Pruning in logic programming

Lee Naish

(lee@cs.mu.OZ.AU, http://www.cs.mu.oz.au/~lee)

Technical Report 95/16

Department of Computer Science
University of Melbourne
Parkville, Victoria 3052
Australia

Abstract

The logic programming community has a love–hate relationship with operators for pruning the search space of logic programs such as cut, commit, once, conditionals and variations on these. Pruning operators typically are not declarative, result in incompleteness and/or unsoundness, decrease readability and flexibility of code and make program analysis and transformation more difficult. Despite this, nearly all non-trivial Prolog programs contain cuts, nearly all more recent logic programming languages have similar pruning operators and many languages insist on pruning operators in every clause. In practice, logic programming is less logical than functional programming.

Why it this so? Do we really need pruning operators? Can we have sufficiently powerful pruning operators which do not destroy the declarative semantics of programs? How are pruning operators related to logic, modes, functions and lazy evaluation? This paper attempts to answer some of these questions.
Keywords: cut, soft cut, commit, negation, modes, functions

1 Introduction

The key idea behind logic programming is that a program can be seen as a formula in some logic and computation can be seen as deduction [Kow74]. The answers computed are logical consequences of the program. Model theory allows us to reason about what is computed declaratively, without any consideration of how computations are performed. By choosing a suitable subset of some logic and a proof procedure, computations can be performed very efficiently. By knowing the proof procedure, programmers can reason about how computations are performed and implement specific algorithms as well as reason declaratively about their programs.

For pragmatic reasons, nearly all logic programming languages have additional facilities which do not fit well with this model. Typical languages allow side effects such as input, output and changing the state of the program, "meta level" primitives, interfaces to non-logical languages and primitives which allow the search space of the proof procedure to be "pruned". Non-declarative pruning is used in almost all substantial Prolog programs and is compulsary in several logic programming languages. Logic programming purists see benefits in either eliminating these facilities or bringing them within the declarative framework. The functional programming language community has been very successful in this respect and is an important source of ideas. Unfortunately, functional programming languages do not have anything closely related to the pruning constructs in logic programming languages. This paper attempts to give a deeper understanding of the role of pruning in logic programming and show how pruning can be done in a more declarative way.

The paper is organised as follows. We first discuss why pruning constructs have been introduced to logic programming languages and the main danger of doing so. Next we discuss how pruning is dependent on input/output modes and how this can influence the design of pruning constructs in various ways. We then discuss pruning operators in more detail, including the logic behind various ways in which they are used. Towards the end of the paper we suggest two new pruning constructs. Two important features of these constructs are the way in which mode information is incorporated and how they are given declarative semantics.

2 Why pruning?

The primary motivation of pruning is efficiency. The traditional way of describing what pruning operators do is in terms of *reducing the search space*. In the following program, the presence of the cut means that on backtracking it is not necessary to search for alternative solutions to `thiscase` or consider the second clause for `p`. This obviously saves execution time.

```
p(X, Y) :- thiscase(X), !, ... p(X1, Y1).
p(X, Y) :- thatcase(X), ...
```

The second way pruning can be used to improve efficiency is to *eliminate redundant tests*. If `thatcase` is the negation of `thiscase` then the call to `thatcase` can be removed since execution will never reach that point unless `thiscase` fails. Thus pruning can be used to implement a form of if-then-else or implicit negation.

There is a third, more subtle way pruning can improve efficiency. Consider the recursive call to `p`. If the language implementation is able to detect that the execution is determin-

istic and there is no further reference to `X`, the space used by `X` and the stack frame used by the original call to `p` can be reclaimed, by "garbage collection" and "last call optimisation", respectively. If it is not known that the execution is deterministic the space cannot be reclaimed since `X` and the stack frame for `p` may be needed on backtracking. Since determinism detection is undecidable, pruning operators generally allow more space to be reclaimed.

Of these three ways in which pruning can improve efficiency, the last one is the least known but the most important for large scale use of Prolog. Very often it makes the difference between a practical program which runs in $O(1)$ space and an impractical program which runs in $O(N)$ space. It is unfortunate that most books and manuals on logic programming languages do not give sufficient guidance to programmers for this very important use of pruning. Different implementations have different degrees of determinism detection and therefore require pruning operators in different places to detect determinism.

It is interesting to compare the situation in logic programming with that in functional programming. Functional programming languages no not have the built in nondeterminism that Prolog has. To evaluate a function call only the *first* matching equation need be considered. Multiple equations can be thought of as an if-then-else construct rather than a more general disjunction. This is one explanation of why functional programming languages do not need additional pruning operators for space efficiency.

