A Partial Evaluation of Partial Evaluators for Pure Prolog

John K. K. Lam
Anthony J. Kusalik

TR 90-9

Department of Computational Science
University of Saskatchewan
Saskatoon, Saskatchewan
S7N 0W0 Canada

November 1990

A Partial Evaluation of Partial Evaluators for Pure Prolog

John K. K. Lam
Anthony J. Kusalik

TR 90-9

Department of Computational Science
University of Saskatchewan
Saskatoon, Saskatchewan S7N 0W0 Canada

Abstract

Although use of partial evaluation is prevalent in logic programming research, few implementation principles have been developed. Partial evaluators are often cited in the literature with little explanation of their type, quality, or general applicativeness. Further, there is no taxonomy by which partial evaluation systems can be characterized and compared.

This paper describes progress towards solving this problem. Five partial evaluation systems for (pure) Prolog are studied. Their general control structures are analyzed, with concentration on their solutions to the issues of when or what to unfold, when to terminate unfolding, and how to pass multiple solutions between subgoals. Seven sample programs are tested using the five partial evaluators. Qualitative and quantitative measures of the residuals are taken. The final results are a comparison of the partial evaluators and their performance, and an initial taxonomy for partial evaluation systems for (pure) Prolog.

November 1990

1. Introduction

Partial evaluation is a specialization technique for transforming a computer program into an equivalent, specialized form based on partial, known input. The resultant program, the **residual**, when given the remaining input, behaves identically to the original program when the latter is given the same, combined inputs. Although partial evaluation has been applied to a wide variety of programming languages and purposes, the technology is not yet mature. Partial evaluators are becoming more commonplace, but the concepts underlying them are not fully explored.

Implementing a general purpose, automatic, and thorough partial evaluation system is not an easy task. Each different type of language presents its own set of problems. These problems are often made worse by a lack of semantic foundation in the target language. Logic-programming languages — Prolog is one example — have, in general, a better defined semantics. Yet, partial evaluation in logic-programming languages is still a difficult task.

One of the problems with the state of partial evaluation research is that little effort has been put into summarizing the current technology. Researchers are struggling with new partial evaluation techniques and algorithms. Meanwhile, there is no classification scheme for the partial evaluators mentioned in the literature. Much of the time they are described in a sketchy manner with few algorithmic or implementational details. Little, if any, comparison is made with others. The published papers tend to de-emphasize the inherent design and implementation problems. Emphasis is typically on the results of the partial evaluation process rather than on how it is achieved. Such papers alone have a misleading effect of over-simplifying the difficulties of partial evaluation. Two collections of papers [Fuch88, Bj¿+88] from the first conference on partial evaluation solved some of these shortcomings, but there is still an absence of a good taxonomy.

Not infrequently, papers appear stating that for a particular research project, a partial evaluator was utilized. Sometimes the partial evaluator was implemented as part of the research work. Unfortunately, all too often it is not stated how the partial evaluator was implemented (or the algorithm is described only superficially), what types of programs the system can work on, how applicable the system is to other types of programs, to what degree the partial evaluator is automatic (what degree of human interaction is necessary), or how this partial evaluator compares (both in terms of its workings and in the quality of its residuals) with others. Many questions, such as how much better is a particular special-purpose partial evaluator for a specific problem than a general-purpose one? , are rarely addressed.

Partial evaluation has been gaining increasing interest in the logic-programming community [SeSo88,Bj¿+88]. However, here again few comparisons of partial evaluators appear. When mentioned in logic-programming literature, there is too often little description of design, implementation techniques, control strategies, or algorithms used.

This work is a step towards the solution of these problems and deficiencies in the context of logic-programming languages. The partial evaluation algorithms used in several published papers or available systems are identified, studied, and compared. Emphasis is on their control strategies. Also, a partial taxonomy for categorizing partial evaluators is devised. To make the task more tractable, input is restricted to pure Prolog; that is, partial evaluation of pure-Prolog programs only is considered. The results provide useful information not only for future implementers and researchers, but also to users and those encountering partial evaluation in the literature.

The paper is organized as follows. Basic concepts of partial evaluation in Prolog are discussed in Section 2. It describes the problems which must be addressed in designing and implementing partial evaluation systems. A detailed description of the research experiment can be found in Section 3, and the results with analysis constitute Section 4. Section 5 generalizes results, draws conclusions, and mentions future work. The contributions of the work are reviewed in Section 6. The test programs used for the experiment are listed in the Appendix.

It is assumed that the reader has some proficiency with Prolog and familiarity with the concepts of partial evaluation. A detailed, theoretical foundation of partial evaluation in logic programming is given by Lloyd and Shepherdson [LlSh87].

2. Partial Evaluation in (Pure) Prolog

Partial evaluation is a specialization process. A residual program is a form of the original program specialized for the particular input known beforehand. Multiple residual programs may exist for the same original program.

Given a program ***P*** and a goal ***G***, the aim of the specialization is to derive a more efficient (according to some criteria) program ***P«*** which gives exactly the same answers for ***G*** as does ***P***. In addition, ***P«*** should finitely fail for every goal for which ***P*** finitely fails. ***P«*** may fail to compute answers for goals which are not instances of ***G*** but for which ***P*** gives an answer.

2.1. Restriction to Pure Prolog

Informally, **pure Prolog** is Prolog without any meta-logical or extra-logical predicates, cut, or negation. The partial evaluation systems considered in this paper need only deal with pure-Prolog programs. This is because many of the impure constructs of the full language do not have a declarative semantics, and because the transformations used in partial evaluation do not, in general, maintain equivalence for side-effecting predicates. Further, the operational semantics of some impure constructs (i.e. built-in predicates) vary among Prolog systems.

2.2. The Basic Process

The basic operation of a partial evaluator is to reformulate, prior to runtime, parts of a program for which it has sufficient input data to do so. This process can be broken down into three suboperations: i) constant and data structure propagation, ii) controlled (subgoal) **unfolding**[1] , and iii) elimination of superfluous information. Unfolding [TaSa84] involves recursively replacing a goal with the body of a clause whose head is unifiable with the goal. It is very similar to the resolution process of logic-programming languages. Each unfolding step is monitored and controlled closely by the partial evaluator's control regime. Partial evaluators employing different control strategies can produce different residuals from the same inputs.

A partial evaluation system needs two inputs: a program and a portion of the (runtime) input to that program. For partial evaluators of Prolog, the latter is a query. Some authors (e.g. Fujita [FuFu88]) consider ground unit clauses to be partial, known input to the program as well.

The query given as the partially-known input actually represents a pattern of queries which may (later) be given to the residual. Binding of arguments in the query restricts the pattern. Variables make it more general. The amount of specialization which can be performed is often dictated by the degree of instantiation. For queries with little instantiation, little or no reformulation may be possible. When the arguments are completely instantiated (to ground terms), a great deal of specialization may be possible and the residual may consist of only unit clauses. Between these extremes, when arguments are partially-instantiated, a moderate amount of specialization can typically be performed. This situation is also where differences in partial evaluators usually are most apparent.

---

[1] Some authors, in fact, define partial evaluation as program specialization based on unfolding [GaBr89].

The control strategy of a partial evaluator decides whether a particular goal is to be unfolded. Some systems might unfold a particular goal, while others would not. Consequently, different control strategies can lead to differing residual programs. These may later be found to vary in quality. Also, some control strategies may work well in certain situations, and not in others. Unfortunately, determinations of quality and applicability are difficult to make a priori.

2.3. Effects of Partial Evaluation

Partial evaluation is an optimization process in that it eliminates overheads brought about by the generality of a program. Usually partial evaluation is employed to improve the runtime efficiency of a program. This is possible because reduction steps which would be performed at runtime are performed instead at partial evaluation time. Fewer reductions at runtime translate into more efficient execution. The degree of improvement is related to the proportion of reductions which are performed at partial evaluation time. The improvement in runtime efficiency can be accounted for in a number of ways, some of which are related: fewer resolution steps, fewer unifications, greater determinacy, and a smaller search space. In some cases a deterministic residual program may be generated from a nondeterministic original.

Partial evaluation has other effects. Elimination of always-failing clauses and non-productive solution alternatives leads to a reduction in search-space size. This reduces (program) memory consumption. The eliminations may also yield a smaller program, which requires less file storage resources.

The size of a residual program is often different from that of the original. In some cases, it is less because the number of unwanted clauses is reduced (as mentioned above). In other cases, the size may be dramatically increased because of expansion of solution alternatives. How the program size is affected is dependent on the original program, the query, and the partial evaluator used. Some generalizations in this respect can be drawn from this work and are presented in Section 4.

Unless a trivial (i.e. uninstantiated) query is used as the partial-evaluation time input, a residual program is not as general as the corresponding original; for some runtime queries it may not behave as the original. Thus, while a residual may be more efficient, there may be otherwise legitimate cases where it cannot be called upon. A user should consider carefully a residual program's generality, as well as speed, before using it.

2.4. Problems with Current Applications of Partial Evaluation in Prolog

Given the potential benefits of partial evaluation, it is often used in logic programming research. A growing number of successful partial evaluators exist. However, evidence suggests that the basic designs and techniques for partial evaluation are not yet fully understood. Production of optimal code is rarely assured. A partial evaluator might run for a long time but produce a residual program which is only a minor improvement over the original. In fact, it is rare to have any indication of how good the residual program is.

Papers regularly appear where it is stated that a partial evaluation was implemented and/or utilized. Unfortunately, many aspects regarding the partial evaluator — characteristics necessary to fully understand its essence or impact — are never addressed. Is the system naive, requiring considerable human interaction? Does it work well only for a certain class of programs? Often it is not stated how much user interaction is necessary, or of what type. There is usually no indication of the optimality of the generated residual, or for what factors it is optimal. Little may be said of the class of programs to which the partial evaluator is applicable. Many evaluators seem quite specific for the context they are described in. Implementation issues are seldom discussed, or only mentioned superficially. There is often little discussion of what kind of loop-detection mechanism is used, or the way it couples multiple solutions, or whether it eliminates bound arguments, always-succeeding goals, trivial clauses, etc. All of these factors directly affect the resulting partial evaluator, the residual programs it produces, and its overall utility.

Another question not usually addressed in the literature is identification of common problems which all partial evaluators must confront. Most published partial evaluation algorithms concentrate on one particular aspect (the principle being expounded), leaving others virtually unmentioned.

Interest is often placed on the results of partial evaluation, rather than how they are achieved. Furthermore, there is no taxonomy or classification scheme which readers or users can employ to rank or compare the partial evaluators.

This research is motivated by these deficiencies. Its goal is to help rectify them.

