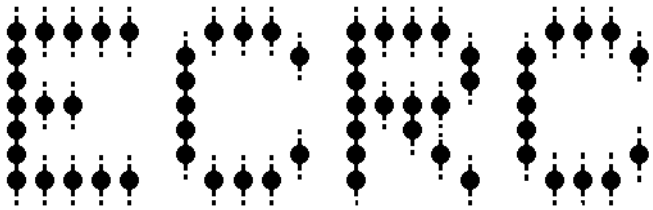
technical report ECRC–92–6

The PADDY Partial Deduction System

Steven Prestwich



ECRC GMBH, ARABELLASTR. 17 D-8000 MÜNCHEN 81, GERMANY - TEL +49 89/926 99 0 - FAX 926 99 170 - TLX 521 6910

For more information about this work, please contact:

Steven Prestwich
ECRC, Arabellastr.17, D-8000 Munich 81, Germany
Tel: +49 89-92699144
email: steven@ecrc.de

Abstract

This report describes the PADDY partial deduction system. Its main features are that it

- can handle full Sepia, including advanced features;
- is automatic;
- always terminates;
- is provably correct on pure Prolog programs;
- uses a powerful adaptive transformation strategy;
- has fast transformation times;
- is based on an unfold/fold system.

Contents

Chapter 1

Introduction

This technical report describes PADDY, a partial deduction system for full Sepia [28] Prolog. There is also a user guide available.

A programmer often has a general purpose program which is to be used for specific applications. For example, a string matching program may be used repeatedly with the same string, a meta-interpreter with the same object program, or an inference engine with the same knowledge base. This can be exploited by deriving versions of the program which run faster for those special cases. Deriving specialised versions is called *program specialisation*. Derivation is done by performing execution steps (resolution or execution of system calls) before run time, by a process called *partial evaluation*. The potential of partial evaluation was independently realised by Ershov [8], Futamura [12] and Turchin [47] in the 1970's. It was first applied to logic programming by Komorowski [20] who gave it the more appropriate name of *Partial Deduction (PD)*. PD performs resolution (or *unfolding*) steps on a logic program in a controlled way prior to execution.

PADDY is a partial deduction [1] system for Sepia Prolog with the following features:

- It can handle full Sepia programs, including advanced features such as modules and delayed goals.
- It is fully automatic, but has a few parameters which may be altered by the user where necessary.
- It is based on an established unfold/fold transformation system which preserves the sequence of answer substitutions, making it provably correct on pure Prolog programs.
- It always terminates.
- It has a powerful adaptive transformation strategy: that is, it alters its behaviour as the transformation proceeds, exploiting useful knowledge gained during the transformation. This makes it potentially more powerful than a system based purely on static analysis.
- It has been designed to have fast transformation times compared to other adaptive strategies. Its transformation time is a linear function of the number of nodes in a partial deduction tree, whereas for many systems it is quadratic.

[1] The transformation we apply is not strictly partial deduction because system calls are handled, and also because the "fold" transformation is allowed. Nevertheless, the aims and results are almost identical to those of PD, and so we shall use this term.

It has been tested successfully on various examples, including several meta-interpreters and PADDY itself.

Chapter 2 describes PADDY's transformation strategy for pure Prolog programs. Chapter 3 describes a general approach to the transformation of full Sepia programs. Chapter 4 combines the two techniques, giving a partial deduction system for full Sepia. Chapter 5 assesses PADDY and compares it with other systems. Appendix A shows some applications of PADDY.

Chapter 2

Transformation strategy for pure Prolog

In this chapter we define a transformation strategy for pure Prolog programs, that is programs consisting only of definite clauses. We describe how to make the transformation terminate in a reasonable time, give correct results and specialise programs satisfactorily.

PD must be guided by a *strategy*, that is a rule specifying the order in which literals are to be selected for unfolding, when to stop unfolding, when to replace atoms by more general ones and so on. The choice of strategy is crucial, as it affects the quality of the transformation (that is, how well it improves programs), how long it takes and whether or not it terminates.

As noted by Jones [17], strategies may be divided into those which are decided before unfolding begins by static analysis (sometimes called binding time analysis [18]) and those which depend upon knowledge derived during the transformation itself, which we shall call *adaptive*. Adaptive strategies are potentially more powerful, because they have access to knowledge which cannot always be predicted by static analysis. However, they tend to be considerably slower.

Many adaptive strategies construct an SLD-tree [2] whose root is a query to be partially deduced. As the tree is constructed, each new atom is compared with its ancestors to detect infinite subtrees. This method has been used successfully in many systems (for example [1, 2, 11, 25, 32, 36, 41]) and gives good results. The price paid is the time spent scanning ancestors at each node of the tree, which increases as the tree gets deeper. This makes these systems impractical for programs which have large search spaces.

An alternative adaptive technique is to store atoms in an unstructured set of definitions used for folding purposes, instead of a tree (for example [9]), but this does not solve the problem: as the transformation proceeds the set of definitions grows, and so the search for appropriate definitions takes longer as the set grows.

The common factor in these strategies is that a growing set of data is derived during the transformation, and must be scanned at each atom. In the first case the transformation time will be quadratic in the depth of the tree, and in the second case quadratic in the number of definitions made.

We propose that partial deduction strategies be designed so that they can adaptively exploit the derived data, but so that the analysis time spent on each atom is independent of the size of the

[2] As SLD-trees play no part in this paper we will not define them here. For a formal definition see [26].

data set. Hence the transformation time will be *linear* in the number of atoms analysed, which is a significant saving. The main problem in designing such a strategy is detecting infinite loops without using the ancestor relationship, and without being so conservative that little unfolding is performed. We show that this can be solved, and in fact that such a strategy can be powerful.

Firstly the transformation rules (Section 2.1) and some preliminary definitions are given (Section 2.2), then our strategy is described (Section 2.3). Its termination (Section 2.4) and correctness (Section 2.5) are shown, and it is then refined to make it more powerful (Section 2.6). The speed of the strategy is discussed (Section 5.2) and demonstrated on a simple example. The quality of the strategy is assessed by applying it to several benchmark programs and comparing the results with those of other systems (Section 5.1).

2.1 The transformation rules

PD as defined in [20, 26] uses only one rule called **unfolding**.

- The **unfold** rule for logic programs is simple: select a resolving atom in a clause body and replace the clause by the set of its resolvents.

We choose to introduce two further rules from the program transformation literature called **definition** and **fold**. The unfold/fold method was first introduced by Burstall & Darlington [4] and Manna & Waldinger [27] for functional languages, and by Tamaki & Sato [43] for logic programming. The other two rules are:

- The **definition** rule allows us to add a new clause to the program at any time during the transformation, as long as its head predicate symbol does not already appear anywhere else in the program. We shall call the new clause a *definition*.

- The **fold** rule is a form of inverse unfold. If an atom in a clause body is an instance of a definition body, then the atom can be replaced by the definition head, with corresponding variable bindings and subject to certain safety criteria. These criteria vary from system to system, but typically say that an atom cannot be folded if it is not the result of some previous unfolding. There are also restrictions on variable bindings to force a correct treatment of local variables (appearing in a clause body but not in the head). These technicalities are not relevant to this paper, and we refer the reader interested to any of several papers [15, 19, 33, 40, 43, 44].