However, *lazy* functional languages can have "space leak" problems which are very closely related to those in logic programming languages. Consider the following two programs which search a tree for "successful" leaves. The Prolog version returns single answers via backtracking and the functional version returns a list of answers.

```
search(Root, Root) :-
    successful(Root).
search(Root, Succ) :-
    child(Root, Child),
    search(Child, Succ).
```

```
>search root
>    = [root],    if successful root
>    = concat (map search (children root)), otherwise
```

If only one successful leaf is needed (perhaps only one exists) then the Prolog code typically wastes space. Choice points are left so that the search could be continued on backtracking and garbage collection and last call optimisation are disabled. A strict functional language would search the whole tree, which is also inefficient. A lazy functional language would compute in a very similar way to Prolog. Initially, only the first solution (the head of the list) is computed. Alternative solutions are searched for only if they are needed. As in Prolog, information must be stored so the search can be continued and this is the source of the space leak. It may be that space leak problems occur more rarely in functional programming languages due to the programming styles employed. However, the relationship between space leak problems in logic and functional languages is certainly worth investigating further.

There is a fourth way in which pruning can be see as contributing to the efficiency of logic programming. It is the implementation of negation as failure. Though negation as failure has been criticised on grounds such as incompleteness, it remains by far the most

efficient form of negation implemented. It is implemented in Prolog and similar languages by a combination of SLD resolution, which can be compiled very efficiently, and pruning, which adds very little overhead the the implementation of SLD resolution.

Negation as failure significantly adds to the expressive power of logic languages. Pruning is also used to implement general conditional constructs, for example, if-then-else. Such constructs are important for clearly expressing many algorithms. Pruning also increases expressive power through the use of "don't care nondeterminism", particularly in the committed choice logic languages. Thus, even if the memory management problems can be overcome sufficiently there are strong grounds for using pruning to improve the expressive power of logic languages.

3 The danger

A danger of adding pruning operators to a logic programming language is that the result is not a logic programming language. Most pruning operators are imperative rather than declarative in nature. Rather than forego the advantages of declarative programming we should attempt to give declarative semantics to programs with pruning operators. If we take the naive view of the semantics of a logic program containing pruning operators, simply ignoring the pruning, completeness is compromised. This can lead to unsoundness if negation as failure is used.

Some forms of completeness are already compromised in logic programming languages. The approach to soundness and completeness in logic programming has followed that in automated theorem proving. This seems appropriate since in logic programming systems computation can be view as construction of a proof. However, there are important differences between executing programs and attempting to prove theorems. It is acceptable for a theorem prover to compute indefinitely without any observable output. No-one is to blame; the theorem prover is just unable to prove the conjecture it is given. In a logic programming system such a situation is generally not acceptable. We would say that the program is in an infinite loop and the programmer is to blame.

Theoreticians are interested in *algorithms* which guarantee to find a proof if one exists. This is the basis of most completeness theorems. However, no implemented system can make such a guarantee. A system may terminate before finding a proof due to lack of memory, lack of patience of the operator, the hardware being struck by lightening or the heat death of the universe. In practice, the important thing is what can be concluded from the output of a system *if* the system terminates "normally". At least to a first approximation, the theorems that logic programmers are interested in are theorems concerning computations which terminate normally. This restriction also allows us to considerably strengthen the normal completeness theorems concerning SLD and SLDNF resolution.

Theorem 3.1 Let P and G be a normal program and goal, respectively, and M a model of $comp(P)$. If SLDNF resolution of $P \cup \{\leftarrow G\}$ terminates normally for all solutions, with computed answer substitutions $\{\theta_1, \theta_2, \ldots, \theta_N\}$ then these are the only instances of G which are true in M.

Proof This theorem is closely related to the soundness of SLDNF resolution (see [Llo87]), which is the basis of our proof. Suppose $N = 0$: $\leftarrow G$ finitely fails. We assume w.l.o.g. that there is a procedure g defined by the clause $g \leftarrow G$. SLDNF resolution of $P \cup \{\leftarrow \neg g\}$

would succeed so $\neg g$ must be true in all models (by the soundness of SLDNF resolution) and hence there is no model in which G is true.

Suppose $N > 0$. We assume w.l.o.g. that there is a procedure ga defined by the clause $ga \leftarrow G, \neg ans(\bar{X})$, where $\bar{X}$ are the variables in G and ans is a procedure which has computed answers $\{\theta_1, \theta_2, \ldots, \theta_N\}$ defined simply by N facts. SLDNF resolution finitely fails for $P \cup \{\leftarrow ga\}$ so, by the argument above, ga is false in all models. The equality theory in $comp(P)$ constrains the interpretation of ans so no other instance of G can be true in any model.

Consider the situation where a programmer writes the following program, with the interpretation that `p`, `q` and `r` are all true.

```
p :- not q.      q :- q.      r :- q, fail.
p :- q.                       r.
```

The intended interpretation is a model of the program (and its completion), which ensures *partial* correctness. However, to use any automated deduction system we must do more than write down a formula (or program) for which the intended interpretation is a model. No sound inference system could prove `q` since it is not a logical consequence of the program and although `p` is a logical consequence of the program, no proof can be constructed using SLDNF resolution. Similarly, although a proof for `r` is possible using SLDNF resolution, the normal Prolog search strategy will not find it. All these forms of completeness are avoided for computations which terminate. Since we are dealing with a programming language, it does not seem unreasonable to restrict attention to programs which are *totally* correct (that is, they terminate for the desired class of queries), rather than just partially correct.