2.5. Partial Evaluation Strategies in Prolog

The control strategy for partial evaluation of Prolog programs can be decomposed into answering two sets of questions. The first set are hard questions, relating to the basic suboperations of the partial evaluation process. These questions are:

- When or what to unfold?
- When to terminate (unfolding)?
- How to couple solutions between goals?

The second set of questions are easier:

- Are completely-instantiated arguments (of a predicate) eliminated?
- Are always-failing clauses eliminated?
- Are always-succeeding goals eliminated?
- Are trivial clauses[2] eliminated?
- Are duplicate clauses eliminated?

These latter questions are considered easier and less discriminating as they can be given simple yes or no answers. This is not the case with the first questions, where the answer must be more descriptive. The eliminations can be considered optimizations.

The last optimization must be used judiciously because the removal of duplicate clauses may result in a change of semantics. The removal of clauses will mean that certain solutions, though duplications, may not be generated.

The first set of questions are the most important in characterizing a partial evaluator. They are each considered in turn. Some known strategies for solving them are outlined.

2.5.1. When to Unfold

The decision of when to unfold a goal (or what goal to unfold) is central to the partial evaluation process. The decision depends on the basis of specialization (i.e. the optimization desired), and whether or not unfolding the particular goal will contribute in the effect. The decision to unfold is usually based upon a general policy. There may be exceptions however, and these are often dealt with by the related question addressed in the next section, When to stop unfolding? .

The decision whether to unfold a particular goal depends on the objective of the specialization. Even if the aim is an improvement in efficiency (an optimized residual), the optimality must be measured against some criteria. Further, a program that is optimal in one respect may not be in another. Consider, for example, the following initial program:

---

[2] ***B*** is a **trivial predicate** if there exists only a single clause defining ***B*** in the database, and that clause is known as a **trivial clause.**

```
ancestor( X, Y ) :- parent( X, Y ).
ancestor( X, Y ) :- parent( X, Z ), ancestor( Z, Y ).

parent( a1, a11 ).                parent( a1, a12 ).
parent( a11, a111 ).              parent( a11, a112 ).
parent( a12, a121 ).              parent( a12, a122 ).
```

Program 2.1: The *ancestor* program

With a control strategy of unfolding every possible goal and a query of *ancestor(X,Y)*, the following residual program may be obtained:

```
ancestor( a1, a11 ).              ancestor( a1, a12 ).
ancestor( a1, a111 ).             ancestor( a1, a112 ).
ancestor( a1, a121 ).             ancestor( a1, a122 ).
ancestor( a11, a111 ).            ancestor( a11, a112 ).
ancestor( a12, a121 ).            ancestor( a12, a122 ).
```

Program 2.2: First residual from the ancestor program

The result is a set of unit clauses. Another possibility is to unfold the *parent* predicate only, and the following program is obtained:

```
ancestor( a1, Y ) :- ancestor( a11, Y ).
ancestor( a1, Y ) :- ancestor( a12, Y ).
ancestor( a11, Y ) :- ancestor( a111, Y ).
ancestor( a11, Y ) :- ancestor( a112, Y ).
ancestor( a12, Y ) :- ancestor( a121, Y ).
ancestor( a12, Y ) :- ancestor( a122, Y ).

ancestor( a1, a11 ).              ancestor( a1, a12 ).
ancestor( a11, a111 ).            ancestor( a11, a112 ).
ancestor( a12, a121 ).            ancestor( a12, a122 ).
```

Program 2.3: Second residual from the *ancestor* program

If other control strategies are used, other residual programs may be obtained.

The problem is which of programs **2.2** or **2.3** is better? Both are longer than the original program. The first seems to be better because all queries will be solved through one reduction against a unit clause (no recursive resolution step). If there were many, many *parent* facts in the original program, however, the second residual may be better. As the number of *parent* facts increases, the number of clauses in the second residual will not grow as rapidly as the number in the first residual. Thus, in general, the second residual will take up less space than the first residual. The execution of the second may be longer, but only by a nominal amount. Thus one residual will be more efficient in space, and the other, in runtime.

From the literature, several identifiable decision procedures can be used at this stage. The most obvious and simplest is a **naive** approach; that is, (sub)goals are unfolded whenever possible. At the other extreme, unfolding may be driven completely by user input; the decision to unfold is **user-specified**. A compromise is to have a **selective** strategy. Criteria for the selective strategy may be user-specified, or based on the results of preprocessing, or incorporated into the partial evaluator *a priori*. If preprocessing is used, it often involves abstract interpretation to generate call-graphs, modes, data-flow graphs, or other types of information. An example of a predetermined, selective strategy is unfolding only when there is one clause for a goal. This approach does not alter backtracking behavior (between the original and residual), does not drastically increase the number of program clauses, and yields a modest reduction in the number of runtime resolutions. Its disadvantage is that it will often perform only a fraction of the unfoldings which are possible. Hence, it is called a **safe** strategy. Lakhotia [1989] suggests that unfolding rules can be extracted automatically from programs by examining mode information. In many cases, greater unfolding yields larger programs. Unfolding too little can miss possibilities for specialization [GaBr89]. In all cases, the decision of what or when to unfold must be guided by the type of specialization required of the partial evaluator.

In most instances, the primary purpose for partial evaluation is to decrease execution time — to improve runtime efficiency — by reducing the number of resolution steps taken at runtime. Thus, partial evaluators for this purpose are emphasized in the remainder of this study.

2.5.2. When to Stop

Even though the general unfolding strategy dictates that a goal is to be unfolded, correct operation may require that it not occur. Unfolding a goal may cause infinite looping on that goal. This is the purview of the loop detection component of a partial evaluator. Looping may occur even if runtime resolution of the goal terminates. Looping is sometimes the result of insufficiently bound arguments.

A variation of the *ancestor* program above (the unit clauses defining *parent/2* have been eliminated) is used to illustrate.

```
ancestor( X, Y ) :- parent( X, Y ).
ancestor( X, Y ) :- parent( X, Z ), ancestor( Z, Y ).
```

Program 2.4: A variation of the *ancestor* program

If it is partially evaluated with the query *ancestor(A,B)*, infinite looping occurs even though the definition of the predicate is not left-recursive. The subgoal *ancestor(A,B)* is unfolded to *parent(X,Z«),ancestor(Z«,Y)*. Since there are no clauses for *parent/2*, its unfolding is suspended and the *ancestor* subgoal is targeted. If there is no stopping mechanism, the latter is unfolded again. This sequence of steps continues repeating, and infinite looping results.

A partial evaluator can be equipped with an **input-driven** or an **automatic** loop detection mechanism. The input-driven approach relies on specifications of suspect or undesirable behavior from the user (manual input), or from a preprocessing stage. Manual input, by leaving the burden with the user, is the simplest approach, but is inconvenient. Also, the user may have insufficient or inaccurate knowledge of the input program or query, or not have the patience to input all the necessary information. Less often the input is provided by a preprocessing stage. Abstract interpretation is one mechanism for doing so.

A number of automatic loop-detection mechanism have been proposed, most notably the **strict-stopping**, **instance**, and **subterm-relation** methods. With the strict-stopping method, the unfolding of a goal is suspended if the same predicate name, with the same arity, has occurred earlier in the derivation. This approach is simple to implement, and is thus popular. It is safe — in fact, too safe. The method suffers from the **premature termination** problem: there are often times when processing of a goal should continue even though the stopping condition is met.

The second technique for automatic loop detection is called subterm relationship checking [FuFu88]. Suppose the resolution of a goal *p(x)* produces a sequence of recursive calls *p(x«)*, *p(x««)*, etc. If *x«* is a proper subterm of *x*, the unfolding of *p(x)* can be performed safely. If *x* is a vector of arguments or a structured structured term, this condition need only be satisfied at one fixed position in the argument list or term. This method is superior to the strict-stopping one, though premature termination can still occur.

Another indication of potential looping behavior is when the new call (of a goal) is an instance of a previous one. More formally, a loop is assumed to occur if during the partial evaluation of the clauses defining a goal ***G***, a goal ***G«*** — which is identical to ***G*** except for the names of variables, or is an instance (by unification) of the goal ***G*** — appears as a new goal to be partially evaluated. This is the instance-checking method.

An automated method of stopping (loop detection) can never be correct and complete in the general case since such a criterion would solve the halting problem for Prolog programs, and hence for Turing machines. As a result, any automatic stopping criterion is either too weak (does not stop on some infinite recursion), or too strong (stops too early on some finite recursion).

An extreme approach to the issue of when to stop unfolding is not having any loop-detection mechanism at all. This has been used in a number of systems, especially prototypes.

Finally, some partial evaluators are equipped with a hybrid solution: they use an automatic detection mechanism, but allow the user to override its decision. Given the current state of understanding of stopping criteria, this appears to be a good compromise: the difficulty in devising a good automatic loop-detecting mechanism is balanced against the inconvenience of user-input.

2.5.3. Passing Solutions

Another difficulty that must be addressed in partial evaluators is the manner in which results (substitutions for variables) are passed from one subgoal to another. What affect do multiple solutions for one subgoal have on the partial evaluation of another conjunctive subgoal? The answer influences the residual programs produced.

Consider the following example:

```
multiple( First, Last ) :- subgoal1( First, Middle ),
                           subgoal2( Middle, Last ).
subgoal1( a, b ).                    subgoal1( k, l ).
subgoal2( b, c ).                    subgoal2( l, m ).
```

Program 2.5: The *multiple* program

If this program is partially evaluated with the query *multiple(L,R)*, the optimal residual is simply the two unit clauses

```
multiple( a, c ).                    multiple( k, m ).
```

However, some existing partial evaluation systems regenerate the original program as the residual. The strategy they use is to assert the solutions of the first subgoal as unit clauses in the residual and then proceed to partially evaluate the second subgoal. This is known as the **naive** solutions passing strategy.

An alternative approach is to evaluate the remainder of a conjunction given each combination of solutions to the preceding subgoals. This is termed the **all-solutions** approach and is more computationally expensive. It can be accomplished in two ways. The first, assuming the partial evaluator is implemented in Prolog, is to use backtracking to try each solution in the remaining conjunction. The other is to explicitly create a list of solutions (as with the ***bagof*** predicate) to the leading subgoal, and to iteratively pass them to the remaining subgoals.

It is also possible to let the user specify what course of action should be taken for each of a set of solutions.

2.5.4. Optimizations

A number of optimizations can be performed during the partial evaluation process. Some common ones, corresponding to the simpler set of questions of Section 2.5, are as follows:

- removal of completely instantiated arguments
- removal of always-failing clauses
- removal of always-succeeding goals (from a clause)
- removal of trivial clauses
- removal of duplicate solutions

The question faced by the developer of a partial evaluator is whether or not to include these optimizations; whether they are cost effective. The implementation strategies for them are usually straightforward and not vigorously debated. As noted earlier, the last optimization may alter program semantics and is typically used with discretion.