A partial deducer normally takes a program consisting of a set of definite clauses plus a query $? - \mathbf{p}$ where $\mathbf{p}$ is an atom with partially-instantiated arguments. We take a slightly different approach and replace the query by a *goal clause* $\mathbf{new}(v) \ :- ; \ \mathbf{p}$ introduced by the definition rule, where v is the set of free variables appearing in $\mathbf{p}$. The input program is then $\mathsf{P} \cup \{\text{goal clause}\}$. The transformation consists of a sequence of rule applications, each one creating an equivalent program.

The fold rule turns out to be very useful for our strategy, which constructs a set of definitions sufficient to fold any atom encountered while unfolding. From now on, in place of the term *unfolding strategy* we will use the more general term *transformation strategy*.

2.2 Preliminary definitions

A *generalisation* of two terms t_1 and t_2 is a term t_g such that $t_g\theta_1 = t_1$, $t_g\theta_2 = t_2$ for some substitutions θ_1, θ_2. The *most specific generalisation* of t_1 and t_2 is a generalisation which is an instance of all other generalisations. It is unique up to variable renaming.

We define a function $\pi_{d,\mathsf{P}}$ from atoms to ground terms which, given a finite program P and a fixed integer d has a finite range. We call the image of an atom under this function its **pattern**. The pattern of a term is the term itself with all variables, subterms whose function symbol does not occur in P and subterms below a fixed depth d (which we shall call the *pattern depth*) replaced by a new constant κ which does not occur in P. For example, given P containing function symbols $\{0, 1, f, g, h, i\}$ and a pattern depth $d = 2$, an atom $f(99, V, g(0, h(i(1))))$ has the variable V, the new symbol 99 and the deep subterm $i(1)$ replaced by κ, giving $f(\kappa, \kappa, g(0, h(\kappa)))$. The pattern function $\pi_{d,\mathsf{P}}$ maps all atoms which can occur during unfolding to a finite set of ground terms — a property which will be important later.

2.3 An adaptive partial deduction strategy

We first describe a basic strategy, which will be refined in Section 2.6. The basic strategy is shown in Figure 2.3.1 in the form of a logic program (but with if-then-else and "for each" added for brevity). It works by applying the unfold rule, making a definition for each new atom at the start of a clause body and folding those atoms, and reunfolding definitions which turn out to be non-recursive. The set of definitions is kept finite by taking msg's at strategic points, so that for each pattern encountered there is a sequence of increasingly general definitions. This ensures that eventually every atom can be folded.

The transformation begins by setting a table D of definitions (used for folding) to the empty state. A pattern depth d is chosen which remains fixed throughout the transformation. Clauses are represented by a pair: the head, and the body which is an ordered list of atoms (the list representation makes the description easier). The main transformation starts by calling **transform** on the goal clause, and the transformation proceeds by applying a sequence of unfold, define and fold steps as described in Section 2.1. The rules are applied as follows.

A: The first **transform** clause applies when everything in the clause has been unfolded, and then a unit clause is added to the output program P_{out}.

B: The second **transform** clause has three cases:

B1: If the clause was not created by the unfold rule then it must be a definition in D, and so the unfold rule is applied. This is to conform with the underlying unfold/fold system (see Section 2.5). **transform** is then applied to each resolvent. If the clause is not in D but *atom* is a new atom (introduced by the folding rule) which is known to be non-recursive, that is it cannot call itself even indirectly, then again the unfold rule is applied and **transform** applied to each resolvent. A predicate cannot be marked non-recursive until it, and all predicates called by it, are completely transformed. We use a cheap, sufficient test for non-recursion, based on a constructed call-graph.

B2: If *atom* is a (possibly recursive) new atom it is not unfolded.

B2a: Instead, an atom *aux* is created for the rest of the atoms in the clause body, and transformation proceeds from the *aux* clause. The arguments of *aux* are the variables occurring in

Input program: $\mathsf{P}_{in} \cup \{\text{goal clause}\}$
Output program: P_{out}
Initialise P_{out} and the table of definitions D to "empty"
Choose pattern depth $d > 0$
Call **transform**($head$, $body$) where goal clause = ($head$, $body$)
(each clause $\mathbf{h} :- \mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n$ is represented by a pair $(h, a_1.a_2 \ldots a_n.nil)$)

A **transform**($head$, NIL) :–
add ($head$, NIL) to P_{out}
B **transform**($head$, $atom.rest$) :–
B1 if $rest = NIL$ and ($head$, $atom.NIL$) was not produced by unfolding
or $atom$ is a non-recursive new atom then
for each resolvent ($head\theta$, $body$) resolving on $atom$
in ($head$, $atom.rest$), **transform**($head\theta$, $body$)
B2 else if $atom$ is a new atom then
B2a create a new atom aux whose arguments are the free
variables of $rest$ which also occur in $head$ or $atom$,
B2b **transform**(aux, $rest$),
B2c if aux has 1 clause then
resolve on aux in ($head$, $fold.aux$) and add the result to P_{out}
else if aux has 0 clauses then
do nothing
else add ($head$, $atom.aux.NIL$) to P_{out}
B3 else
B3a **matching-def**($atom$),
B3b find the most recent (def', $atom'.NIL$) $\in \mathsf{D}$ such
that $\pi_{d,\mathsf{P}_{in}}(atom') = \pi_{d,\mathsf{P}_{in}}(atom)$,
B3c **choose-fold**($atom$, def', $atom'$, $fold$),
B3d if $fold$ is completely transformed and has 1 clause then
resolve on $fold$ in ($head$, $fold.rest$) to get ($head\theta$, $body$),
transform($head\theta$, $body$)
else **transform**($head$, $fold.rest$)

matching-def($atom$) :–
if $\nexists(def', atom'.NIL) \in \mathsf{D}$ such that $\pi_{d,\mathsf{P}_{in}}(atom') = \pi_{d,\mathsf{P}_{in}}(atom)$ then
add a new definition (def, $atom.NIL$) to D,
transform(def, $atom$)

choose-fold($atom$, def', $atom'$, $fold$) :–
if $atom = atom'\theta$ for some substitution θ then
let $fold = def'\theta$
else let $atom_g = msg(atom, atom')$ where $atom = atom_g\theta$,
add a new definition (def_g, $atom_g.NIL$) to D,
transform(def_g, $atom_g$),
let $fold = def_g\theta$

Figure 2.3.1: *The basic transformation strategy*

these atoms (excluding those not occurring in *atom* and *head*) and the predicate symbol is new.

B2b: *aux* is transformed.

B2c: The fold rule is applied to introduce *aux* to the current clause, but *aux* may be unfolded back into the clause if it only has 0 or 1 clause. If it has more than 1 clause it could be unfolded back, but this would shift the choice point to the left of *atom*, violating Prolog's left-to-right computation rule. This would not cause any correctness problems but might increase the amount of backtracking at run time.