We feel that the design of logic programming languages should aim to achieve the following:

1. Declarative semantics and soundness: the meaning of a program can be given in terms of a logical formula and computation can be seen as deduction;
2. Completeness for computations which terminate normally for all solutions;
3. Expressiveness;
4. Reasonable space efficiency; and
5. Reasonable time efficiency.

Pruning can help with the last three points but typically hinders the first two. Some pruning operators also make program transformation and meta interpretation more difficult due to their scope. Such problems are fundamentally syntactic and are best solved by initially transforming programs into a form which has more convenient pruning operators [O'K85].

4 Modes

Before considering specific pruning operators in more detail, it is worth noting the strong relationship between input/output modes and pruning. It it not possible to reasonably prune the search space without knowing certain mode information. This fact is manifest

in different ways with different constructs which use pruning. We briefly discuss conventional Prolog, IC-Prolog, MU-Prolog, NU-Prolog, Mercury, Gödel, Parlog, GHC, AKL and functional programming.

In conventional Prolog it is not generally possible to include pruning operators in a procedure without compromising correctness unless the mode of the procedure is known. Different modes require pruning to be done in different places for the set of computed answers to be consistent. Changing the literal order changes the modes which changes the set of computed answers. This is one way of explaining why Prolog with cut is "non-logical". Pruning in the wrong place results in missing answers (a desired answer is pruned from the search space) or wrong answers (due to missing answers inside negation or conditionals or code which is not "steadfast" [O'K90]).

Several variations on conventional Prolog have soundly implemented negation facilities with well defined declarative semantics, implemented using pruning. Examples include (the original) IC-Prolog [CM79], MU-Prolog [Nai85], NU-Prolog [TZ86] and Mercury [SHC95]. The original work on negation as failure [Cla78] noted that all variables in a negated goal must be input. An more obvious example of pruning in these languages is the conditional construct: `if C then A else B` (the syntax is slightly different in some of the languages).

All these languages restrict the modes of `C` to ensure the implementation soundly implements the declarative semantics. However, the languages differ in four ways: how the modes are specified; the strictness of the mode constraint; the method of checking the mode; and what occurs if the check fails. In IC-Prolog and MU-Prolog all variables in `C` must be input. NU-Prolog allows `C` to explicitly quantify variables. This changes the declarative semantics and relaxes the mode constraint on these variables. Implicit quantification for singleton variables is also supported. A singleton variable obviously cannot be input because there is no other occurrence of the variable which can be bound. Mercury also supports quantification which is implied by modes rather than explicit. Mercury uses mode declarations to give complete information about modes so more implicit quantification can be used.

In IC-Prolog, the inputs must not be further instantiated by a successful derivation of `C`. In the other languages the inputs must be sufficiently instantiated (ground in MU-Prolog and NU-Prolog) when `C` is called. IC-Prolog performs the mode check at runtime after `C` succeeds and a failed check results in a runtime error. MU-Prolog and NU-Prolog perform the mode check at runtime before `C` is called and a failed check results in the construct suspending. The suspended call may be resumed later when the inputs become further instantiated, or the computation may *flounder*. Mercury performs the mode check at compile time by analysing mode declarations and possibly reordering conjuncts in the program to achieve the desired mode. If this is not possible a compile time error results.

Gödel [HL94] has very similar negation facilities to NU-Prolog. It also has a pruning operator called commit. Commit does not cause unsoundness because it performs no pruning when called inside a negation. A form of completeness has also been proven: if a Gödel program without commits computes a certain answer then the program with commits added *could* compute the answer for *some* execution strategy [HLS90]. This result is useless in practice since the execution strategy which is used may not be the "right" one. The Gödel programmer is in a similar situation to the Prolog programmer, needing to know what the modes are in order to put commits in the right place. A miss-placed commit will not result in a wrong answer but it may result in a missing answer. This form of incompleteness can be used to implement versions of `var` and `nonvar` if certain assumptions about the computation rule and clause selection rule can be made.

Parlog programs have a commit operator in every clause [Gre87]. An important part of the language design is the "guard safety" requirement, which is essentially an input mode for the calls before a commit. Certain variables must not be further instantiated by the execution of these calls. If the mode declarations in a program do not ensure guard safety the result can be a compile time error or unpredictable behaviour at runtime. GHC has a similar commit operator to Parlog's, but guard safety (or "quietness") is ensured by the implementation [Ued86]. Calls which would violate quietness always suspend. An inconsistency in the (implicit) modes generally results in a *deadlock* at runtime. AKL [JH91] supports essentially the same commit operator as GHC and a variant of it has been shown to behave like a logical if-then-else construct [Fra91]. The mode constraint is checked *during* the execution of the "condition" (the guard) and if the constraint is not satisfied the computation suspends.