All these optimizations can contribute to improved efficiency of the residual. For example, removing always-failing clauses reduces the size of the search tree, and removing always-succeeding goals reduces the size of the proof tree. Removing completely instantiated arguments reduces the number or complexity of runtime unifications. The optimizations can also reduce the space requirements of the residual. Removal of completely-instantiated arguments means less storage space for predicates and goals, both in stored program and at execution time. Removal of duplicate solutions results in fewer clauses in the residual.

2.5.5. Preprocessing

A number of strategies outlined above utilize preprocessing, often in the form of abstract interpretation. However, appropriate extent and depth of analysis are crucial to providing good input to the partial evaluator, and hence to the ultimate performance of the residual. Problems of control are reintroduced, then, at another processing step.

3. The Experiment

Many partial evaluation systems have been developed in the logic-programming community. The general objective of this work is to study a number of them in depth. The intent is to identify the control regimes used, to compare the partial evaluators based on the quality of their residuals, and to create a taxonomy or classification scheme to categorize them. The taxonomy is to be extensible, allowing for future refinement, as well as being applicable to other partial evaluators.

Five partial evaluation systems are examined in this work. Their performance on seven selected programs is compared. They are gauged according to the goodness of the residual programs they produce. This is determined through empirical measures (e.g. the efficiency of the residual programs) and a subjective measure.

3.1. The Partial Evaluators Examined

Five partial evaluators provide a representative, yet manageable, set of systems for examination. In selecting them, preference was given to crossing application and national boundaries. Priority was given to partial evaluators whose implementations are widely-available and in the public domain. Two of these were known. Three additional partial evaluators — for which implementations are not widely available — were also chosen. Preference was given to systems for which more detailed descriptions of their algorithms exist, and which are (possibly) better known. As source code was not available for the latter three, implementations in Quintus™ Prolog were performed locally according to the specifications given in the published descriptions. While the re-implementations cannot be guaranteed identical to the original systems, it was verified that they reproduce all behaviors specified or described by their authors. In some cases, the original authors were contacted for assistance.

Each of the partial evaluation systems is now described.

3.1.1. *Constraints*

The first partial evaluator is based on Fujita's algorithm [Fuji87]. It is termed ***constraints*** for purposes of this discussion. An implementation was made based on published documentation.

The partial evaluation technique here consists of two phases: a specialization phase and a shortcutting phase. The first phase takes the original program and query as input, and through unfolding operations, generates a set of specialized program clauses and a specialized query. Unnecessary (completely instantiated) arguments are also eliminated. This phase often results in redundancies: trivial clauses and unreachable clauses[3]. These can be revealed by considering some residual goals as constraints for other goals, and by checking the consistency of those constraints. For simplicity, constraints are restricted to equalities and non-equalities.

From the algorithm s description, it was ambiguous what solutions-passing strategy was to be used. A naive strategy was adopted in the implementation.

3.1.2. *Pure*

The second partial evaluator considered (henceforth called ***pure***) is based on the concepts presented by Kursawe [Kurs88]. A call (a goal) is interpreted as searching a search tree spanned by the call and all applicable rules until an expression is reached which represents a solution. Partial evaluation can then be performed by extracting clauses from the **compact search tree** to form the residual program, where the compact search tree is the usual search tree with all the succeeding goals removed. Fortunately, the compact search tree need not be explicitly built. Some shortcutting mechanisms can be used to determine the information in the tree without actually constructing it. A goal is unfolded when it is not a node in the compact search tree. This strategy can, in some cases, result in a goal not being unfolded even though it is safe to do so.

A system corresponding to the published description was implemented locally. Although Kursawe explicitly suggests that some kind of loop detection mechanism be present, there is no description of what it should be. An arbitrary decision was made to incorporate a method based on subterm-relation in the local implementation.

3.1.3. *ProMIX*

The next partial evaluator is a widely-available system called ***ProMIX*** [Lakh89]. *ProMIX* is driven entirely by unfolding rules, instructions to the system indicating when to unfold. Currently, the unfolding rules are user-supplied. However, the system s author suggests that they could be developed automatically using mode information, with manually-specified rules providing an override capability. He also suggests that only recursive predicates need special unfolding rules, that others can always be unfolded safely. The system distribution includes the unfolding rules for specializing interpreters.

A deliberate shortcoming of *ProMIX* is its absence of a loop-detection mechanism. This design decision was made because of the extra overhead of maintaining a call history , and because of the intent to eventually incorporate the feature by partially evaluating *ProMIX* with respect to a loop-detecting meta-interpreter [Lakh89]. Because of this shortcoming, *ProMIX* may loop endlessly when presented with a recursive program as input. A workaround is to simulate the strict-stopping method through the use of meta-rules. A new version of *ProMIX*, containing a stopping mechanism, has since become available.

---

[3] **Unreachable** clauses cannot be called by any goal in the set of specialized clauses.

3.1.4. *Mult*

The fourth partial evaluator is derived from the automatic compilation tool of the knowledge-base management system, EPSILON [LeSa88]. It is based on a multiple partial evaluations algorithm and was designed for meta-level programming. A local implementation was completed, and referred to here as ***mult.*** The original partial evaluator was intended for full Prolog; the implemented version, while using the same algorithm, only deals with pure Prolog.

The technique used in *mult* is to try to unfold all possible goals encountered, unless the goal is a recursive call or a primitive call which cannot be evaluated. In the first case, the partial evaluator attempts to fold the subgoals remaining in the conjunct into a new goal, and perform a separate partial evaluation on it.

3.1.5. *Peval*

The final partial evaluator considered was implemented by Takeuchi [TaFu86] and was one of the earliest widely-distributed Prolog partial evaluators. The system is completely user-driven; that is, all unfolding and specialization is done according to the user s instructions. These instructions can be provided through meta-rules added to the input program, or by interactive input during the partial evaluation process. This system is termed ***peval*** for purposes of this discussion.

3.1.6. Language Restrictions

This study is concerned with partial evaluation in the context of pure Prolog. However, *ProMIX* and *peval* support the full language. Also, for reasons outlined in the next section, rigid adherence to the pure subset limits the range of potential test programs. Hence, support for a small set of common evaluable built-in predicates was included in each of the three locally implemented systems. These predicates are *=/2*, *\==/2*, *</2*, and *</2* (i.e. tests for unifiable, not identical, less than, and greater than). Occurrences of *=/2* are reduced at partial evaluation time only if either argument is ground; i.e. the subgoal is effectively an assignment or test. Occurrences of the other predicates are reduced only if both arguments are ground. Otherwise, processing of the subgoal is suspended, and it is simply retained in original form in the residual. This extension can be seen simply as an additional optimization step, and the language of input programs still restricted to pure Prolog.

3.2. Test Programs

Seven test programs are used to gauge the performance of the five partial evaluators. The programs and input queries were chosen to demonstrate a variety of situations, programming techniques, and styles. Preference was given to shorter and more familiar programs. Represented are recursion-intensive, deterministic, and nondeterministic programs, meta-programs, and infinite-recursion situations. The programs, and the queries for which they were specialized, are described in this section. Listings of the programs are given in Appendix A.

The choice of suitable candidate programs was complicated by the restriction to pure Prolog. This became especially true when considering longer programs; as real programs get longer, they tend to incorporate more impure features. Thus, as described above, the partial evaluators were extended (where necessary) to treat specially four system predicates. These allowed the set of candidate test programs to be enlarged. Some of the test programs take advantage of this extension.

The first program, referred to as ***relative***, is the familiar relative relation defined in terms of ancestors, parents, mothers and fathers. The program was specialized for the query

```
relative( john, Who )
```

The program is nondeterministic, and recursion- and database-intensive for such a query. The problem size is easily increased by adding more of unit clauses for *mother/2* and *father/2*.

Two string-matching programs, ***match*** and ***contains***, were chosen. Both determine whether a substring (pattern) is wholly contained within a larger string, were strings are simply lists of terms. *Match* follows the common definition for such a predicate: starting at the current position in the searched string, try to recursively match terms with the substring. If the attempt fails, move along one position in the searched string, and try again. *Contains* follows a much more complicated strategy. It also tries to recursively match terms of the pattern with those of the searched string. However, on failure, the search is repositioned within the searched string using nondeterministic calls to append. The queries given at partial evaluation time were

```
match( [a,a,b], BigString )
```

and

```
contains( [a,a,b], BigString ).
```

For these queries, *match* behaves deterministically and has a shallow search tree. In contrast, *contains* is nondeterministic and has a much larger search tree.

The fourth program is a straightforward implementation of an algorithm to perform matrix transposition, where a matrix is represented by a list of rows, and each row is, in turn, a list of elements. The program is referred to as ***transpose***. For the query

```
transpose( [[A,B,C,D,E,F,G,H,I],R1,R2], Transpose).
```

the program is deterministic and deeply recursive.

The next program, ***ssupply***, is an instance of the supplier-parts-supplies database problem popularized by Kowalski [Kowa77]. The predicate *ssupply(SN,PN,Qty)* is true if a supplier *SN* is one of the best suppliers who can provide in quantity *Qty* red, heavy parts with part number *PN* which are needed in London. Its definition follows the familiar generate-and-test style. For most queries, the program is nondeterministic, solution-intensive, and database-intensive.

Since one of the partial evaluators (*mult*) was specifically designed to support a meta-programming technique, it is appropriate to have as a test case a simple meta-interpreter. The interpreter chosen computes the depth of the proof tree for a goal. Hence, it is called ***depth*** for purposes of discussion. Successor arithmetic was used to avoid the need for arithmetic, built-in predicates. The interpreted program, *member/2* defined using the familiar *append/3* predicate, was encoded as part of *depth*. The interpreted program operates nondeterministically.

A common application of Prolog is for grammars. The final test program chosen, ***grammar***, was the translation of a definite clause grammar for a restricted functional language. The full language and grammar were developed for another project [McKu88]. By restricting the target language to a only certain types of expressions involving specific constants and identifiers, the program corresponding to the grammar rules required no extra-logical features. The top-level predicate defined is

```
expression( Term, Qualification, HeadOfString, RestOfString )
```

and names the relation the expression in the string *HeadOfString* less *RestOfString* parses to *Term* with qualifiers *Qualification* The program was partially evaluated with respect to the query

```
expression( n, [], String, [] )
```

Such a query causes the program to run backwards , generating strings which parse to the term *n*. The program behaves nondeterministically for such a query. An outline of the grammar (rules only sufficient to deal with the above query) is given in Appendix A.