B3: Otherwise apply the fold rule to *atom*, and if there is no definition to fold with we create one:

B3a: **matching-def** checks to see if there is an existing definition whose clause body atom has the same pattern as *atom*. If not, then it creates a definition for *atom* itself and recursively transforms it.

B3b: Now there exists at least 1 definition with the required pattern, and the recursive call to **transform** may have created more. We choose the most recent definition (def', $atom'.NIL$) because it will subsume previous definitions with the same pattern (this will become clearer in Section 2.4).

B3c: Now we try to apply the fold rule by calling **choose-fold**. The fact that the definition has the same pattern does not guarantee that the fold rule will apply. If it does apply then we apply it, otherwise we make another definition whose clause body $atom_g$ is the msg of *atom* and $atom'$, and recursively transform it. The fold rule certainly applies with this definition because *atom* is an instance of $atom_g$.

B3d: Now if $fold$ is completely transformed and has only 1 clause, we unfold $fold$ back into the clause. Unfolding new atoms with only 1 clause would not be safe at B1 because there may be a single recursive clause, in which case unfolding would not terminate. However, it is safe here because it is only unfolded once. **transform** proceeds with the (possibly unfolded back) clause.

The strategy is adaptive in the sense that it modifies its behaviour according to the contents of the definition table D, which grows as the transformation proceeds. At points B3a and B3b, D must be searched for a definition with a given pattern. However, we can make this search in constant time if we store definitions using a hash function based on the pattern (since the pattern is a ground term, the hash function is simple). Thus the total analysis time is a linear function of the number of atoms analysed.

2.4 Termination

For the transformation to terminate, the set of generated definitions must (i) be finite, and (ii) have the property that any leftmost atom (*atom* in Figure 2.3.1) occurring during transformation can be folded using at least one definition from the set.

(i) The pattern function $\pi_{d,\mathsf{P}}$ has all atoms in its domain, and its range is a finite set of patterns. For each pattern encountered, a sequence of definitions is made as the transformation proceeds, and **choose-fold** ensures that each definition in the sequence is the

msg ($atom_g$) of the previous definition ($atom'$) and the latest atom ($atom$). Because generalisation only occurs when $atom \neq atom'\theta$ for any substitution θ we know that $atom_g$ is not a variant of $atom$, and therefore that each pattern gives rise to a sequence of *strictly* increasingly general definitions. Such a sequence must be finite. There are a finite number of possible patterns, and therefore a finite number of possible definitions.

(ii) In the limit, each definition sequence with pattern π approaches a definition whose body $atom_g$ is the most general term with the pattern π, that is π itself with the constant κ replaced by variables. Any atom with pattern π can be folded by its limiting definition.

Hence the strategy always terminates.

2.5 Correctness

There are several unfold/fold systems in the literature whose correctness is well-established, with respect to various program semantics. Our strategy is based upon a recent system of Proietti & Pettorossi [33] which is flexible enough for our purposes, and which preserves the *sequence of answer substitutions* of pure Prolog programs. We superimpose our strategy upon this unfold/fold system to achieve a PD-like transformation. Each application of a transformation rule in the strategy is allowed by the underlying system. In particular, the test B1 of Figure 2.3.1 derives from a folding restriction in that system: the fold rule cannot be applied to any clause which is not the product of previous applications of the unfold rule.

In partial deduction as defined in [20, 26] the input is a set of definite clauses plus a query, and only the unfold rule is applied. Lloyd & Shepherdson [26] state two conditions which must be satisfied in order to preserve correctness (success and finite failure sets), called *closedness* and *singularity*. Similar results were given by Cheng, van Emden and Strooper [5]. As noted in the former paper, if there is no specialised query but just a clause to be unfolded then these conditions are automatically satisfied. Unfold/fold transformation systems are analogous to this special case of partial deduction, in the sense that there is no specialised query but only clauses to be unfolded or folded. Any "specialisation" occurs in the application of the definition rule, in particular when adding the goal clause.

2.6 Refinements

The basic strategy can be refined to make it more powerful without affecting its correctness or termination. The strategy referred to after this section is the refined version.

2.6.1 Choosing non-recursive definitions

The choice of matching definition (see B3a, Figure 2.3.1) can be made more flexible than simply choosing the most recent definition. Searching a growing set of definitions for the "best" one would cause the overhead we want to avoid, but we can make a small fixed number of other definitions immediately available. An obvious choice is the definition (def, $atom.NIL$) created in **matching-def**. We have found a useful heuristic to be: if **def** is non-recursive, then choose **def** instead of **def**$'$ (at B3b). This leads to extra unfolding back (at B1) if we ensure that **choose-fold** selects $fold = def$.

2.6.2 Reducing generalisation

The indexing of definitions by pattern is sometimes too crude. Atoms which are quite different yet have the same pattern are subsumed by their msg, losing useful information. In particular, say we have

$$\begin{aligned} atom &= \mathbf{append}(1.2.3.4.5.6.7.8.NIL, a, b) \\ atom' &= \mathbf{append}(1.2.3.4.5.6.7.8.9.NIL, a, b) \end{aligned}$$

(at B3c) which are both finitely computable, and the pattern depth $d = 5$. Then both *atom* and $atom'$ have the same pattern $\mathbf{append}(1.2.3.4.5.\kappa, \kappa, \kappa)$ and may therefore be generalised to an atom $\mathbf{append}(1.2.3.4.5.c, a, b)$, which is not finitely computable. A solution is not to take the msg of *atom* and $atom'$ if the term size of *atom* is strictly less than that of $atom'$. Instead we can simply use *atom* as the new definition body. Termination is not affected by this refinement because any sequence of atoms which either (strictly) increases in generality or (strictly) decreases in term size must terminate. We assume that the term size function satisfies

$$a = b\theta \;\rightarrow\; \text{termsize}(a) \geq \text{termsize}(b)$$

The following definition of term size satisfies this requirement:

$$\text{termsize}(t) = \begin{cases} 0 & \text{if } t \text{ is a variable} \\ 1 + \sum_{i=1}^{n} \text{termsize}(a_i) & \text{if } t = f(a_1, \ldots, a_n) \end{cases}$$

2.6.3 System calls and undefined predicates

Calls to executable system predicates (for example $1 < 2$) can be handled by generalising B1 to include the case that *atom* is an executable system call. Instead of resolving with clauses, the execution must be simulated by a metacall (**call**/1). If *atom* is a non-executable system call (for example $1 < v$) or an undefined predicate (a common situation in database applications, for instance) then generalise B2 to include these cases.

2.6.4 The auxiliary definition hash table

Auxiliary predicates are often introduced because the leftmost atom in a clause body is not to be unfolded. Many auxiliary predicates may be introduced, and some may have identical definitions (up to variable renaming). To reduce transformation time and generated code size, auxiliary predicates are stored in a hash table similar to the definition hash table but indexed by a different key. Recall that a definition ($\mathbf{d} \;:-\; \mathbf{a}$) is indexed by $\pi_{d,\mathsf{P}}(\mathbf{a})$. Similarly we index an auxiliary predicate $\mathbf{new} \;:-\; \mathbf{a}_1, \ldots \mathbf{a}_n$ by the term $(\pi_{d,\mathsf{P}}(\mathbf{a}_1), \ldots \pi_{d,\mathsf{P}}(\mathbf{a}_n))$. Instead of creating a new auxiliary predicate for a conjunction, we may then fold with an existing one, if its body is a variant of the conjunction.

