation of a CLP Language, in proceedings of the North American Conference on Logic Programming, MIT Press, 1990

SmHi91 Smith, D.A. and Hickey, T., Partial Evaluation and Constraint Logic Programming, in proceedings of Symposium on Partial Evaluation and Semantics Based Program Manipulation (PEPM), 1991, forthcoming

Smit90a Smith, D.A. and Hickey, T., Partial Evaluation of CLP($\mathcal{FI}$), Technical Report CS-90-148, Computer Science Department, Brandeis University, March 1990

Smit90b Smith, D.A., Constraint Operations for CLP($\mathcal{FI}$), Department of Computer Science, Brandeis University, 1990

Ster86a Sterling, L. and Beer, R. D., Incremental Flavor-Mixing of Meta-Interpreters for Expert System Construction, Proceedings of the Third Logic Programming Symposium, Salt Lake City, Utah, 1986

Ster86b Sterling, L. and Shapiro, E., *The Art of Prolog*, MIT Press, 1986

Ster90 Sterling, L., *The Practice of Prolog*, MIT Press, 1990

Stro88 Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley Series in Computer Science, 1988

Take86 Takeuchi, A. and Furukawa, F., Partial Evaluation of Prolog Programs and its Application to Meta Programming, *Information Processing 86*, Elsevier Science Publishers, 1986

Tama84 Tamaki, H. and Sato, T., Unfold/fold Transformations of Logic Programs, Proceedings of the Second International Conference on Logic Programming, Uppsala, Sweden, pp. 127–138, 1984

Turc79 Turchin, V.F., A Supercompiler System Based on the Language Refal, *SIGPLAN Notices*, 14(2):46–54, February 1979

Venk84 Venken, R., A Prolog meta-interpreter for partial evaluation and its application to source-to-source transformation and query optimisation, Proceedings of the European Conference on Artificial Intelligence, Pisa 1984

Venk88 Venken, R., A Partial Evaluation System for Prolog: some Practical Considerations, in [Fuch88], 1988

Warr77 Warren, D.H.D., Implementing Prolog — Compiling Predicate Logic Programs, Research Reports No. 39–40, Department of Artificial Intelligence, University of Edinburgh, 1977

Warr83 Warren, D.H.D., An Abstract Prolog Instruction Set, Technical Note 309, SRI International, Menlo Park, CA, October 1983

Appendices

A. Non-cyclic and cyclic unification and anti-unification

The partial evaluator uses various forms of unification and anti-unification. In this section the code for doing anti-unification in Mixtus is shown. Cyclic structures are not yet fully supported by SICStus and Mixtus. A novel algorithm for anti-unification on cyclic terms will be shown in this section, which could be incorporated into a forthcoming version of Mixtus.

A.1 Unification

In order to ease the understanding of the code for anti-unification, some code for unification is shown below. No occur-check is made, and it is not capable of handling cyclic structures. Built-in unification is only used for binding variables.

```
unify(X,Y) :- (var(X); var(Y)), !, X=Y.
unify(X,Y) :-
          functor(X,FX,NX), functor(Y,FY,NY),
          FX=FY, NX=NY, !,
          unify_args(NX,X,Y).

unify_args(0,_,_).
unify_args(N,X,Y) :- N>0,
          arg(N,X,XN), arg(N,Y,YN),
          unify(XN,YN),
          N1 is N-1,
          unify_args(N1,X,Y).
```

Notice that no special cases are needed above for atoms or numbers; they are considered functors with arity 0 by `functor/3`.

A.2 Cyclic unification

The code below will be able to unify two cyclic structures, even if used on a Prolog system that does not support cyclic unification. The underlying Prolog system must however not perform an occur-check, as cyclic structures may be created. Also, there must be a built-in predicate able to compare the *addresses* of two terms or variables (the predicate `eq/2` below).

An extra argument containing a list of pairs of terms `u(`t_1`,`t_2`)` has been added, which keeps track of all terms while they are being unified. Whenever a pair of terms is encountered again, a

cyclic structure has been detected. Since creating a cyclic structure alone is no cause for failure, the unification simply succeeds for this pair.

```
c_unify(X,Y,_) :- (var(X); var(Y)), !, X=Y.
c_unify(X,Y,U) :-
          member(u(X2,Y2),U), eq(X,X2), eq(Y,Y2), !.
c_unify(X,Y,U) :-
          functor(X,FX,NX), functor(Y,FY,NY),
          FX=FY, NX=NY, !,
          c_unify_args(NX,X,Y,[u(X,Y)|U]).

c_unify_args(0,_,_,_).
c_unify_args(N,X,Y,U) :- N>0,
          arg(N,X,XN), arg(N,Y,YN),
          c_unify(XN,YN,U),
          N1 is N-1,
          c_unify_args(N1,X,Y,U).
```

The above code should be viewed as a minimalistic implementation of cyclic unification. The cyclic unification in SICStus Prolog is far more efficient, having a negligible overhead [Morr81].