3.2.1. Supplemental Case

A very large pure Prolog program, an implementation of the Data Encryption Standard (DES encryption algorithm) [BenA89], was also tested. The program is long (151 clauses) and makes no use whatsoever of built-in Prolog data structures or predicates. The DES algorithm s computations are performed in fixed-length bit strings, requiring only permutation and bitwise exclusive-OR operations. The program was originally proposed as a benchmark case for Prolog implementations.

Unfortunately, this program did not differentiate between the partial evaluators; all but *constraints* produced the same residual (modulo renaming of variables and predicates) for all the queries tried. However, due to the size of the program, it did serve to highlight differences in the execution speed of the systems.

3.3 Queries and Other Input

For each of the programs, the query given to the residual at runtime was an instance of the query given to the partial evaluator. The queries used are given in Table 3.1.

Program	Partial-evaluation Time Query	Runtime Query
ancestor	*relative(john, Who)*	*relative(john, peter)*
match	*match([a,a,b], BigString)*	*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])*
contains	*contains([a,a,b], BigString)*	*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]))*
transpose	*transpose([[A,B,C,D,E,F,G,H,I],* *R1,R2], 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)*
ssupply	*ssupply(Supplier, Part, Quantity).*	*ssupply(s3, p1, Quantity)*
depth	*depth(member(X,* *[a,b,c,m,d,e,m,f,g,m,i,j]), D)*	*depth(member(i,* *[a,b,c,m,d,e,m,f,g,m,i,j]), D)*
grammar	*expression(n, [], String, [])*	*expression(n, [],* *[(, (, (, (,n,) ,) ,) ,)],* *[])'*

Table 3.1: Queries given during analysis

The query for partial evaluation of *grammar* was specially chosen to exercise the loop-handling capabilities of the systems.

Peval and *ProMIX* require user-specified unfolding rules. For the most part, unfolding of all goals was indicated. One exception is for the *depth* program and *ProMIX*. For this case the special facilities in *ProMIX* for handling meta-interpreters were utilized [LaSt89].

3.4. Performance Measurement

Each of the preceding programs was processed by each of the partial evaluators. The residuals were then examined for goodness . Both qualitative and quantitative measures were taken. The former involved visual inspection and a subjective characterization of the residual's quality. The quantitative measures taken were the size of the residual (number of clauses) and its efficiency for solving a specific goal. The efficiency involved the number of inferences (size of the search tree), the number of deductions (size of the proof tree), and the total runtime (execution speed). The counting of inference and deduction steps was done by a meta-interpreter. Each quantitative measure was also obtained for the original program with the same query. Quintus Prolog was the implementation platform.

Runtime was measured as follows. A standalone SUN™-3/50 workstation with 8MB of main memory and running SUN OS V3.5 was the testbed machine. The measurement for a particular residual was taken over a single, continuous time period when the workstation was otherwise idle. Three trials were run, and an average taken. For some queries, runtimes were too short to be accurately measured given the resolution of the runtime information available from the Prolog system. To obtain more accurate measurements, the query was repeated multiple times. The aggregate time was then divided by the repetition count to obtain a final per-query value. The Prolog system was given memory resources sufficient to prevent garbage collections during solution of the query.

The runtime performance of the five subject partial evaluators was not measured. Such a measure can be misleading, given the inability to control differences in implementation in the experiment. Different implementers were responsible for the various partial evaluators. The local re-implementations may be less efficient than their authors' original. Thus, differences in runtime may only reflect differences between implementers, rather than differences in inherent algorithm quality.

4. Results and Observations

The five partial evaluation programs were examined and tested in the manner described above. This section reports the results: the proposed classification scheme and the goodness measures.

4.1. The Taxonomy

By examining how the partial evaluation systems solve the three basic problems outlined in Section 2.4, an initial taxonomy for comparing partial evaluators can be formed. It is given in Table 4.1.

System	When to Unfold?	Loop Detection	Solution Coupling
constraints	all possible goals	subterm relation	naive*
pure	goal is not an internal node in the compact search tree	subterm relation*	complete
ProMIX	user-specified (meta-rules)	strict stopping (meta-rules)	complete
mult	all possible goals	instance checking	complete
peval	user-specified	instance checking	complete

* This approach was adopted in the local implementation. As noted earlier, the exact mechanism was not specified by the system s author.

Table 4.1: Classification according to control strategies

An ancillary taxonomy is due to the second, easier group of decisions and given in Table 4.2.

System	Eliminate Bound Arguments	Eliminate Succeeding Goals	Eliminate Trivial Clauses	Eliminate Failing Clauses	Eliminate Duplicate Solutions
constraints	yes	yes	yes	yes	n/a
pure	no	n/a	yes	yes	n/a
ProMIX	no	yes	yes♦	n/a	no
mult	no	n/a	yes	n/a	n/a
peval	no	yes	yes♦	yes	yes

The published description did not consider this issue. However, the feature was included in the locally implemented system.

The published description did not consider this issue. The feature was not included in the locally implemented system.

♦ Dependent upon user input.

Table 4.2: Classification according to elimination

Other partial evaluators can easily be classified according to this scheme by subjecting them to the same type of analysis and answering the same set of questions. Also, this taxonomy can be extended by considering other optimization steps or other aspects of the general partial evaluation process. These are discussed further in the section on future work.

4.2. Goodness Results

With the seven test programs and queries as input to the five partial evaluation systems, thirty-five residual programs were produced. In this section, the relative performance of the five systems is analyzed from the perspective of both visual inspection and quantitative results.

When testing the residuals, a check was made to ensure that they gave the same set of solutions as the original program for the stated runtime query (see Table 3.1). For the most part, the output solutions were identical, though some residuals gave alternate numbers of answers (having removed duplicates) or produced the solutions in different order.

4.2.1. Visual Inspection

Table 4.3 gives subjective evaluations of the residual programs produced. They are based upon visual inspection only. The goodness reflects the apparent amount of specialization which was performed.

	System				
Program	*constraints*	*pure*	*ProMIX*	*mult*	*peval*
relative	little specialization	good code (set of solutions as facts)	good code (set of solutions as facts)	some specialization	good code (set of solutions as facts) but no duplicates
match	good code	good code	original (no specialization)	some specialization plus much of original	moderately good code
contains	little specialization	good code	original (no specialization)	little specialization	moderately good code
transpose	little specialization	good code (single unit clause)	good code (single unit clause)	good code (single unit clause)	good code (single unit clause)
ssupply	some specialization	good code (set of solutions as facts)	good code (set of solutions as facts)	some specialization but much bigger	good code (set of solutions as facts)
depth	some specialization	good code	good code	good code	good code
grammar	good code	good, but not complete, code	good, but not complete, code	good, but not complete, code	good code

Table 4.3: Quality of Residuals Based on Visual Inspection

For the test case *relative*, *constraints* gives up (does not do any specialization) almost immediately due to the naive solution-passing strategy it employs. *Mult* manages to do some specialization work. The other partial evaluation systems perform well, settling upon a few ground clauses. The original *relative* program produces many identical answers. Some of the systems (*constraints*, *mult*, and *ProMIX)* retain these, while two (*peval* and *pure*) change the program's semantics and eliminate them. In the case of *pure*, this was a local implementation decision.

For the *match* program, *ProMIX* does not do any specialization due to its simplistic loop-detection mechanism. The meta-rules introduced to prevent looping are strict-stopping and prevent unfolding. The subgoal *match/4* in the *match* program can be unfolded safely in some cases. *Mult* expands all subgoals involving *match/2*, but gives up when *match/4* is encountered. *Constraints*, *peval*, and *pure* manage to expand both *match/2* and *match/4* for all cases.

Contains is the most troublesome program; most of the partial evaluation systems have problems with it. Only *pure* and *peval* manage to do all possible specializations. The rest either give up at the very beginning (*ProMIX)*, or manage to do something, but stop afterwards (*mult* and *constraints*).

For the *transpose* program, *constraints* achieves little specialization due to the naive solution-passing strategy it employs. The other partial evaluators manage to do all possible specialization work, ending up with a one-clause program. The situation is similar for *depth*, except that optimal residual is three clauses long.

For the *ssupply* program, three partial evaluation systems are able to perform all permissible specialization steps and produce residual programs with three ground clauses. *Mult* s special control strategy introduces extra dead code which increases the size of the residual dramatically. *Constraints* achieves little specialization and was not able to capitalize on the multiple solutions available for one of the subgoals.

Grammar is an interesting example as it highlights a common problem with implementations of loop-detection strategies. Three of the partial evaluation systems (*ProMIX, pure,* and *mult*) diagnose a loop and terminate the unfolding process. However, clauses to solve the subgoal upon which unfolding terminated are not included in the residual. For example, a clause similar to the following is generated by each of these three partial evaluators:

```
    expression( n, [], Str, [] ) :- value( n, Str, [] ).
```

A loop was suspected during the unfolding of *value/3*, so either no clauses are placed in the residual for the *value/3* subgoal, or the clauses that are included are insufficient for resolution of the original query. Hence, clauses necessary to resolve *value(n,Str,[])* and its recursive subgoals must be added manually to the residual. *Peval* and *constraints*, on the other hand, produced well-optimized code without manual intervention.

4.2.2. Empirical Measures

In this section quantitative measures of the residuals are given. The measures are the number of clauses in the residual and its performance in solution of a runtime query (as given in Table 3.1). The performance metrics are: the number of deductions in the proof tree, search tree size, and the execution time. They were gathered as set out in Section 3.4.

The measures are given in two forms: tables and graphs. The tables give the raw data. For the graphs, the performance of the original program is not included. Rather, all data is normalized with respect to the performance for the original program.