2.6.5 Accelerated termination

Another modification to the transformation strategy has been added to accelerate the transformation when it has generated many patterns. When the number of patterns exceeds a threshold value, the term depth d is automatically set to 1 for the rest of the transformation, thus limiting the possible number of patterns and allowing the system to terminate gracefully.

Chapter 3

Unfolding tactics for full Sepia

This chapter is concerned with the tactical problems of applying the unfold rule to full standard Sepia programs. We focus on the correctness of the unfold rule and ignore the strategic aspects of when or where to apply it.

Source-level transformation tools for Prolog often use the *unfold* rule, for example partial deduction systems and some compilers. The unfold rule consists of applying the resolution rule prior to execution, and can improve the efficiency of programs by eliminating resolution steps, propagating variable bindings and discovering dead end computations.

Although this rule is very simple when applied to pure Prolog, it becomes problematic when applied to full Prolog, that is Prolog with the usual control structures: conjunction (**,**), disjunction (**;**), the cut (**!**), negation-as-failure (**not**), if-then-else ($\ldots \rightarrow \ldots ; \ldots$) and once-only calls (**once**). The advanced features of Sepia (delayed atoms, modules and so on) create further problems.

This paper proposes a three phase approach to unfolding full Sepia programs:

1. Program simplification, which avoids the incorrect propagation of the cut during unfolding, and also avoids generating complex control structures. All the usual control structures are reduced to conjunction and the ancestral cut, giving a program with a simple structure which can be unfolded more easily.

2. Unfolding the simplified program, for which an unfold rule is specified which avoids various pitfalls. The unfold rule is a tactical rule which can be applied in various ways. It can form the basis for a transformation strategy, defined by a user.

3. Elimination of a class of redundancies which are commonly introduced by unfolding simplified programs.

The problems associated with unfolding full Sepia are described in Section 3.1. Section 3.2 describes the program simplification method. Section 3.3 describes the unfold rule for simplified programs. Section 3.4 describes how to eliminate some redundancies in unfolded simplified programs. An illustrative example is given in Section 3.6. An example of the application of the method to unfolding interpreters is given in Section 3.7.

The six problems are described in Section 3.1. Section 3.2 describes program simplification. Section 3.3 describes the unfold rule for simplified programs. Section 3.4 describes how to eliminate some redundancies in unfolded simplified programs. An illustrative example is given

in Section 3.6. An example of the application of the method to unfolding interpreters is given in Section 3.7.

Notation: predicate symbols and atoms will be written $\mathbf{x}$, constants X, function symbols and general terms x, a conjunction of atoms (which may be empty) $\mathbf{X}$ and a tuple of terms $\underline{x}$. An expression $\mathbf{a}[x/y]$ will denote an atom $\mathbf{a}$ with each occurrence of the term x replaced by the term y.

3.1 Problems with unfolding full Sepia

In this section we describe some of the problems encountered when designing a transformer for full Sepia, providing motivation for the subsequent sections. Most of the problems have been mentioned in other papers, for example [3, 23, 25, 31, 49, 50].

3.1.1 Generating complex control structures

Full Sepia has several control structures which make unfolding problematic. Complex expressions may be generated by unfolding, and to handle these correctly a transformer must know many rules such as:

- disjunction and if-then-else are transparent to cut, that is the effects of a cut reach outside these control structures;
- negation and metacalls (**call**, **once**) are not transparent to cut, that is the effects of the cut are local;
- the disjunction operator '**;**' is associative except when it appears in if-then-else;
- negated atoms cannot be executed until they are ground;
- variable bindings cannot be propagated from one disjunct to another, for example $x = 0$ cannot be unfolded in the disjunction $(x = 0;\ \mathbf{a}(x))$.

Correctly implementing all these rules is an error-prone task. Furthermore, it is possible that not all cases will be considered, so that opportunities for unfolding are missed.

3.1.2 Propagating cut incorrectly

Consider the program

$$\begin{array}{lll} \mathbf{p}(x) & :- \mathbf{q}(x). & \mathbf{q}(0) \quad :- !. \\ \mathbf{p}(2). & & \mathbf{q}(1). \end{array}$$

Unfolding $\mathbf{q}$ naively we get

$$\begin{array}{l} \mathbf{p}(0) \ :- !. \\ \mathbf{p}(1). \\ \mathbf{p}(2). \end{array}$$

which is not equivalent. Given a query $? - \mathbf{p}(x)$ the original program returns $(x = 0;\ x = 2)$ whereas the unfolded version returns only $x = 0$. The incorrectness arises because the scope of the cut has been propagated from $\mathbf{q}$ to $\mathbf{p}$. In general, an atom which matches a clause containing

a cut cannot be unfolded. This is serious because a program may be written with many cuts, in which case very little unfolding can be done.

There are cases where atoms which call the cut directly can be unfolded [3, 25, 31, 36] but they are the exception rather than the rule.

3.1.3 Propagating failure incorrectly

In pure Prolog an atom may be unfolded which is not leftmost in a clause body, and which does not match any clause head. This causes the clause to be deleted, pruning the search space of the program. We refer to this as *failure propagation*. However, in full Sepia this is unsafe. Say we have a clause

$$\mathbf{p} \ :- \ \mathbf{write}(HELLO), \ \mathbf{f}.$$

where **f** is an atom matching no clauses. Unfolding **f** deletes the **p** clause altogether, which is incorrect because the side effect of the **write** atom is lost with it. As well as **write**, the cut and all other atoms with side effects have this problem.

3.1.4 Propagating variable bindings incorrectly

Unfolding an atom which is not leftmost in a clause body may cause variable bindings to be propagated backward; that is, variables in preceding atoms (including the clause head) may become bound earlier than they would under the Prolog left-to-right computation rule. This is useful because it may lead to earlier detection of failure but it can be incorrect. Say we have a program

$$\mathbf{p}(x) \ :- \ \mathbf{var}(x), \ \mathbf{a}(x). \qquad \mathbf{a}(0).$$

A query $? - \mathbf{p}(v)$ succeeds with the answer $v = 0$. Unfolding **a** gives

$$\mathbf{p}(0) \ :- \ \mathbf{var}(0).$$

The same query now fails, and so the unfolding was incorrect. Even with ground atoms backward binding propagation can be incorrect. Consider

$$\mathbf{p}(x) \ :- \ \mathbf{write}(HELLO), \ \mathbf{a}(x). \qquad \mathbf{a}(0).$$

A goal $? - \mathbf{p}(1)$ instantiates $\mathbf{a}(x)$ to $\mathbf{a}(1)$, which causes the same problem as **f** in Section 3.1.3.