A.3 Anti-unification

As already mentioned in the main text (see section 2.7), the anti-unification of two terms is the least general term that is more general than both terms. For instance, the anti-unification of `f(1,1,2)` and `f(2,2,2)` is the term `f(X,X,2)`.

From Plotkin [Plot70] we get the following algorithm (slightly modified) for anti-unification.

Input: two terms W_1 and W_2
Output: the anti-unification of W_1 and W_2
$V_1 := W_1$
$V_2 := W_2$
WHILE there are two terms t_1 and t_2 which have the same place in V_1 and V_2 respectively and do not have the same main functor
DO
    choose a variable z distinct from any in V_1 or V_2 and wherever t_1 and t_2 occur in the same place in V_1 and V_2, replace each by z.

V_1 and V_2 will now be identical and are the anti-unification of W_1 and W_2.

The corresponding Prolog-predicate has three arguments, two for the input terms and one for the resulting term.

```
anti_unify(X,Y,Z) :- a_unify(X,Y,[],_,Z).

a_unify(X,Y,S,S,X) :- X==Y, !.
a_unify(X,Y,S,S,Z) :-
          member(subst(X2,Y2,Z),S), X==X2, Y==Y2, !.
a_unify(X,Y,S1,S2,Z) :-
          functor(X,FX,NX), functor(Y,FY,NY),
          FX=FY, NX=NY, !,
          functor(Z,FX,NX),
          a_unify_args(NX,X,Y,S1,S2,Z).
a_unify(X,Y,S1,[subst(X,Y,Z)|S1],Z).

a_unify_args(0,_,_,S,S,_).
a_unify_args(N,X,Y,S1,S3,Z) :- N>0,
          arg(N,X,XN), arg(N,Y,YN),
          a_unify(XN,YN,S1,S2,ZN),
          arg(N,Z,ZN),
          N1 is N-1,
          a_unify_args(N1,X,Y,S2,S3,Z).
```

The main difference between Plotkin's algorithm and the Prolog code, is that the substitutions are not carried out at once in the Prolog code. Instead a substitution list is maintained, where each element is of the form `subst(`t_1`,`t_2`,`z`)`.

A.4 Cyclic anti-unification

The anti-unification algorithm can be extended to handle cyclic structures, much in the same way as cyclic structures can be handled in unification. For this purpose, one more argument is needed: the list of pairs which keeps track of all terms while they are being anti-unified. The elements in cyclic unification having the form `u(`t_1`,`t_2`)` are now instead `u(`t_1`,`t_2`,`z`)` as z is the anti-unification of t_1 and t_2.

```
anti_unify(X,Y,Z) :- ca_unify(X,Y,[],_,[],Z).

ca_unify(X,Y,S,S,_,X) :- c_equal(X,Y), !.
ca_unify(X,Y,S,S,_,Z) :-
         member(subst(X2,Y2,Z),S), c_equal(X,X2), c_equal(Y,Y2), !.
ca_unify(X,Y,S,S,U,Z) :-
         member(u(X2,Y2,Z),U), eq(X,X2), eq(Y,Y2), !.
ca_unify(X,Y,S1,S2,U,Z) :-
         functor(X,FX,NX), functor(Y,FY,NY),
         FX=FY, NX=NY, !,
         functor(Z,FX,NX),
         ca_unify_args(NX,X,Y,S1,S2,[u(X,Y,Z)|U],Z).
ca_unify(X,Y,S1,[subst(X,Y,Z)|S1],_,Z).

ca_unify_args(0,_,_,S,S,_,_).
ca_unify_args(N,X,Y,S1,S3,U,Z) :- N>0,
         arg(N,X,XN), arg(N,Y,YN),
         ca_unify(XN,YN,S1,S2,U,ZN),
         arg(N,Z,ZN),
         N1 is N-1,
         ca_unify_args(N1,X,Y,S2,S3,U,Z).
```

The predicate `c_equal/2` is used above to check two cyclic structures for equality. Ordinary cyclic unification (`c_unify/2`) can easily be changed into `c_equal/2` by replacing the first clause by

```
c_unify(X,Y) :- eq(X,Y).
```

Although the concepts of unification of cyclic structures and anti-unification are well-known, we are not aware of any previously published algorithm for combining them into cyclic anti-unification.