Source Size		**System**				
Program	*original*	*constraints*	*pure*	*ProMIX*	*mult*	*peval*
relative	15	17	10	21	36	10
match	4	9	7	4	7	8
contains	7	13	10	7	15	24
transpose	6	14	1	1	1	1
ssupply	30	23	3	3	45	3
depth	9	11	12	3	12	12
grammar	82	7	67	65	65	5

Table 4.4: Source Size of Residual Programs

Proof Tree Size		**System**				
Program	*original*	*constraints*	*pure*	*ProMIX*	*mult*	*peval*
relative	22	20	1	1	16	1
match	50	49	49	50	48	45
contains	76	74	19	76	48	8
transpose	50	46	1	1	1	1
ssupply	20	9	1	1	8	1
depth	25	25	1	12	1	1
grammar	30	8	26	30	30	5

Table 4.5: Number of deductions (Proof Tree Size) for Residual Programs

Search Tree Size		**System**				
Program	*original*	*constraints*	*pure*	*ProMIX*	*mult*	*peval*
relative	112	88	2	2	73	2
match	59	56	50	59	57	46
contains	109	109	30	109	57	9
transpose	69	63	2	2	2	2
ssupply	25	19	2	2	8	2
depth	38	38	2	13	2	2
grammar	160	17	146	77	77	6

Table 4.6: Number of inferences (Search Tree Size) for Residual Programs

Runtime		**System**				
Program	*original*	*constraints*	*pure*	*ProMIX*	*mult*	*peval*
relative	82.333	51.105	1.661	1.900	47.400	2.111
match	124.239	75.539	123.850	123.920	125.167	109.739
contains	184.384	179.739	59.894	183.070	159.556	43.056
transpose	97.611	97.139	2.394	2.428	2.394	2.400
ssupply	26.572	5.489	1.295	1.287	3.644	1.322
depth	54.210	55.590	4.455	26.640	4.455	4.455
grammar	94.822	8.272	96.739	65.533	65.533	3.655

Table 4.7: Runtime (in msec.) for Residual Programs

User-specified unfolding rules are necessary for *peval* and *ProMIX*. Consequently, the performance of these systems, as measured by the efficiency of the residuals, can vary depending on the rules given. For example, for *ProMIX* and the *depth* program, the residual was produced by the recommended facilities accompanying the system for handling meta-interpreters. However, if simple unfolding of all goals is specified, the residual is the same as that for *mult* (and *ProMIX* would then show the same performance as *mult* for this case). Another example is *peval*, again for the *depth* program. The residual was produced by specifying that all goals be unfolded. However, the original paper suggests user input for dealing with meta-interpreters. If it is used, the resulting residual is much longer and has poorer runtime performance.

These results are more easily grasped when presented in graphical form. The dependent axis gives a normalized form of the measurement in each case: it is expressed as a ratio of the value for the original program. Since the original program would have 1.0 as its value in each trial, it is not included in the graphs. Two groups of graphs are given. The first graphs, in Figures 4.1 through 4.4 have the different test programs on the independent axis. In the second group, Figures 4.5 through 4.8, the independent axis enumerates the different partial evaluation systems.