3.1.5 Changing the order of solutions

The order of solutions is important in full Sepia, because changing the order may make some solutions unreachable even if there are only a finite number of solutions. For example if we have a clause

$$\mathbf{goal1}(x) \ :- \ \mathbf{goal}(x), \ !.$$

then the *order* of the solutions $(x = t_1; \ x = t_2; \ \ldots)$ for **goal** determines the *set* of solutions $\{x = t_1\}$ for **goal1**. In general we must preserve the order of solutions, but simply taking care not to change the order of clauses when unfolding is not sufficient. Consider the program

$$\mathbf{p}(x) \ :- \ \mathbf{q}(x), \ \mathbf{r}. \qquad \mathbf{q}(0). \qquad \mathbf{r}.$$
$$\mathbf{q}(1). \qquad \mathbf{r}.$$

Calling $? - \mathbf{p}(v)$ gives $(v = 0;\ v = 0;\ v = 1;\ v = 1)$. Unfolding $\mathbf{r}$:

$$\begin{array}{ll} \mathbf{p}(x) & :- \mathbf{q}(x). \qquad \mathbf{q}(0). \\ \mathbf{p}(x) & :- \mathbf{q}(x). \qquad \mathbf{q}(1). \end{array}$$

now calling $? - \mathbf{p}(v)$ gives $(v = 0;\ v = 1;\ v = 0;\ v = 1)$. The order has changed because $\mathbf{r}$ was not the leftmost atom in the clause and the choice point of $\mathbf{r}$ was moved to the left, violating Sepia's left-to-right computation rule. It is not safe in general to unfold a non-leftmost atom in a goal which matches more than one clause head.

3.1.6 Redundant unfolding

A common programming technique is to use *grue* cuts [30] to avoid computations which will fail anyway. If the effects of grue cuts are ignored then the search space covered by a program transformer will be larger than necessary, and a transformer will waste time exploring branches of this space which would never be reached during run time.

Another common technique is to use *red* cuts [30] to throw away solutions. If the effects of red cuts are ignored then this may affect the termination of the unfolding process, because a red cut may be used to select a solution from an infinite set of solutions.

Even if a program transformer is guaranteed to terminate, ignoring the effect of a cut may affect the quality of the unfolded program. Suppose the transformer ensures termination by taking a generalisation of certain visited atoms (a common technique in partial deduction systems). Ignoring a cut may cause more atoms to be visited, making the generalisation more general than necessary and possibly losing important variable bindings. This effect has been observed when specialising meta-interpreters to object programs by partial deduction. An example is described in Section 3.7, and similar problems are mentioned in [31].

3.2 Simplifying Sepia programs

To avoid the problem of Section 3.1.2, namely the restriction on unfolding atom calling cuts, Venken [49] proposed annotating cuts where necessary during unfolding to make their scope explicit. The annotated cut is sometimes called the *ancestral cut* [30], and it is expressed by two predicates: $\mathbf{mark}(v)$ succeeds on being called, binds v to a unique value, and fails on backtracking; $!(v)$ succeeds on being called and removes all choice points back to $\mathbf{mark}(v)$.

We use ancestral cuts to avoid the unfolding restriction, but by transforming all cuts into ancestral cuts *before* unfolding begins. At the same time, we eliminate all other control constructs except conjunction. This is done as follows:

Negation: replace $(\mathbf{not}\ \mathbf{a})$ by $(\mathbf{a} \rightarrow \mathbf{fail};\ \mathbf{true})$.

Once-only calls: replace $\mathbf{once}(\mathbf{a})$ by $(\mathbf{a} \rightarrow \mathbf{true})$.

If-then-else: replace $(\mathbf{a} \rightarrow \mathbf{b})$ by $(\mathbf{mark}(v),\ \mathbf{a},\ !(v),\ \mathbf{b})$ and $(\mathbf{a}_1 \rightarrow \mathbf{b}_1;\ \mathbf{a}_2 \rightarrow \mathbf{b}_2;\ \ldots;\ \mathbf{a}_n)$ by

$$\mathbf{mark}(v),\ (\mathbf{a}_1,\ !(v),\ \mathbf{b}_1;\ \mathbf{a}_2,\ !(v),\ \mathbf{b}_2;\ \ldots;\ \mathbf{a}_n)$$

where v is a new variable.

Cut: for each predicate **p** which has a clause containing a cut, replace every atom $\mathbf{p}(\underline{x})$ by $\mathbf{p}(\underline{x}, v)$ where v is a new variable, precede each atom $\mathbf{p}(\underline{x}, v)$ in a clause body by $\mathbf{mark}(v)$, and in each **p** clause replace each cut by $!(v)$ where v is the new argument in the clause head.

Disjunction: replace

$$\mathbf{p} \ :- \mathbf{L},\ (\mathbf{a}_1;\ \ldots;\ \mathbf{a}_n),\ \mathbf{R}$$

by

$$\begin{cases} \mathbf{p} \ :- \mathbf{L},\ \mathbf{new}(\underline{x}),\ \mathbf{R}. \\ \\ \mathbf{new}(\underline{x}) \ :- \mathbf{a}_i. \quad (i = 1 \ldots n) \end{cases}$$

where **new** is a new predicate symbol created for each disjunction (which we shall refer to as an *auxiliary predicate*), and $\underline{x}$ is the set of variables occurring in both the disjunction and **p**, **L** or **R**.

After applying these transformations each clause body consists only of conjunctions of atoms. From a transformation point of view, simplified programs are written in pure Prolog plus some predicates with special properties. This avoids the problem of generating complex control expressions described in Section 3.1.1.

3.3 Unfolding simplified programs

Now we specify an unfold rule for simplified programs which avoids the problems described in Sections 3.1.3, 3.1.4, 3.1.5 and 3.1.6, namely the useless unfolding of redundant branches, the incorrect propagation of variable bindings and failure, and changing the order of solutions.

3.3.1 A predicate classification

Before describing the unfold rule, we make a classification of predicates. A predicate **p** is classed as either:

- **binding sensitive** if $(\mathbf{a},\ v_1 = v_2) \ \neq \ (v_1 = v_2,\ \mathbf{a})$ for some atom **a** with predicate symbol **p** (where the v_i are variables and $\neq$ denotes different operational semantics). Examples of binding sensitive predicates are **var** and $==$.

- **failure sensitive** if $(\mathbf{a},\ \mathbf{fail}) \ \neq \ \mathbf{fail}$ for some atom **a** with predicate symbol **p**. The cut and all predicates with side effects are failure sensitive. All failure sensitive predicates are also binding sensitive, because any atom $v_1 = v_2$ may become equivalent to **fail** during unfolding, if v_1 and v_2 become bound to non-unifiable terms.

- **pure** if it is neither binding nor failure sensitive.

This classification is similar to that of [36]. The system predicates of Sepia must be classified by hand when designing a program transformer. The classification of all other predicates can be deduced by the rule:

any atom matching a clause head whose clause body contains a binding [failure] sensitive atom is itself classed as binding [failure] sensitive, otherwise it is classed as pure.

When in doubt (for example when handling atoms whose clauses are unknown) it is always safe to classify an atom as failure sensitive. A predicate such as **loop** defined by (**loop** :– **loop**) should strictly be classed as failure sensitive, but we shall class it as pure because non-productive infinite loops are rarely used.