Functional programming languages naturally have modes defined — functions have inputs and an output. This is one reason why functional programming systems can simply choose the first applicable equation for evaluation (essentially pruning the rest). There have been many proposals for combining the functionality of logic and functional programming languages [Han94] and some of these proposals allow functions to be defined but used in more than one mode. This can be implemented by techniques such as narrowing. However, as soon as modes are relaxed the automatic pruning must also be abandoned.

5 Soft cut

The most ubiquitous pruning operator, cut in Prolog, is best understood as the combination of two operations:

- *once* prevents alternative solutions to calls before the cut being considered;
- *soft cut* prevents alternative clauses being considered (a single "choice point" is removed).

With well-defined modes, soft cut can be given declarative semantics in terms of negation [Nai89b]. This is illustrated by the code below. The first version of procedure `p` has a soft cut (`$`). If `I` is assumed to be an input argument the definition can be rewritten using negation and an existential quantifier (`some`) but no pruning, without affecting the set of computed answers.

```
   % soft cut ($) version              % equivalent logical version
p(I, ...) :- q(I, O), $, r(...).   p(I, ...) :- q(I, O), r(...).
p(I, ...) :-...                    p(I, ...) :- not (some [O] q(I, O)),...
```

The close relationship between soft cut and negation can also be seen in the implementation of the if-then-else construct of NU-Prolog. In NU-Prolog `(if some Vars C then A else B)` is declaratively equivalent to `(some Vars (C, A) ; not (some Vars C), B)`. It is implemented using pruning and if a variable in `Vars` appears in `A`, and `C` has multiple answers, soft cut *must* be used to avoid incompleteness [Nai86].

6 Once

Unfortunately, the logical reconstruction of `once` is much more difficult than for soft cut because there are many different uses of once. The different uses correspond to different

underlying logic and hence are best replaced by distinct constructs, each with unambiguous logic. We now discuss several such constructs.

For a call such as `p(I,_)`, where `I` is input and the output is never used, finding multiple solutions is pointless. They can only lead to the same answer being repeated at a higher level. In NU-Prolog this reasoning can be implemented using an existential quantifier: `some [O] p(I,O)`. The effect of the quantifier is that the scope of `O` is restricted, so it can't be used elsewhere, and if `I` is ground at the time of the call pruning is done when the call succeeds. Mercury implements this logic in a similar way, but the mode checking is done at compile time.

Sometimes it is known that a call has only one distinct solution but that solution may be returned more than once or at the time of finding the solution there may still be unexplored parts of the search tree (choice points have been left behind). For example, `slowsort(I,O)` sorts list of numbers by generating permutations and testing for sortedness. The same solution may be returned several times if the list has duplicates and choice points are nearly always left. Because of our knowledge of the `slowsort` relation is it reasonable to prune the search space when a solution is found if `I` is input. In the next section we propose a new primitive for this.

Sometimes `once` is used to find the *first* solution, rather than any solution. For example, to find the shortest solution to a searching problem a depth first iterative deepening algorithm may be used. The simplest way to code this in Prolog is to generate increasing depth bounds by backtracking and perform a depth bounded search for each one. As soon as a solution is found the search space is pruned. One difficulty with this style of programming is that the order of solutions depends on the order of clauses, the clause selection rule, the order of literals and the literal selection rule. The logic behind finding the first solution is thus much more complicated than the normal declarative semantics of a program. However, at least in some systems it is possible to precisely specify the logic. This has been done for the first solution operator of Trilogy [Vod88].

There are other uses of `once` which are normally thought of as cases of don't care nondeterminism. The programmer doesn't care which of several solutions are found, though the output is used. At the top level of a computation this causes no problem. For example, a compiler has a program as input and object code as output. There are many different possible ways of compiling a program and a compiler may potentially return several solutions but we are typically only interested in one. We are interested in soundness (the compiler output is correct) but not in completeness (all correct object programs in some class are computed).

Inside a larger computation don't care nondeterminism can cause incompleteness and with negation as failure it can result in unsoundness. If only one solution is required at some point, for example, the top level or where there are existentially quantified variables, then completeness is not required for that subcomputation. All that is required is that if a solution exists then at least one solution is found. This is referred to as *semicompleteness* in [Nai89a]. Later we discuss a primitive which allows this fact to be exploited to intruduce additional pruning in a safe way.