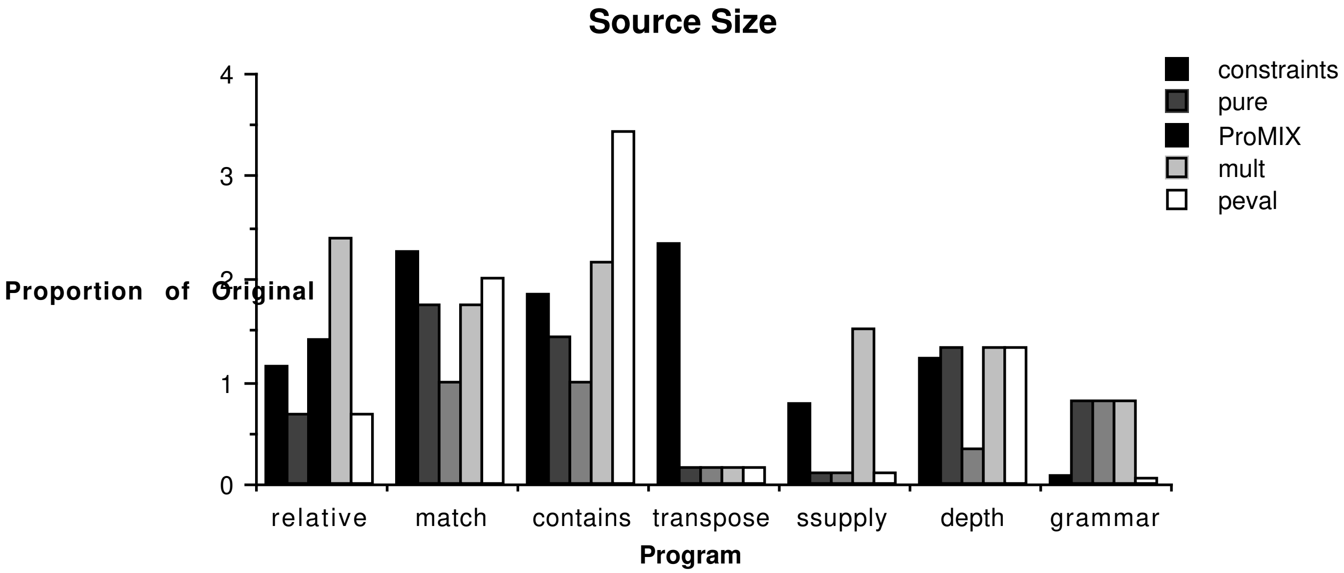

Figure 4.1: Source Size of the Residuals

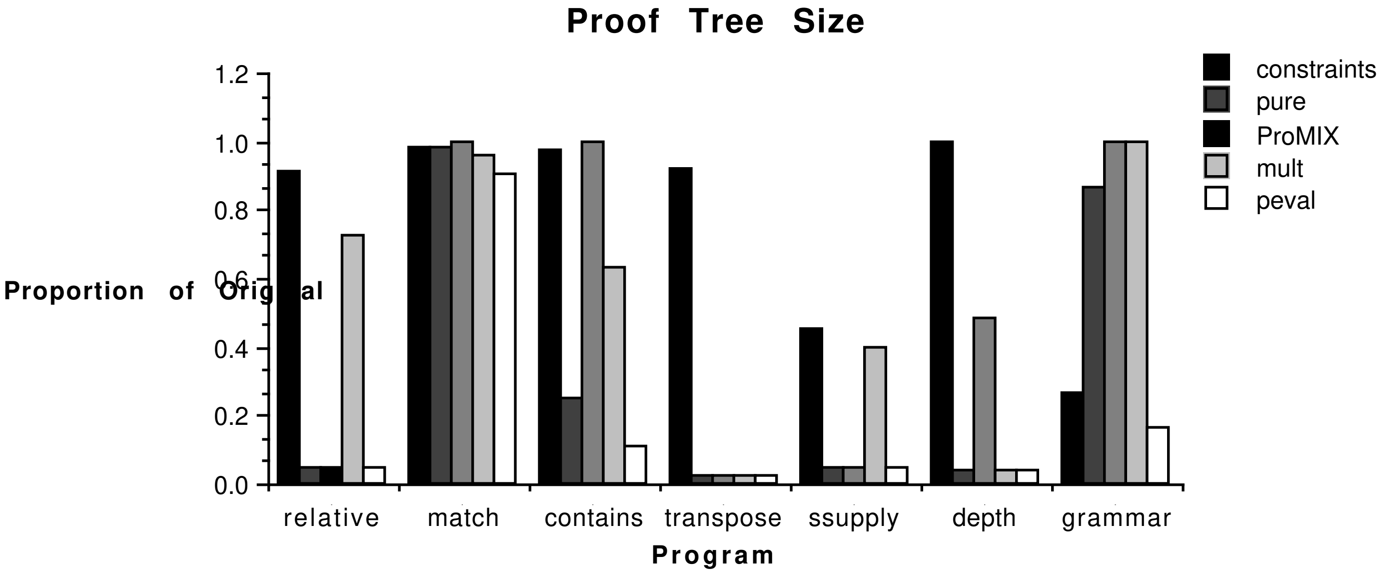

Figure 4.2: Number of Deductions (Proof Tree Size) for the Residuals

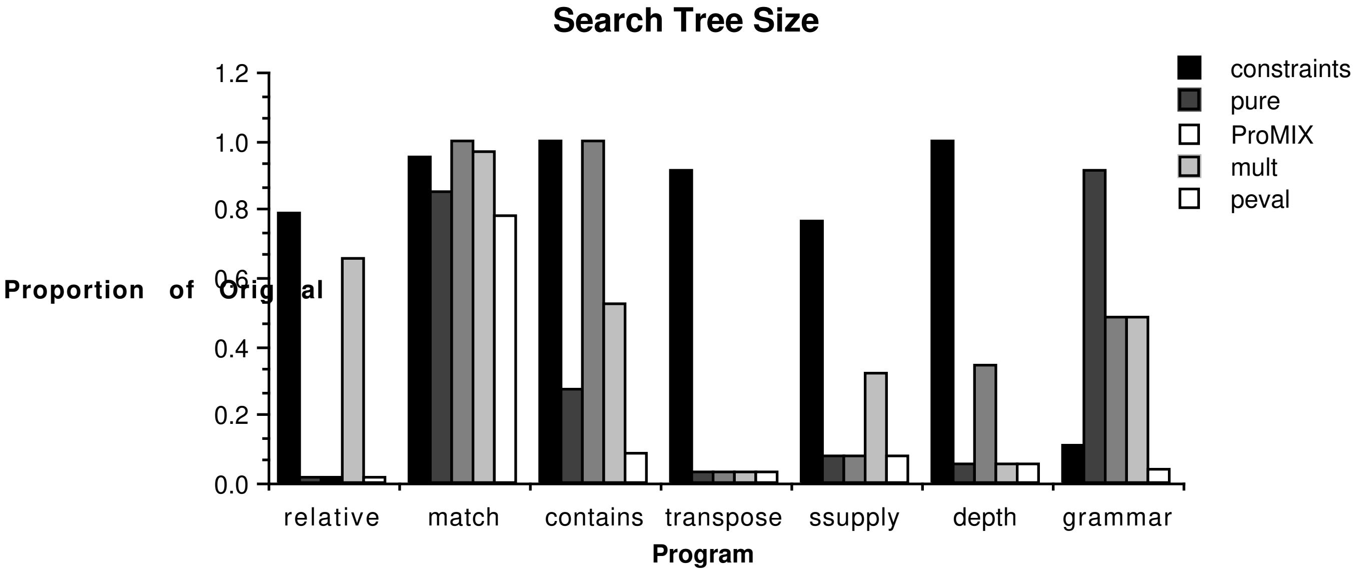

Figure 4.3: Number of Inferences (Search Tree Size) for the Residuals

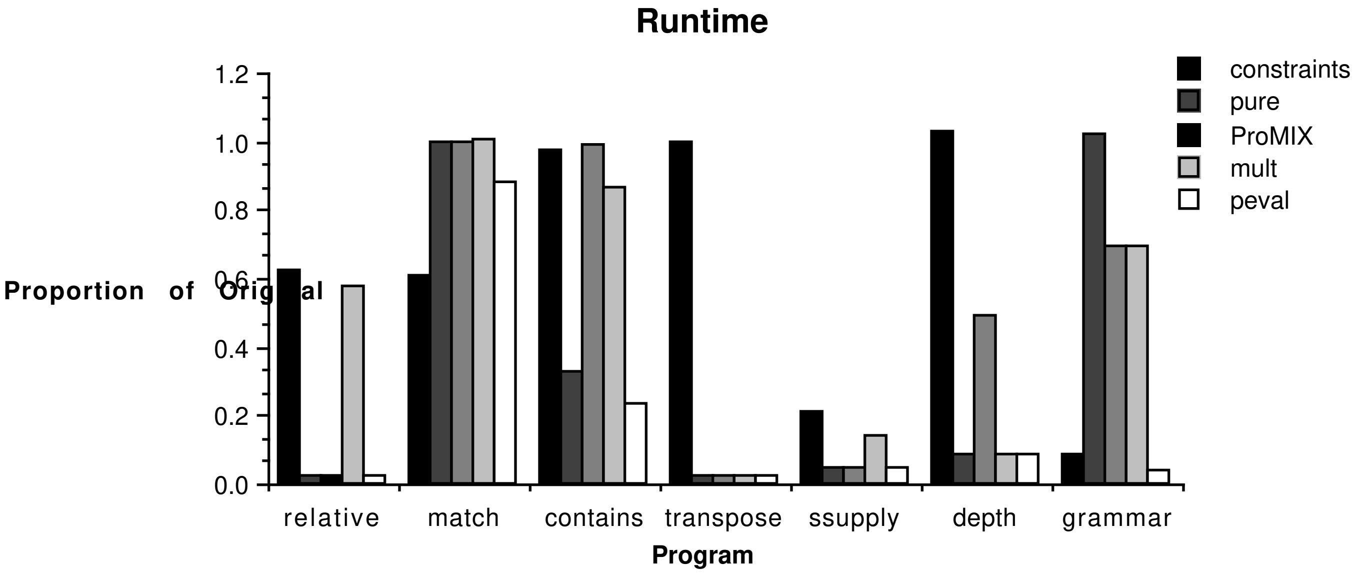

Figure 4.4: Runtime of the Residuals

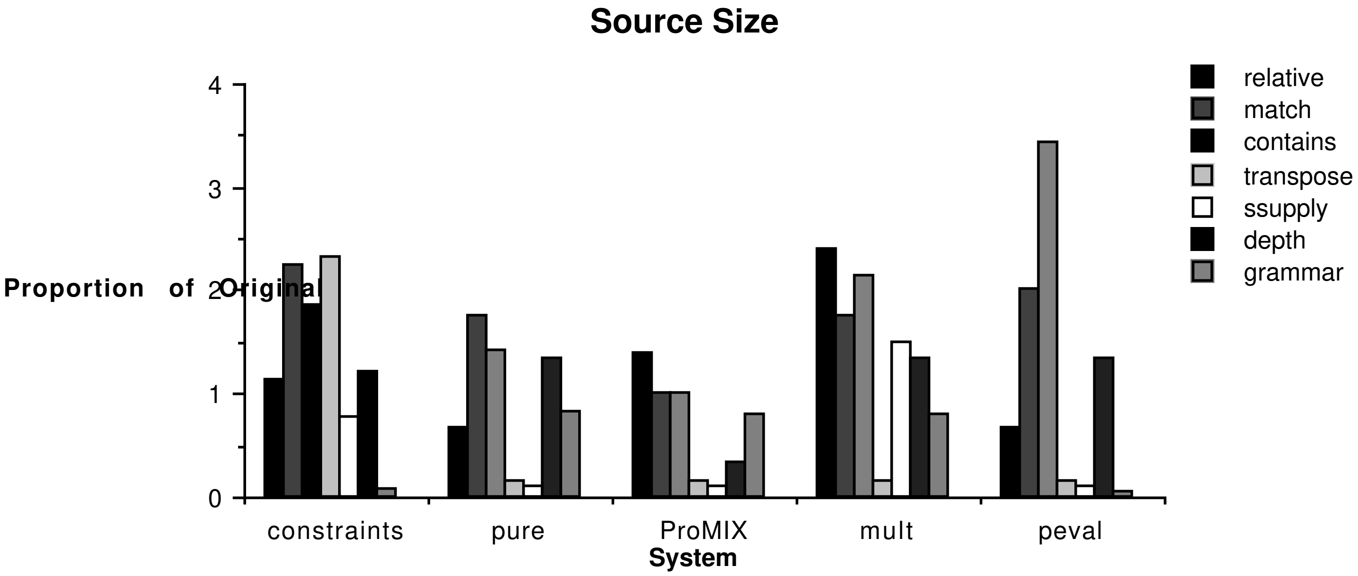

Figure 4.5: Source Size of the Residuals

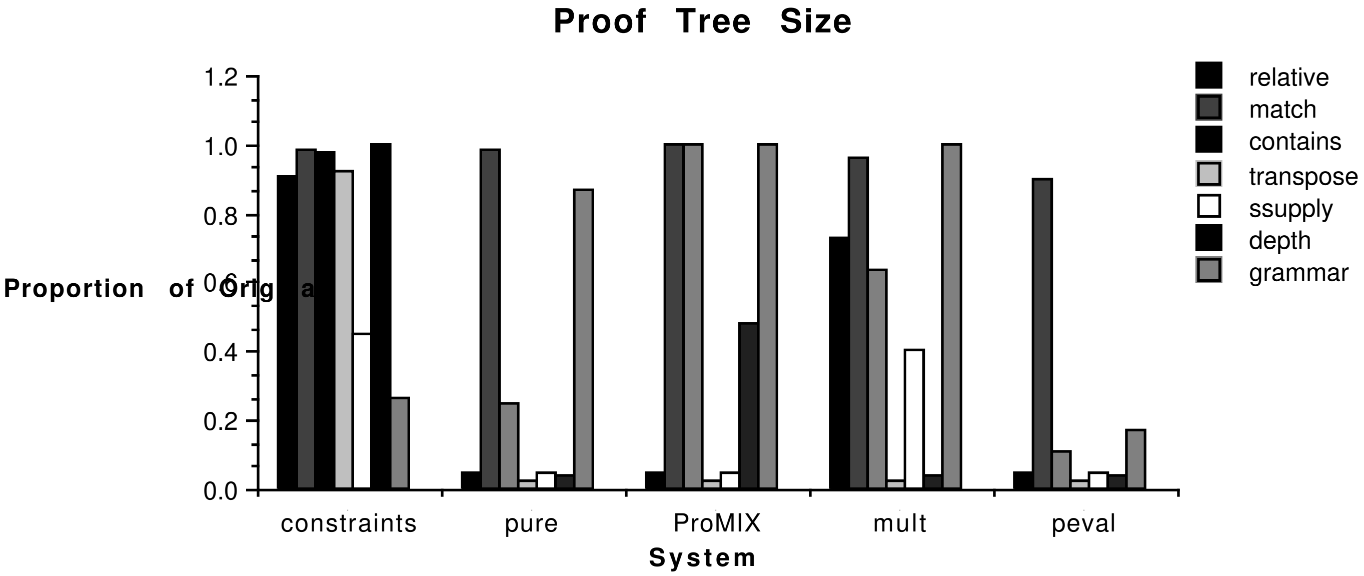

Figure 4.6: Number of Deductions (Proof Tree Size) for the Residuals

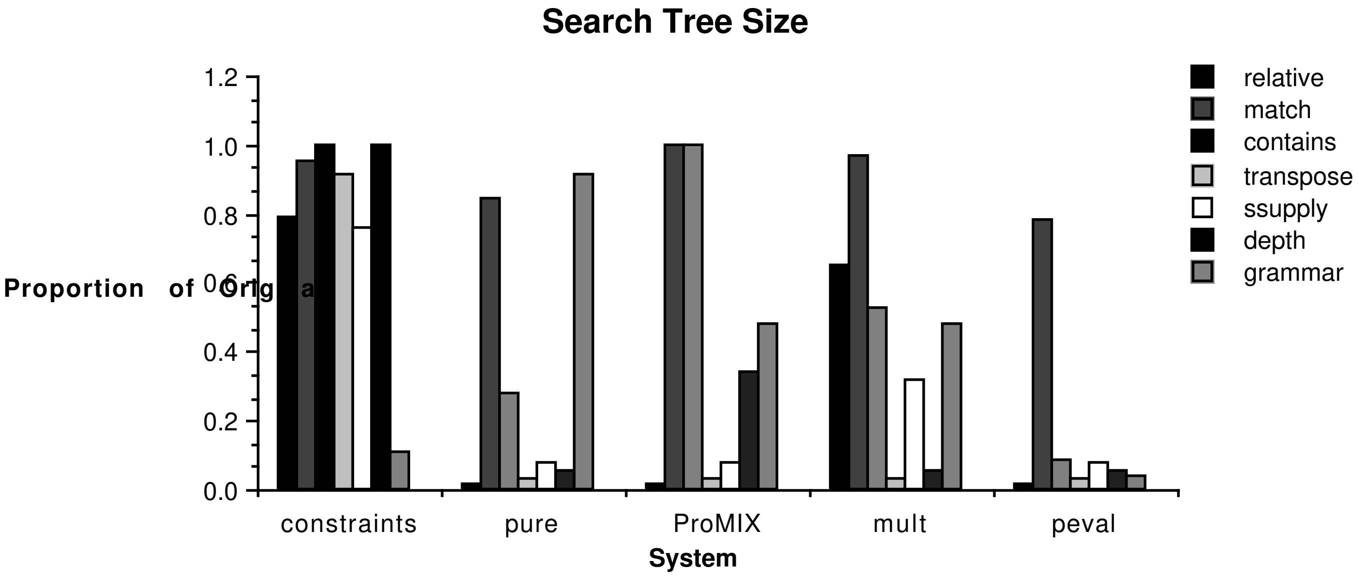

Figure 4.7: Number of Inferences (Search Tree Size) for the Residuals

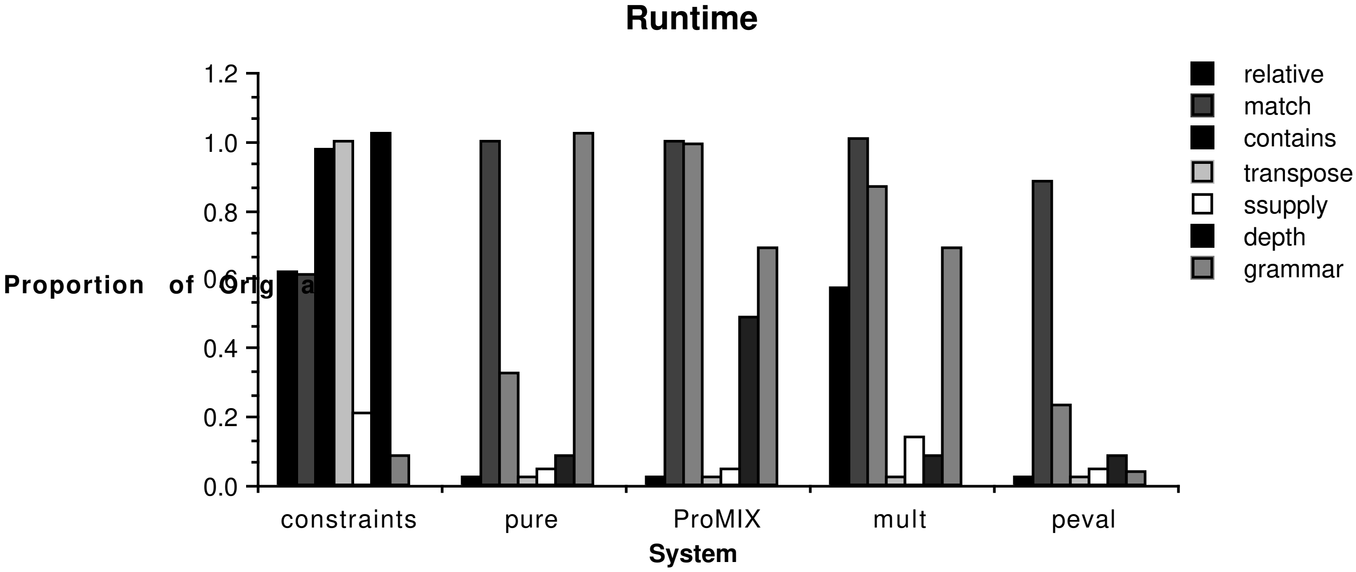

Figure 4.8: Runtime of the Residuals

4.3. Comparative Summary

The performance of the five partial evaluators is discussed based on the tabular and graphical results in the previous section.