The !/1 predicate is failure sensitive. The **mark**/1 predicate is not failure sensitive because (**mark**(v), **fail**) is always equivalent to (**fail**, **mark**(v)): it sets up a mark for subsequent computation, and **fail** has no subsequent computation. Nor is **mark**/1 binding sensitive because (**mark**(v), $v_1 = v_2$) is always equivalent to ($v_1 = v_2$, **mark**(v)): either $v_1 = v_2$ succeeds in which case it is equivalent to **true**, or it fails, and **mark**/1 is not failure sensitive.

3.3.2 The unfold rule

The unfold rule given in this section avoids all the problems mentioned in Section 3.1 except one, which is partially solved in the next section. It begins with a clause

$$\mathrm{C} : \mathbf{p} \ :- \ \mathbf{a}_1, \ \ldots, \ \mathbf{a}_N$$

Unfolding a chosen atom $\mathbf{a}_i$ is done as follows:

1. If $i = 1$ then replace C by the set of resolvents using all matching clauses for $\mathbf{a}_1$.
2. If $i > 1$ then:
   a. If $\mathbf{a}_i$ matches no clauses then C is replaced by
   $$\mathbf{p} \ :- \ \mathbf{a}_1, \ \ldots, \ \mathbf{a}_{i-1}, \ \mathbf{fail}$$
   and **fail** is propagated back through $\mathbf{a}_1 \ \ldots \ \mathbf{a}_{i-1}$.
   b. If $\mathbf{a}_i$ matches one clause ($\mathbf{h} \ :- \ \mathbf{T}$) then C is replaced by
   $$\mathrm{C}' : \mathbf{p} \ :- \ \mathbf{a}_1, \ \ldots, \ \mathbf{a}_{i-1}, \ (\mathbf{a}_i = \mathbf{h}), \ \mathbf{T}, \ \mathbf{a}_{i+1}, \ \ldots, \ \mathbf{a}_N$$
   and $(\mathbf{a}_i = \mathbf{h})$ is propagated through C′.
   c. If $\mathbf{a}_i$ matches more than one clause then make a new auxiliary definition
   $$\mathrm{D} : \mathbf{new} \ :- \ \mathbf{a}_i, \ \ldots, \ \mathbf{a}_N$$
   where the arguments of **new** are the free variables of $\mathbf{a}_{i+1}, \ \ldots, \ \mathbf{a}_N$ which also occur in $\mathbf{p}$, $\mathbf{a}_1$. Fold C using D giving
   $$\mathrm{C}' : \mathbf{p} \ :- \ \mathbf{a}_1, \ \ldots \ \mathbf{a}_{i-1}, \ \mathbf{new}$$
   C is replaced by C′ and unfolding is applied to $\mathbf{a}_i$ in D.

Some notes on the unfold rule follow.

Failure propagation

This is done in (a) by applying the rule:

replace (**a**, **fail**) *by* **fail** *if* **a** *is not failure sensitive*

repeatedly until either a failure sensitive $\mathbf{a}_j$ ($j < i$) is encountered, or until the clause (**p** :– **fail**) is reached (which can be deleted in most Prologs).

Variable binding propagation

This is done in (b) as follows. First split the atom $\mathbf{a}_i = \mathbf{h}$ into several atoms of the form $v = t$ where v is a variable and t is a term (this is always possible). Then propagate each $v = t$ through $\mathbf{T}$, $\mathbf{a}_{i+1}$ ... $\mathbf{a}_n$ to give $\mathbf{T}[v/t]$, $\mathbf{a}_{i+1}[v/t]$, ..., $\mathbf{a}_n[v/t]$. Then move each $v = t$ in turn as far to the left as possible by repeatedly applying the rule:

replace **p** :– **X**, **a**, $v = t$, **Y**.
by **p** :– **X**, $v = t$, **a**$[v/t]$, **Y**.
if either

(1) **a** *is binding or failure sensitive and*
(i) v *does not occur in* **p**, **X** *or* **a**
(ii) or t *is a variable which does not occur in* **p**, **X** *or* **a**
(2) or **a** *is pure.*

The condition that v (or t) is a variable which does not occur anywhere to the left of $v = t$ ensures that $v = t$ always succeeds so that it can be safely swapped with failure sensitive **a**. It also ensures that v (or t) cannot occur in **a** even by aliasing so that $v = t$ can be safely swapped with binding sensitive **a**. The condition is sufficient to ensure safety, but could be weakened by abstract interpretation to detect aliasing and freeness of variables. This is outside the scope of this paper.

System calls

If $\mathbf{a}_i$ is a system call then instead of using clauses the program transformer must simulate its execution. System calls cannot always be unfolded: those with side effects, and some without sufficient bindings on their arguments. For example, $a < b$ can only be finitely unfolded if a and b are both bound to numbers. A program transformer must therefore know when system predicates can be unfolded and how to unfold them.

The effects of folding

The introduction of the auxiliary predicate by folding in (c) negates any direct advantage gained by the unfolding of $\mathbf{a}_i$. However, it propagates all the variable bindings gained from unfolding $\mathbf{a}_i$ to the atoms $\mathbf{a}_{i+1} \ldots \mathbf{a}_N$, which is likely to lead to directly useful unfolding later.

3.3.3 Executing ancestral cuts

We have not yet addressed the problem in Section 3.1.6: that a transformer may explore branches of a program which will never be reached during execution, by ignoring the effects of the cut. If enough is known at unfolding time about the status of predicate arguments, then some cuts can be executed to prune the program as it is being unfolded. This idea is used in [3, 31, 36]. We shall adapt the standard cut execution rule to the ancestral cut as follows.

Say we have chosen $\mathbf{a}_i = \mathbf{p}(\underline{x}, v)$ for unfolding, and the clauses for $\mathbf{p}$ are

$$\mathrm{C}_j: \ \mathbf{p}(\underline{x}_j, v_j) \ :- \mathbf{X}_j. \qquad (j = 1 \ldots k)$$

where $\mathbf{X}_i = (!(v_i),\ \mathbf{R})$ for some i. If the unifier of $\mathbf{p}(\underline{x}, v)$ and $\mathbf{p}(\underline{x}_i, v_i)$ does not bind any variables of $\mathbf{p}(\underline{x}, v)$ then only clauses $\mathrm{C}_1 \ldots \mathrm{C}_i$ need be used for unfolding, and clauses $\mathrm{C}_{i+1} \ldots \mathrm{C}_k$ can be discarded. In fact by analogy with standard Prolog, if we have a mode analysis of the program then we only need to ensure that the unifier does not bind any *input variables* of $\mathbf{p}(\underline{x}, v)$. For further details on this idea see [3, 31, 36].