An example of cyclic anti-unification

For non-cyclic structures, the cyclic anti-unification algorithm will return the same results as the non-cyclic anti-unification algorithm. Let us therefore chose an example with cyclic structures.

If `A=[A1,A2|A]` and `B=[B1,B2,B2|B]` then the result of cyclic anti-unification of A and B is a new cyclic structure `C=[C1,C2,C3,C4,C5,C6|C]`. The result seems reasonable: the least general generalization of an infinite list with periodicity 2 and an infinite list with periodicity 3 is a list with periodicity 6, since the least common multiple of 2 and 3 is 6.

It is interesting to compare this with the result of unifying `A` and `B` above, which is a cyclic structure `D=[D1|D]`. Here the periodicity is 1, which is the greatest common denominator of 2 and 3.

B. member/2 and append/3

For reference, since these two common predicates are used throughout the text, they are defined here.

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

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

C. Utility predicates

Here is the source-code of some utility predicates used by the interpreters and partial evaluators in chapter 5.

clause_disj/3

```
clause_disj(G,Body) :-
        findall((G:-B), clause(G,B),Blist),
        list_to_disjunction(Blist,G,Body).

list_to_disjunction([],_G,fail) :- !.
list_to_disjunction([(H:-B)|Cls],G,Disj2) :-
         list_to_disjunction(Cls,G,Disj),
         cons_disj((G=H,B),Disj,Disj2).

cons_disj(false,A,A) :- !.
cons_disj(A,false,A) :- !.
cons_disj(A,B,(A;B)).
```

can_fold/3

```
can_fold(A,GS,Anew) :-
         member(gs(G,Gnew,true),GS),
         isinstance(A,G), !,
         copy_term(gs(G,Gnew),gs(A,Anew)).

isinstance(X,Y) :- verify((numbervars(X,1,_), X=Y)).

verify(X) :- \+ (\+ X).
```

The code above is used by partial evaluator 3. The code is adapted to partial evaluator 4 if the head of `can_fold/3` is changed to

```
can_fold(A,state(_C,GS,_Rin,_Rout),Anew) :-
```

to_disjunction/3

```
to_disjunction(Cls,Goalnew,Cls_or) :-
          Goalnew =.. [_|Args],
          to_disj(Cls,Args,Cls_or).

to_disj([],_Args,fail) :- !.
to_disj([(H :- B)|Cls],Args,B4) :-
          H =.. [_|Hargs],
          inst_u(Hargs,Args,Args,B,B2),
          to_disj(Cls,Args,B3),
          cons_disj(B2,B3,B4).

inst_u([],[],_,B,B).
inst_u([H|Hs],[A|As],Args,B,B2) :-
          (var(H), \+ varmember(H,Args) ->
                    A = H,
                    inst_u(Hs,As,Args,B,B2)
          ;         cons_conj((A=H),B,B3),
                    inst_u(Hs,As,Args,B3,B2)
          ).

varmember(X,[Y|_]) :- X==Y.
varmember(X,[_|L]) :- varmember(X,L).

cons_conj(true,A,A) :-!.
cons_conj(A,true,A) :-!.
cons_conj(A,B,(A,B)).
```

Index

Swedish Institute of Computer Science

SICS Dissertation Series

01. Bogumil Hausman, *Pruning and Speculative Work in OR-Parallel PROLOG*
02. Mats Carlsson, *Design and Implementation of an OR-Parallel Prolog Engine*
03. Nabiel A. Elshiewy, *Robust Coordinated Reactive Computing in SANDRA*
04. Dan Sahlin, *An Automatic Partial Evaluator for Full Prolog*

Printed in The Netherlands. Typeset in Times Roman, Avant Garde and Courier by the author, using a LaserWriter II and Microsoft Word 4.0 on a Macintosh Plus.

ISBN 91-7170-049-8
TRITA-TCS-9101
ISSN 0284-4397

ISRN SICS/D--91/04--SE
SICS Dissertation Series 04
ISSN 1101-1335