The partial evaluators, except for *constraints,* tend not to be able to improve the efficiency of *match.* In fact, the residual programs produced by those four are almost the same. *Match* is concise and written in such a way that little improvement is possible. *Constraints* produces a residual in which bound arguments are eliminated. On the other hand, almost all the partial evaluators are able to achieve significant efficiency improvement in the case of *transpose*. Again the exception is *constraints*, likely due to the naive solution-passing strategy employed in the local implementation.

The partial evaluation systems all tend to produce residual programs which are longer than the original, except when a collection of unit clauses (constituting the solution) can be generated. They succeed in improving runtime efficiency: the residuals are at least as fast as the original, and often times faster. A correlation between search tree size, proof tree size, and runtime for the residuals is evident in the graphs; smaller trees are reflected in generally shorter runtimes.

According to the residuals, *constraints* performs poorer than the other partial evaluators. This may be due to the naive solution-passing strategy used in the local implementation. The only case where *constraints* does well is with the *match* program where it eliminates extra bound arguments from predicate arguments, cutting down runtime of the residual.

ProMIX gives little improvement for the *match* and *contains* test cases. This is likely due to *ProMIX* not having a real loop-detection mechanism, and a strict-stopping method being simulated through user-specified unfolding rules.

Mult appears to perform moderately well. The special control strategy it employs introduces additional overhead or extra, dead code in some residual programs. It does, however, perform optimally for the meta-programming case *depth*, the type of application for which it was specifically designed.

The residuals produced by *peval* are generally the most efficient in terms of runtime. They perform as well or better than those produced by *pure* and *ProMIX.* However, *peval* s residuals tend to be longer than those of *pure* and *ProMIX. Peval* s performance is entirely dependent on the quantity and quality of user guidance. In this experiment, all possible unfolding is specified, and it performs well. However, having to specify various control input is annoying, cumbersome, and — at times — error prone.

The residuals produced by *pure* are almost as efficient as those of *peval.* It does well in most cases, especially in terms of code size. *Pure* does not require any user guidance. However, *pure* does not unfold goals as often as it could.

The processing times of the partial evaluators was not reported. It was observed, however, that reifying unification and nondeterministic searching (i.e. backtracking) requires more CPU and memory resources. Partial evaluators which take advantage of the operations of the underlying Prolog system tend to require fewer resources for the same inputs. Extra checks and eliminations for optimization also slow a partial evaluator.

5. Generalizations and Conclusions

Although the goal of the five partial evaluation systems is the same, the residuals they produce can differ significantly. This is due to different decisions being made during the specialization process. These relate to the answers to the three basic questions: When or what to unfold? When to terminate unfolding? How to pass solutions? Additional optimizations also have an effect.

Most partial evaluators in this sample do not eliminate bound values, although the S-m-n theory [Klee52] deliberately suggests so. As demonstrated by *constraints* and the *match* example, this step can decrease the runtime of residual programs significantly.

Most of the partial evaluators manage to traverse all possible branches of the search tree. As suggested by *constraints,* the naive solution-passing strategy, in which bindings are not passed to other subgoals, prevents some branches of the search tree being traversed. This contributes to *constraint* s poorer performance. Thus, the strategy used for passing multiple answers from one conjunctive subgoal to another greatly affects the residual programs produced. Poorer strategies handicap some of the specialization processes and lead to semi-specialized residuals.

Some partial evaluation systems unfold all possible goals. This decreases the runtime of the residual program significantly since the corresponding resolution steps are not be performed at runtime. However, it may also dramatically increase the size of the residual (over the original).

Loop detection is an important criterion in partial evaluation, possibly the most important one. Without a stopping mechanism a partial evaluator cannot deal with recursive programs. Therefore it is fundamental in the context of Prolog. The stopping mechanism affects the degree of specialization. For instance, partial evaluation systems with strict-stopping mechanisms often perform little specialization. An example are the *grammar* and *contains* cases under *ProMIX.* Many partial evaluators have more sophisticated mechanisms. However, no general stopping criterion for all cases is known. Finding a good one will be one of the hardest problems to solve in the field of partial evaluation in logic programming. One advantage of user-specified stopping rules is that finer control, specific to a particular situation, can be achieved. A good compromise solution to this problem is to have a cautious, though automatic mechanism which can be overriden by user-supplied information. Just detecting a loop is not sufficient, however. Appropriate clauses for the goal initiating the loop must still be placed in the residual — without succumbing to the looping behavior.

Additional optimizations through eliminations (trivial clauses, etc.) are practical and beneficial in improving a residual. For example, removal of unnecessary bound arguments improves execution speed of a residual. This action makes the residual harder to understand and follow (clauses have less obvious meaning). However, users are usually not interested in reading and analyzing a residual; they are primarily concerned with its efficiency or form. For modularity, it makes sense to have the eliminations in a separate stage after unfolding. Although these operations can be carried out during unfolding, they would complicate that stage. The eliminations may in fact be harder and more expensive then. A separate module containing the optimizations is not difficult to implement.

Good, general purpose partial evaluators for (pure) Prolog are possible. *Pure* and *peval* seem to be the best of the five partial evaluators, at least for the implementations used in this experiment. However, *pure* can be considered better as it does not require user input. For short, uncomplicated programs *peval* is still attractive. (For long input programs the user would have to spend a long time determining and supplying the correct set of rules.) *ProMIX* can be considered a close second to *pure* and *peval*. Since it does not have a loop-detection mechanism[4], it may be more useful for less recursive programs. *Constraints* and *mult* seem to be more applicable to select types of program. For example, *mult* was designed for meta-level programming and it does perform well in that test case.

The classification scheme developed in this work is generally applicable. It allows major characteristics of partial evaluation systems for pure Prolog to be seen in comparison; the scheme identifies the major steps in partial evaluation. It also clearly indicates to developers features or questions others have considered. The shortcoming of the scheme is that it is not predictive. It sometimes gives a general indication how a partial evaluator may perform for a given type of problem, but not to a fine degree of accuracy. Fortunately, the classification scheme is extensible; additional columns can easily be added as necessary. It is therefore a good first step towards a more definitive taxonomy, and the three general questions identified are a good basis for a first classification of partial evaluators.

5.1. Future Work

Further investigation in this area is warranted. Some possibilities are given in this section.

The proposed taxonomy is only an initial one; it is intended to be extended in future. More criteria (corresponding to other important questions in the partial evaluation process) could be considered. More options per criteria are possible. Other optimizations may exist. All these can be used to further discrimination between systems.

The concept of **folding** is becoming more common in partial evaluators. However, it is not yet clear what impact the folding step has on the entire partial evaluation process. Should folding be considered one of the basic suboperations (as discussed in Section 2.5)? How does it affect the residuals? This research could be extended to take folding into account. The taxonomy could be extended with a further category, When or what to Fold? . Different folding strategies would then be identified and used for classification purposes. It may also be that termination of the folding process is a further issue (just as termination of unfolding has been shown to be).

An obvious pursuit is to broaden the scope of the study: more and other partial evaluators and test cases. This would provide more information on different control strategies and algorithms, as well as evidence to support (or not support) the proposed taxonomy. For example, partial evaluators employing breadth-first or limited depth-first unfolding could be studied.

Preprocessing, in particular abstract interpretation, is becoming more popular in partial evaluators [Fuji88]. The preprocessing can, for example, be aimed at determining when goals can safely be unfolded or determining the modes of various predicates. Subsequent research in this area should take such preprocessing stages into account.

Abstract interpretation has been used as a direct means for specialization [Gal+88]. The approach is one of building an abstract computation tree from the input program, and

---

[4] A version of *ProMIX* which includes a form of automatic loop detection is now available.

constructing the specialized program from this tree. Although this technique does not fully use all given information (like the query), promising results are reported. The analysis and comparison work reported here could be extended to include such partial evaluation techniques.

In general partial evaluation can improve the performance of Prolog programs. However, one would expect special circumstances (input programs or queries) where the residual is significantly less efficient than the original. One potential case may be where indexing is used to great advantage in compiling the original program, and the partial evaluation process disturbs the basis for this in the residual. As such a case was not included in this experiment, it remains an interesting aspect to explore.

The quantitative analysis can be made more systematic. Test cases worth considering, other than the index one above, include more large programs (similar or greater in size to *grammar* or the DES implementation), and multiple queries for the same input program.

Parts of the experiment could be redone. Ad hoc decisions were made as to certain features of the locally implemented partial evaluation systems (stopping mechanism, elimination of trivial or redundant information from the residual, etc.). Alternate decisions could be taken, and the results investigated. For example, the naive solution passing strategy in *constraints* could be replaced by a more intelligent one, and the test cases rerun. A new version of *ProMIX*, incorporating a loop-detection mechanism, now exists. The experiment should be rerun using it, and any differences in performance noted and explored.

Given that most Prolog programs are not restricted to the pure subset, it would be beneficial to extend this investigation to full Prolog, with both quantitative measures and comparative analysis. This study, and its results, could be used as an instructive guide.

A new partial evaluation system could be designed based on the comparative results of this research. The new system would use the best features of the five analyzed partial evaluators. A related idea is the construction of partial evaluators from pre-existing modules, where each module performs one subtask of the partial evaluation process according to a particular strategy or method. Different partial evaluators could then be realized through different combinations of modules. Modules could also be used to efficiently deal with the extra-logical aspects of full Prolog, or the differences between different Prolog implementations.