In the standard cut execution rule the executed cut can be removed, but this is not generally possible with ancestral cuts. For example, say we have a program

$$\mathbf{p}(x) \ :- \mathbf{mark}(v),\ \mathbf{q}(x),\ \mathbf{r}(x, v).$$

$$\begin{array}{ll} \mathbf{r}(x, v) \ :- \ !(v). & \mathbf{q}(0). \\ \mathbf{r}(2, v). & \mathbf{q}(1). \end{array}$$

A query $? - \mathbf{p}(v)$ would give $v = 0$. If we prune $\mathbf{r}$ using $!(v)$ to give

$$\mathbf{r}(x, v) \ :- \ !(v).$$

the answer is the same. But if we also delete $!(v)$ to give

$$\mathbf{r}(x, v).$$

then there are two answers ($v = 0$; $v = 1$) because the cut did not refer to its parent predicate $\mathbf{r}$. Hence it is incorrect to remove $!/1$ in general. In Section 3.4.2 we show that it is possible under certain circumstances.

3.4 Post-unfolding optimisations

Mechanically generated programs often contain redundancies of various kinds, and it is usually profitable to apply some simple tidying up rules after unfolding. For example [14] provides rules for the removal of redundant function symbols, [34] describe how to eliminate certain variable arguments, and some general optimisation techniques are given in [37].

Unfolding simplified programs creates a special class of redundancies associated with the ancestral cut predicates, and we now describe some new ways of eliminating these redundancies. These are intended to be applied automatically after unfolding.

3.4.1 Reintroduction of standard cuts

The standard cut can be implemented more efficiently than ancestral cuts, and so it is beneficial to replace ancestral cuts by standard cuts after the unfolding where possible. This can be done as follows. First we make a definition:

replace	$\mathbf{p} :- \mathbf{L}, \mathbf{mark}(v), \mathbf{mark}(v'), \mathbf{R}.$
by	$\mathbf{p} :- \mathbf{L}, \mathbf{mark}(v), \mathbf{R}[v'/v].$
replace	$\mathbf{p} :- \mathbf{L}, \mathbf{mark}(v), !(v), \mathbf{R}.$
by	$\mathbf{p} :- \mathbf{L}, \mathbf{mark}(v), \mathbf{R}.$
replace	$\mathbf{p} :- \mathbf{L}, !(v), !(v'), \mathbf{R}.$
by	$\mathbf{p} :- \mathbf{L}, !(v'), \mathbf{R}[v/v'].$
replace	$\mathbf{p} :- \mathbf{L}, !, !(v), \mathbf{R}.$
by	$\mathbf{p} :- \mathbf{L}, !(v), \mathbf{R}.$
replace	$\mathbf{p} :- \mathbf{L}, !(v), !, \mathbf{R}.$
by	$\mathbf{p} :- \mathbf{L}, !, \mathbf{R}[v/v']).$
replace	$\mathbf{p} :- \mathbf{L}, !(v), \mathbf{mark}(v'), \mathbf{R}.$
by	$\mathbf{p} :- \mathbf{L}, !(v), \mathbf{R}[v'/v].$
replace	$\mathbf{p} :- \mathbf{L}, \mathbf{mark}(v), \mathbf{R}.$
by	$\mathbf{p} :- \mathbf{L}, \mathbf{R}.$ (if v does not occur in $\mathbf{R}$)

Figure 3.4.1: *Eliminating redundant cut predicates*

Definition 1 (local cut argument) *Argument i of a predicate $\mathbf{p}/n$ $(1 \leq i \leq n)$ is a* local cut argument *of $\mathbf{p}/n$ if:*

- *it is a variable in every $\mathbf{p}/n$ atom in the program,*
- *every $\mathbf{p}/n$ atom is immediately preceded by an atom $\mathbf{mark}(v)$ where v is the i^{th} argument of the atom,*
- *in every $\mathbf{p}/n$ clause, every occurrence of the i^{th} argument v of the head occurs only in atoms of the form $!(v)$.*

Now if we have a predicate $\mathbf{p}(a_1, \ldots, a_n)$ where a_i is a local cut argument of $\mathbf{p}/n$ then we can replace all corresponding atoms $!(v)$ by $!$ in the clauses for $\mathbf{p}/n$. Moreover the atoms $\mathbf{mark}(a_i)$ preceding each $\mathbf{p}/n$ atom can be deleted, and the i^{th} argument can be dropped from every $\mathbf{p}/n$ atom in the program. This is the inverse of the transformation in Section 3.2 which replaced standard cuts by ancestral cuts.

3.4.2 Removal of ancestral and standard cuts

Figure 3.4.1 shows some rules for removing both standard and ancestral cut atoms. The last rule can be applied much more often if redundant arguments are first removed from predicates. A simple rule which is sufficient to detect unused local cut arguments is:

> *an argument of a predicate is redundant if in each clause head for the predicate it is a variable and does not appear in any other argument, nor in any atoms in the clause body.*

The next to last rule is a generalisation of a rule in [7] which optimises contiguous "functional" atoms, and is related to intelligent backtracking strategies.

In Section 3.3.3 it is noted that an "executed" $!(v)$ cannot in general be deleted. However, if v is a local cut argument then $!(v)$ is replaced by ! using the cut reintroduction rule of Section 3.4.1. We can then apply the well known rule:

delete any cut at the start of the last clause for a predicate.

3.5 Handling advanced Sepia features

Apart from the extra complexity imposed by the advanced features, they each cause special problems when unfolding. A conservative but simple solution is to treat all such atoms as non-unfoldable system calls. We now discuss each feature.

3.5.1 Modules

The Sepia module system is a partitioning of program components, and between modules is a relation called *visibility*. If module A is invisible to module B then the predicates of A cannot be seen by B. This affects unfolding, because an invisible predicate cannot be unfolded. Where visibility can be predicted statically, the unfold rule can be modified to distinguish between visible and invisible predicates. However, if visibility can change dynamically then unfolding between modules is unsafe. For complete safety, no predicate is allowed to be unfolded which is defined inside any module other than the top level (`sepia`) module.

3.5.2 Dynamic predicates

In Sepia most predicates are classed as *static*, which means that they cannot be asserted nor retracted. This enables them to be compiled more efficiently. Predicates which are to be asserted or retracted must be declared as *dynamic*. The unfolding of dynamic predicates is unsafe because their clauses at transformation time may be different to those at runtime. No atom whose predicate symbol is classed as dynamic is unfolded.

Dynamic atoms must be classed as failure sensitive, because any dynamic predicate **p** may have a clause such as

$$\mathbf{p} \ :- \ \mathbf{write}(HELLO).$$

added at runtime, causing it to have side effects.

3.5.3 Delay declarations

Sepia predicates may have delay declarations, that is declarations stating when they should not be selected for resolution. These can only be safely unfolded when it is certain that these conditions are satisfied, and so no atom with a delay declaration is unfolded.

Delayed atoms also affect the correctness of failure and binding propagation. We assume here that all predicates with delay declarations are pure, and that atoms can never be delayed across impure atoms. In a future paper delayed atoms will be treated in more detail.

3.5.4 Meta-level predicates

There are several predicates which call others, and are treated as non-unfoldable and failure sensitive: all solution predicates (**setof**, **findall** etc.), sound negation calls ($\tilde{}/1$, although $\tilde{} = /2$ is unfolded when it is safe to do so), constructive negation calls (**neg** and **if-then-else**) and catch-and-throw calls (**block**, **exit_block**). The **call**/1 predicate is only unfolded if its argument is a non-variable unfoldable atom, in which case it inherits the purity or impurity of the atom. If its argument is a variable, it is treated as non-unfoldable and failure sensitive.