6.1 Only

To deal with the case where the programmer knows a call has only one solution, we propose the following quantifier-like construct:

`only Vars Goal`

Procedurally, this is equivalent to `once Goal` if the free variables in `Goal` are input. We leave open the possibility of implicit quantification and the details of mode checking but mention some of the many possibilities:

- the call suspends until all free variables are ground (like if-then-else in NU-Prolog);
- pruning is done only if all free variables are ground at the time of calling (like `some` in NU-Prolog);
- pruning is done only if no free variables are further instantiated by solving `Goal` (somewhat like IC-Prolog; this can be tested quite efficiently in a WAM based system by inspecting the trail[1]);
- the computation suspends rather than further instantiating any free variables (like commit in AKL);
- any of the above with some compile time analysis to reduce the number of runtime tests;
- compile time mode analysis and reordering is performed and an error results if any free variables could not be produced before the call (like if-then-else in Mercury);
- no mode checking or analysis is done (like conventional Prolog but with well-defined declarative semantics);
- no checking is done but debugging tools are provided to help locate any errors.

It would also be possible to use some form of declaration that a predicate is deterministic in a particular mode. This is supported in Mercury but only for determinism which can be verified by the compiler If `slowsort` was declared deterministic an error would result. Declarations which are not checked by the compiler [Smo84] could be given semantics in the way we suggest below for only.

In the body of a clause or goal, `only Vars Goal` can be viewed declaratively as being equivalent to `Goal`. However, this alone does not solve the problem of incompleteness — if `Goal` actually has multiple solutions then the pruned execution may be incomplete. We propose that the meaning of a program containing `only` is the program clauses (or the completion) plus some additional *assertions*. Each occurrence of `only` in the program results in an assertion that the corresponding goal is a partial function when used in the specified mode. For example, `only [O] slowsort(I,O)` results in the following assertion.

$$\forall x(slowsort(x, y_1) \wedge slowsort(x, y_2) \Rightarrow y_1 = y_2)$$

The intended interpretation should be a model of the assertions as well as the rest of the program. With this constraint the pruning cannot introduce incompleteness. The meaning of a program can still be seen as a logical formula and the expected soundness and completeness results hold. The assertions can be seen as additional logic which the execution mechanism can use to improve the efficiency of the theorem proving process. Consider the following example:

[1] An observation probably first made by Mats Carlsson.

```
p :- not q.     q :- only [X] r(X), X=b.     r(a).
                                             r(X) :- r(X).
```

The call to `r(X)` could return the answer `X=a`. The alternatives would be pruned, resulting in `q` failing and `p` succeeding. This result is unsound using the normal meaning of the program but it is a logical consequence of the completion of the program *plus* the assertion $r(x_1) \wedge r(x_2) \Rightarrow x_1 = x_2$.

Proposition 6.1 If M is a model of $comp(P)$ and the assertion $\forall I(p(I, O_1) \wedge p(I, O_2) \Rightarrow O_1 = O_2)$ and $p(I, O)$ has a computed answer substitution θ which does not further instantiate I, then θ subsumes all computed answers.
Proof Let $p(t_1, t_2)$ be a ground instance of a computed answer. There exists γ s.t. $p(I, O)\theta\gamma = p(t_1, t_3)$, since θ does not further instantiate I. By the assertion, $t_2 = t_3$ is true in M. Due to the equality theory in $comp(P)$, t_2 must be syntactically identical to t_3. Thus any ground instance of a computed answer is an instance of $p(I, O)\theta$ so θ subsumes all computed answers.

Definition *SLDNFP resolution* of a normal program and goal which may contain "only" is the same as SLDNF resolution ignoring occurrences of *only* $Vars$, except for the following. If there is an SLD derivation $\leftarrow G_0, \leftarrow G_1, \ldots, \leftarrow G_i, \ldots, \leftarrow G_j, \ldots$ where G_i is of the form $P, only\ Vars\ Q, R$ and G_j is of the form $(P, R)\theta$ and θ does not further instantiate the free variables in Q then the clause and atom selections in the goals $G_i \ldots G_{j-1}$ (the computation which solved Q) can be fixed rather than nondeterministic.

Theorem 6.1 If P is a normal program and G a normal goal, both of which may contain "only", and M is a model of $comp(P)$ and the assertions corresponding to the occurrences of only, then SLDNFP resolution is sound with respect to M and is complete with respect to M for terminating all solutions computations.
Proof Follows from the previous proposition and theorem and the soundness of SLDNF resolution.

A program with an incorrect assertion (the assertion is not true in the intended interpretation) should be treated in a very similar way to a program with an incorrect clause. Both can result in missing or wrong answers. Declarative debugging techniques `Sha83` can be extended to diagnose such programs. As well as returning such errors as incorrect clause instances and uncovered atoms, an *incorrect assertion* may be returned. Diagnosis of potentially missing answers in `only Vars Goal` can be done as follows. Call `Goal` and if it fails diagnose missing answers in `Goal` using existing methods. Otherwise, if the solution found for `Goal` is incorrect, diagnose the wrong answer using existing methods. Otherwise, use an oracle to check if there are any other valid instances of `Goal` (an "incompleteness" question [Nai92]). If there are other valid instances the assertion is wrong; if not the goal executes correctly.