For some of the partial evaluators, a small change in strategy or mechanism can yield a large difference in the performance of the residuals it generates. A more thorough exposition on the cause(s) of these differences is warranted.

Possibly the most ambitious and long-term prospect is development of a predictive taxonomy.

6. Contributions and Final Remarks

Most research and effort in partial evaluation in Prolog (and logic-programming languages in general) has dealt with development of new or alternate systems and algorithms. Seldom have existing partial evaluators been analyzed and compared. This work provides a comparison of five previously known or available partial evaluation systems. The results are three-fold: comparative measures of the performance of the partial evaluators, an initial taxonomy or classification scheme, and the identification of possible control structures and algorithms. These are valuable to the future development and use of better partial evaluation algorithms and systems.

Partial evaluation has been used for specialization in logic programming for almost ten years. This has been long enough for it to become an accepted and widely-used technique. Numerous partial evaluation systems exist. However, researchers are still struggling to understand the technology better, and to produce more automatic systems with more general applicability. In early partial evaluation systems, the unfolding of individual goals was monitored and controlled by a human operator/user. Later systems automated the partial evaluation process as much as possible. However, lessening the degree of external direction requires a better understanding of the process. The realization of partial evaluators as mature, useful tools in logic-programming environments will require greater degrees of automation, and hence, analysis and understanding. The results reported here are another step towards that analysis and understanding.

Acknowledgements

We would like to thank John Gallagher, Jan Komorowski, Arun Lakhotia, Chris Moss, Steven Prestwich, and Dan Sahlin for their assistance, encouragement, comments and contributions. Thanks are also due Ian Olthof for suggesting the title.

Bibliography

[BenA89] M. Ben-Ari. An Extremely Pure Logic Program . Technion — Israel Institute of Technology and Brandeis University, 1989.

[Bj¿+88] D. Bj¿rner and A. P. Ershov and N. D. Jones, editors. *Partial Evaluation and Mixed Computation,* North-Holland, 1988.

[Fuch88] K. Fuchi, editor. *New Generation Computing*, 6(2—3), June 1988, Springer-Verlag.

[FuFu88] H. Fujita and K. Furukawa. A Self-Applicable Partial Evaluator and Its Use in Incremental Compilation . In *New Generation Computing*, 6(2—3):91—118, June 1988.

[Fuji87] H. Fujita. An Algorithm for Partial Evaluation with Constraints . Technical Memorandum TM-0367, I.C.O.T., Tokyo, August 1987.

[Fuji88] H. Fujita. Abstract Interpretation and Partial Evaluation of Prolog Programs . Technical Memorandum TM-0484, I.C.O.T., Tokyo, March 1988.

[GaBr89] J. Gallagher and M. Bruynooghe. The Derivation of an Algorithm for Program Specialization . Technical report, Department of Computer Science, Katholieke Universiteit Leuven, Heverlee, Belgium, 1989.

[Gal+88] J. Gallagher, M. Codish, and E. Shapiro. Specialization of Prolog and FCP Programs Using Abstract Interpretation . In *New Generation Computing*, 6(2—3):159—186, June 1988.

[Klee52] S. C. Kleene. *Introduction to Mathematics.* Van Nostrand, 1952.

[Kowa77] R. A. Kowalski. *Logic for Problem Solving.* North-Holland, 1977.

[Kurs88] P. Kursawe. Pure Partial Evaluation and Instantiation . In *Partial Evaluation and Mixed Computation,* D. Bj¿rner and A. P. Ershov and N. D. Jones, editors, pages 283—298, North-Holland, 1988.

[Lakh89] A. Lakhotia. Personal communications.

[LaSt89] A. Lakhotia and L. Sterling. How to Control Unfolding When Specializing Interpreters . Personal communication, 1989.

[LeSa88] G. Levi and G. Sardu. Partial Evaluation of Metaprograms in a Multiple Worlds Logic Language . *New Generation Computing,* 6(2—3):227—248, June 1988.

[LlSh87] J. W. Lloyd and J. C. Shepherdson. Partial Evaluation in Logic Programming . Technical Report CS-87-09, Department of Computer Science, University of Bristol, Bristol, England, 1987.

[McKu88] C. D. McCrosky and A. J. Kusalik. Improving First-Class Array Expressions Using Prolog . In *Proceedings, Fifth International Conference on Logic Programming*, R. A. Kowalski and K. A. Bowen, editors, MIT Press, pages 18—25, Seattle, Washington, August 15—19, 1988.

[SeSo88] P. Sestoft and H. S¿ndergaard. A Bibliography on Partial Evaluation . In *SIGPLAN Notices,* 23(2):19—27, February 1988.

[TaFu86] A. Takeuchi and K. Furukawa. Partial Evaluation of Prolog Programs and its Application to Meta Programming . In *Proceedings, IFIP-86*, H.-J. Kugler, editor, North-Holland, pages 415—420, Dublin, Ireland, 1986.

[TaSa84] H. Tamaki and T. Sato. Unfold/fold Transformations of Logic Programs . In *Proceedings, Second International Logic Programming Conference*, S.- . T rnlund, editor, pages 127—138, Uppsala, Sweden, 1984.

Appendix A: Test Programs

A.1. *ancestor*

```
relative( Person1, Person2 ) :-   ancestor( CommonAncestor, Person1 ),
                                  ancestor( CommonAncestor, Person2 ).

ancestor( Parent, Child ) :- parent( Parent, Child ).
ancestor( Ancestor, Person ) :- parent( Ancestor, CloserAncestor ),
                                ancestor( CloserAncestor, Person ).

parent( Parent, Child ) :- father( Parent, Child ).
parent( Parent, Child ) :- mother( Parent, Child ).

mother( carol, paulina ).            mother( carol, albertina ).
mother( albertina, peter ).          mother( maria, mary ).
mother( maria, jose ).               mother( mary, anna ).
mother( mary, john ).                father( zach, carol ).
father( zach, jonas ).               father( jonas, maria ).
```

A.2. *match*

```
match( Pat, Str ) :- match1( Pat, Str, Pat, Str ).

match1( [], _, _, _ ).
match1( [Term|_], [DifTerm|_], WholePat, [Discard|RestOfStr] ) :-
   Term \== DifTerm,
   match1( WholePat, RestOfStr, WholePat, RestOfStr ).
match1( [Term|RemPat], [Term|RemStr], WholePat, WholeStr ) :-
   match1( RemPat, RemStr, WholePat, WholeStr ).
```

A.3. *contains*

```
contains( Pat, Str ) :- con( Str, ( [], Pat ) ).

con( _, (_,[]) ).
con( [T|RemStr], PatInfo ) :-
   new( T, PatInfo, NewPatInfo ),
   con( RemStr, NewPatInfo ).

new( T, (Prefix,[T|RemPostfix]), (NewPrefix,RemPostfix) ) :-
   append( Prefix, [T], NewPrefix ).
new( T, (Prefix,[DifT|RemPostfix]), (NewPrefix,NewPostfix) ) :-
   T \== DifT,
   append( Prefix, [T], Temp ),
   append( NewPrefix, Rest, Prefix ),
   append( _, NewPrefix, Temp ),
   append( Rest, [DifT|RemPostfix], NewPostfix ).

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

A.4. *transpose*

```
transpose( Matrix, [] ) :- null_rows( Matrix ).
transpose( Matrix, [TransRow|RemTransMatrix] ) :-
   transpose_row( Matrix, TransRow, RemMatrix ),
   transpose( RemMatrix, RemTransMatrix ).

transpose_row( [], [], [] ).
transpose_row( [[Elem|RemRow]|Matrix],
               [Elem|TrRow],
               [RemRow|RemMatrix]
) :-
   transpose_row( Matrix, TrRow, RemMatrix ).

null_rows( [] ).
null_rows( [ [] | RestOfMatrix ] ) :- null_rows( RestOfMatrix ).
```

A.5. *ssupply*

```
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( p1, nut, red, 12, london ).
parts( p2, bolt, green, 17, paris ).
parts( p3, screw, blue, 17, rome ).

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 ).

good_cities( paris ).                 good_cities( london ).
good_cities( hongkong ).              good_cities( regina ).
good_cities( saskatoon ).

heavyweights( PNum, PName, Colour, Weight, City ) :-
   parts( PNum, PName, Colour, Weight, City ), Weight > 10.

london_heavyweights( PNum, PName, Colour, Weight ) :-
   heavyweights( PNum, PName, Colour, Weight, City ),
   City = london.

red_parts( PNum, PName, Weight, City ) :-
   parts( PNum, PName, Colour, Weight, City ), Colour = red.

london_red_parts( PNum, PName, Weight ) :-
   red_parts( PNum, PName, Weight, City ), City = london.

london_red_heavy_parts( PNum, PName, Weight ) :-
   london_red_parts( PNum, PName, Weight ),
   london_heavy_weights( PNo, PName, Colour, Weight ),
   PNum = PNo.
```

```
good_suppliers( SNum, SName, Status, City ) :-
   suppliers( SNum, SName, Status, City ), Status > 10.

best_suppliers( SNum, SName, Status, SCity ) :-
   good_suppliers( SNum, SName, Status, SCity ),
   good_cities( City ),
   City = SCity.

ssupply( SNum, PNum, Qty ) :-
   london_read_heavy_parts( PNum, PName, Weight ),
   best_suppliers( SNum, SName, Status, City ),
   supply( SNum, PNum, Qty ).
```

A.6. *depth*

```
depth( true, 0 ).
depth( (G1,Gs), Depth ) :- depth( G1, DepthG1 ),
                           depth( Gs, DeptGs ),
                           max( DepthG1, DepthGs, Depth ).
depth( Goal, s(Depth) ) :- prog_clause( Goal, Body ),
                           depth( Body, Depth ).

max( Num, 0, Num ).                    max( 0, Num, Num ).
max( s(X), s(Y), s(Max) ) :- max( X, Y, Max ).

/* prog_clause( Head, Body ) */
prog_clause( member( X, Xs ), append( _, [X|_], Xs ) ).
prog_clause( append( [], L, L ), true ).
prog_clause( append( [X|L1], L2, [X|L3] ), append( L1, L2, L3 ) ).
```

A.7. *grammar*

```
/* selected, representative grammar rules */
expression( Term, Qualifiers ) --> value(Term),
qualification(Qualifiers).

value( Term ) --> identifier( Term ).
value( Term ) --> leftparen, value( Term ), rightparen.
value( Term ) --> . . .
.
.
.
leftparen --> ['('].
rightparen --> [')'].
.
.
.
identifier( Identifier ) --> common_variable( Identifier ).
identifier( Identifier ) --> . . .
.
.
.
common_variable( n ).
common_variable . . .
.
.
.
```