6.2 Exists

If any one solution operator is used in a program then efficiency can often be increased if the pruning is "pushed" inside the computation which is being pruned. Rather than completely computing a solution then pruning the search space, some pruning can be done before the computation succeeds. We will illustrate this with the following program scheme containing a generic one solution construct.

```
P :- once Q.
Q :- (R ; S), T.
T :- U, V.
R :- W.
...
```

Rather than waiting until `Q` succeeds, the pruning can be done inside `Q`. The body of the clause for `Q` can be replaced by `once ((R ; S), T)`. Then, since the call to `T` is the last conjunct inside a `once` construct, multiple solutions to `T` would never be used so it can be replaced by `once T`. Since `T` is only called in context where one solution is required, the pruning can be pushed into the definition of `T` and similar steps can be repeated. This leads to the following code:

```
P :- once Q.
    % only one solution needed
Q :- once ((R ; S), once T).
    % only one solution needed
T :- once (U, once V).
R :- W.
...
```

Further pushing of pruning can be done with more information about the program. Suppose we know that for all solutions to `R`, there is also a solution to `T`. The body of `Q` can be transformed and another `once` construct added around the call to `R` then pushed inside the definition of `R`:

```
P :- once Q.
    % only one solution needed;
    % any solution to R can be
    % extended to a solution to T
Q :- once (once R, once T ; S, once T).
    % only one solution needed
T :- once (U, once V).
    % only one solution needed
R :- once W.
...
```

Some occurrences of `once` in the program are redundant but even if they are eliminated the program is much more obscure than the original. The procedures have been specialised so they can only be used in the context where a single solution is required, making code reuse more difficult. However, the fact that pruning has been pushed inside `R` may halve the space usage of the program. Pushing pruning into recursive procedures may reduce the space complexity from $O(N)$ to $O(1)$.

To design pruning operators which correspond to pushing pruning in the way we have illustrated, we need to understand the logic behind the process [Nai89b]. The key factors are maintaining what calls are in a single solution context, the computation rule, additional information relating solutions to different calls (for example, `R` and `T`) and modes. The computation rule is determined by the system and we will initially restrict attention to left to right computation rules as this greatly simplifies pushing pruning. Similarly, maintaining

the single solution context can be done very easily by the system. However, pushing pruning only results in significant benefits when pruning can be pushed to before the rightmost call in a conjunct. This can only be done safely with additional information.

The pruning primitive we suggest provides exactly this information and the implementation handles the relatively simple aspects of pushing pruning. The end result is programs which are declarative, are less cluttered with pruning operators, can be used in more than one mode and which prune the search space even sooner than would be done in Prolog. The basic syntax is a refinement of the construct proposed in [Nai89b]. It is intentionally similar to the conditional construct in Prolog (`->`) and logical implication (`=>` in NU-Prolog). For the program scheme above we would write the following:

```
P :- once Q.
    % any solution to R can be
    % extended to a solution to T
Q :- (R => T ; S, T).
T :- U, V.
R :- W.
...
```

This basic syntax, and the statement that any solution to `R` can be extended to a solution to `T`, relies on mode information. To make the mode information explicit we propose a quantifier-like construct (other ways of specifying the modes would also be possible). If each predicate has two arguments, the first of which is input, the definition of `Q` would be written as follows.

```
    % any solution to r can be
    % extended to a solution to t
q(I,O) :- (r(I,L) => exists [O] t(L,O) ; s(I,L), t(L,O)).
```

The declarative semantics of this code is the clause

```
q(I,O) :- (r(I,L), t(L,O) ; s(I,L), t(L,O)).
```

plus the assertion

$$\forall x \forall y (r(x,y) \Rightarrow \exists z \; t(y,z))$$

The procedural semantics of the code is that if `Q` is called in a single solution context with the appropriate mode then `R` is also called in a single solution context and additional pruning will be done. The mode constraint is that `O` must be output in the call to `R`, that is it must be a free variable at the time of the call. Note that for the other pruning operators discussed the mode requirement was that certain variables be input. Runtime testing of output modes is generally much cheaper than runtime testing of input modes, which makes this pruning operator especially attractive for implementations which do not do extensive compile time mode analysis.

Conceptually, the system has a "pruning context" for each call at runtime. This can be thought of as the choice point to cut back to when it is known that the call will succeed, or a special *null* value if no pruning should be done. The pruning context of a call is inherited by the body of the matching clause. The following rules specify how the pruning context P of a formula is inherited to sub-formulas. For a formula $A; B$, both A and B have context P.

For a formula A, B, by default A has context *null* and B has context P. If it is known that B must succeed then both A and B have context P. For a formula A=->B, both A and B have context P, assuming the mode constraint is satisfied. For a formula *if* C *then* A *else* B using soft cut, the pruning is inherited in the same way as $C, A; B$. For a formula *once* A (or a similar one solution operator), the context of A is P, if $P \neq null$, otherwise it is a new context which would prune the execution back to the start of the execution of A.

In the program above, the `once` construct introduces a new pruning context. This context is passed to `Q`, `T`, `R`, `V` and `W` and the *null* context is passed to `S` and `U`. As soon as it is know that `W` will succeed the disjunction in `Q` (and any other choice points which have been introduced since `Q` was called) can be pruned. For example, if `W` was defined using the clause `W :- X => Y`, as soon as it is known that `X` will succeed all the choice points can be pruned. Using cut, the disjunction would only be pruned when `R` succeeded. In the worst case this may double the space usage because garbage collection of the variable `I` would not be possible. If `Y` contained a recursive call to `Q` the space saving using exists instead of cut could be arbitrarily large. Examples like this do occur in practice [Nai89b].

6.3 Other pruning operators

So far we have not discussed committed choice languages in detail. Many uses of commit in these languages are either redundant or can be seen as an implementation of if-then-else [Nai88]. However, the cases of "don't care" nondeterminism such as merge cannot be explained by this logic. Many committed choice programs containing this style of nondeterminism can be explained logically in terms of the exists construct. We can start with a logic program which has no pruning and may have lots of unwanted nondeterminism. At the top level a one solution operator is used. The program is written in such a way that the pruning at the top level can be pushed into every nondeterministic procedure in the whole program. Consider the following simple example, where `prod1` and `prod2` produce two lists deterministically, `merge` merges the two lists in any order (it would return many solutions on backtracking) and `combine` combines the list elements to produce the final output.

```
p(O) :- prod1(Ps), prod2(Qs), merge(Ps, Qs, Rs), combine(Rs, O).
```

If `combine` always succeed the last conjunction operator can be replaced by `=> exists [O]` and the pruning context of `p` will be inherited by `merge` as well as `combine`. Furthermore, we know each recursive call to `merge` will succeed, which is vital for semicompleteness [Nai89b]. The definition of `merge` can therefore be as follows:

```
merge([], Bs, Bs).
merge(As, [], As).
merge(A.As, Bs, A.Cs) :- true => exists [Cs] merge(As, Bs, Cs).
merge(As, B.Bs, B.Cs) :- true => exists [Cs] merge(As, Bs, Cs).
```

The pruning context of `p` is passed to each call to `merge` and the calls to `true`. If a one solution operator is used at the top level then pruning can be done as soon as any clause of `merge` is selected, eliminating the (don't know) nondeterminism completely. The clause selection rule used for `merge` determines which interleaving of `Ps` and `Qs` is passed to `combine`. In committed choice languages the computation rule is also made more flexible. A more flexible computation rule can lead to non-termination, but there are two important pitfalls most committed choice programs avoid: cyclic modes and "speculative" bindings

[Nai93]. The exists construct identifies where bindings would be speculative and hence it may play an important role in computation rules as well as pruning.

We do not claim that the selection of pruning constructs we have discussed covers all possible reasonable uses of pruning. It will require experimentation with these primitives to determine if they are sufficiently expressive. If they are not then it may be possible to refine these constructs or develop new logical pruning primitives to cover the additional cases. There are two possible refinements which we briefly mention now.

The first refinement concerns exists. In the version of exists we have described only local information can be used to determine a solution exists. In practice there may be more global information available, such as that described by type declarations. Type information is vital for the semantics of logic programs and should not be ignored when giving semantics to pruning operators. As an example, the use of exists in `merge` is actually incorrect if we ignore types, since if `merge` is called with non-lists a solution may not exist. A refined version of exists could consider a restricted set of calls to `merge`, such as those where there arguments are lists.

The second refinement concerns only. It is sometimes the case that multiple answers to a procedure are considered equivalent even though they are not syntactically identical. For example, a given multiset could be represented by many different permutations of the same list. Pruning may be used so only one permutation is returned. The logical basis of this is a version of only which uses a different equality theory. Using a non-standard equality theory has many ramifications for language design. Here we simply note its connection with this form of pruning.

7 Conclusion

Pruning is important for efficiency of logic programs. It is particularly important for space efficiency although this is often overlooked. An implementation of a logic programming system must have some way of detecting determinism in order to safely reclaim memory to avoid space leaks. There are interesting parallels between this and space leaks in lazy functional languages. Pruning is also important for increasing expressibility of logic programming languages by implementing negation as failure and related constructs.

The main danger of pruning constructs is they can destroy the declarative reading of programs. However, it is possible to design declarative pruning constructs. There are two key elements to the design of such constructs. The first is that different kinds of pruning should be done by different constructs: general purpose pruning operators such as cut have an ambiguous declarative reading at best. The second is the role of modes: all pruning requires knowledge of modes. Different modes typically correspond to different quantifiers in the corresponding logic.

To develop declarative semantics for more complex uses of pruning it is helpful to separate the meaning of a construct into two components. The first is the meaning within a goal. The second is an additional assertion. Treating the assertions as part of the meaning in a similar way to clauses means we can have very efficient proof procedures (using pruning) without losing soundness. When negation as failure is used, soundness is closely related to completeness for finite computations. In a programming language the programmer is responsible for termination and by restricting attention to finite computations we get a stronger and more practical completeness theorem. The programmer need only consider truth in the intended interpretation rather than all models.

There is significant flexibility in the way mode information is specified and checked, and what occurs if mode constraints are not satisfied. Pruning constructs can emphasise the logic by such means as explicit quantifiers, leaving the mode information implied or the mode information can be more explicit and the underlying logic implied, or there can be some combination of the two. The strictness of mode checking can be varied. Inputs can be constrained to be ground or just not further instantiated in all derivations or all successful derivations or some successful derivation. Mode checking can be done at compile time, at runtime before, during or after some computation, or some combination of both or not at all. If a mode constraint is not satisfied the result can be compile time or runtime call reordering or error messages, floundering or deadlock, no pruning, wrong or missing answers or unpredictable behaviour.

All language design requires compromise and the pruning constructs of a logic programming language are closely related to other features. However, there seems to be sufficient flexibility in implementation (particularly the relationship with modes) and the way semantics are specified (for example, using assertions) that practical logic programming languages which use only declarative pruning constructs can be designed and implemented.

References

[Cla78] Keith L. Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and data bases*, pages 293–322. Plenum Press, 1978.

[CM79] K.L. Clark and F.G. McCabe. The control facilities of IC-prolog. In Donald Michie, editor, *Expert systems in the microelectronic age*, pages 122–149. Edinburgh University Press, 1979.

[Fra91] Torkel Franzén. Logical aspects of the Andorra Kernel Language. Research report R91:12, Swedish Institute of Computer Science, Kista, Sweden, October 1991.

[Gre87] Steve Gregory. *Design, application and implementation of a parallel logic programming language.* Addison-Wesley Publishers, Reading, Massachusetts, 1987.

[Han94] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[HL94] P.M. Hill and J.W. Lloyd. *The Gödel programming language.* MIT Press, Cambridge, Massachusetts, 1994.

[HLS90] P.M. Hill, J.W. Lloyd, and J.C. Sheperdson. Properties of a pruning operator. *Journal of logic and computation*, 1(1):99–143, 1990.

[JH91] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra Kernel Language. In *Proceedings of the 1991 International Symposium on Logic Programming*, pages 167–183, San Diego, June 1991.

[Kow74] R. Kowalski. Predicate logic as programming language. In Jack L. Rosenfeld, editor, *Proceedings of the Sixth IFIP Congress (Information Processing 74)*, pages 569–574, Stockholm, Sweden, August 1974.

[Llo87] John W. Lloyd. *Foundations of logic programming (second, extended edition).* Springer series in symbolic computation. Springer-Verlag, New York, 1987.

[Nai85] Lee Naish. The MU-Prolog 3.2 reference manual. Technical Report 85/11, Department of Computer Science, University of Melbourne, Melbourne, Australia, October 1985.

[Nai86] Lee Naish. Negation and quantifiers in NU-Prolog. *Proceedings of the Third International Conference on Logic Programming*, pages 624–634, July 1986.

[Nai88] Lee Naish. Parallelizing NU-Prolog. *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, pages 1546–1564, August 1988.

[Nai89a] Lee Naish. Proving properties of committed choice logic programs. *Journal of Logic Programming*, 7(1):63–84, July 1989.

[Nai89b] Lee Naish. Pruning the search space of Prolog: cut and other primitives. *Proceedings of Australian Joint Artificial Intelligence Conference*, pages 238–253, November 1989.

[Nai92] Lee Naish. Declarative diagnosis of missing answers. *New Generation Computing*, 10(3):255–285, 1992.

[Nai93] Lee Naish. Coroutining and the construction of terminating logic programs. *Australian Computer Science Communications*, 15(1):181–190, 1993.

[O'K85] Richard O'Keefe. On the treatment of cuts in prolog source level tools. In *Proceedings of the Second IEEE Symposium on Logic Programming*, pages 68–72, Boston, Massachusetts, July 1985.

[O'K90] Richard A. O'Keefe. *The Craft of Prolog.* Logic Programming. MIT Press, Cambridge, Massachusetts, 1990.

[SHC95] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. Technical Report 95/14, Department of Computer Science, University of Melbourne, Melbourne, Australia, March 1995.

[Smo84] Gert Smolka. Making control and data flow in logic programs explicit. In *Conference Record of the ACM Symposium on LISP and Functional Programming*, pages 311–322, Austin, Texas, July 1984.

[TZ86] NU-Prolog reference manual, version 1.0. Technical Report 86/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1986.

[Ued86] Kazunori Ueda. *Guarded Horn clauses.* D.Eng. thesis, University of Tokyo, Tokyo, Japan, March 1986.

[Vod88] Paul J. Voda. The logical reconstruction of cuts as one solution operators. In John W. Lloyd, editor, *Proceedings of the Workshop on Meta-programming in Logic Programming*, pages 379–384, Bristol, England, June 1988.