3.6 A simple example

To illustrate our program simplification, unfold rule and optimisation rules, consider the program:

$$\begin{array}{l} \mathbf{p}(x) \ :- \ \mathbf{q}(x),\ \mathbf{r}(x). \\ \mathbf{p}(2). \end{array}$$

$$\begin{array}{ll} \mathbf{q}(1) \ :- \ !. & \mathbf{r}(1) \ :- \ !. \\ \mathbf{q}(3). & \mathbf{r}(4). \end{array}$$

We would like to perform some unfolding on this program to make **p** more efficient, but neither **q** nor **r** can be unfolded in the **p** clause because they both call a cut. Nor is a cut execution rule applicable. Hence no improvement by unfolding is possible by the program in this form.

We now apply our approach. The simplified program is:

$$\begin{array}{l} \mathbf{p}(x) \ :- \ \mathbf{mark}(v_1),\ \mathbf{q}(x, v_1),\ \mathbf{mark}(v_2),\ \mathbf{r}(x, v_2). \\ \mathbf{p}(2). \end{array}$$

$$\begin{array}{ll} \mathbf{q}(1, v_1) \ :- \ !(v_1). & \mathbf{r}(1, v_2) \ :- \ !(v_2). \\ \mathbf{q}(3, v_1). & \mathbf{r}(4, v_2). \end{array}$$

To unfold **q**, create an auxiliary predicate:

$$\begin{array}{l} \mathbf{p}(x) \ :- \ \mathbf{mark}(v_1),\ \mathbf{new}(x, v_1). \\ \mathbf{p}(2). \end{array}$$

$$\mathbf{new}(x, v_1) \ :- \ \mathbf{q}(x, v_1),\ \mathbf{mark}(v_2),\ \mathbf{r}(x, v_2).$$

q can now be unfolded in the **new** clause:

$$\begin{array}{l} \mathbf{new}(1, v_1) \ :- \ !(v_1),\ \mathbf{mark}(v_2),\ \mathbf{r}(1, v_2). \\ \mathbf{new}(3, v_1) \ :- \ \mathbf{mark}(v_2),\ \mathbf{r}(3, v_2). \end{array}$$

r can be unfolded in both these clauses. The first clause becomes

$$\mathbf{new}(1, v_1) \ :- \ !(v_1),\ \mathbf{mark}(v_2),\ !(v_2).$$

and the second clause is deleted via failure propagation. Now **new** can be unfolded in the first **p** clause:

$$\begin{array}{l} \mathbf{p}(1) \ :- \ \mathbf{mark}(v_1),\ !(v_1),\ \mathbf{mark}(v_2),\ !(v_2). \\ \mathbf{p}(2). \end{array}$$

Finally, the conjunction ($\mathbf{mark}(v_1),\ !(v_1),\ \mathbf{mark}(v_2),\ !(v_2)$) has no effect and can be deleted using the rules of Section 3.4:

$$\begin{array}{l} \mathbf{p}(1). \\ \mathbf{p}(2). \end{array}$$

3.7 Application to meta-interpreters

The example of Section 3.6 does not illustrate the ancestral cut execution rule, which has been found particularly useful in meta-interpreter specialisation. A common interpretation strategy is to split up a conjunction of atoms, process these separately, join the results and continue:

$$\begin{array}{l}\vdots\\ \mathbf{int}((a,b)) \ :- \ \mathbf{p}(a,a'),\ \mathbf{p}(b,b'),\ \mathbf{join}(a',b',c),\ \mathbf{int}(c).\\ \vdots\\ \\ \mathbf{join}(TRUE,b,b) \ :- \ !.\\ \mathbf{join}(a,TRUE,a) \ :- \ !.\\ \mathbf{join}(a,b,(a,b)).\end{array}$$

Now consider the case where the two **p** atoms are unfolded and bind a, b, a' and b' to $TRUE$. The **int** clause then becomes

$$\mathbf{int}((TRUE,TRUE)) \ :- \ \mathbf{join}(TRUE,TRUE,c),\ \mathbf{int}(c).$$

The **join** of $TRUE$ and $TRUE$ is $TRUE$, and the last atom in the clause will become $\mathbf{int}(TRUE)$. But if the pruning effect of the cut is ignored then **join** gives this answer twice, plus an extra answer $\mathbf{int}((TRUE,TRUE))$. These cases are never reached during execution, and unfolding time is wasted. Worse, in some interpreters the effects may propagate to give atoms

$$\begin{array}{l}\mathbf{int}(TRUE)\\ \mathbf{int}((TRUE,TRUE))\\ \mathbf{int}((TRUE,TRUE,TRUE))\ldots\end{array}$$

This either prevents the transformer from terminating, or forces it to replace all these atoms by the more general atom $\mathbf{int}(v)$, sacrificing specialisation.

A cut execution rule can avoid this. If we simplify the program then after unfolding we have

$$\mathbf{int}((TRUE,TRUE)) \ :- \ \mathbf{mark}(v),\ \mathbf{join}(TRUE,TRUE,c,v),\ \mathbf{int}(c).$$

where

$$\begin{array}{l}\mathbf{join}(TRUE,b,b,v) \ :- \ !(v).\\ \mathbf{join}(a,TRUE,a,v) \ :- \ !(v).\\ \mathbf{join}(a,b,(a,b),v).\end{array}$$

Now to unfold **join** we create an auxiliary predicate

$$\mathbf{int}((TRUE,TRUE)) \ :- \ \mathbf{mark}(v),\ \mathbf{new}(v).$$

$$\mathbf{new}(v) \ :- \ \mathbf{join}(TRUE,TRUE,c,v),\ \mathbf{int}(c).$$

then unfold **join**:

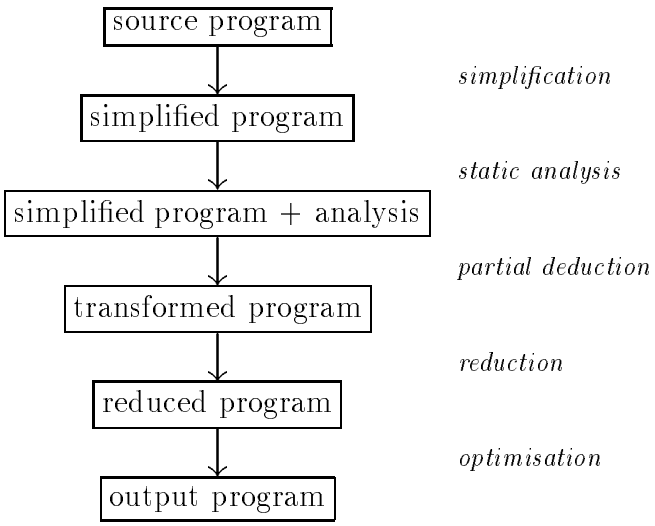
$$\begin{array}{l}\mathbf{new}(v) \ :- \ !(v),\ \mathbf{int}(TRUE).\\ \mathbf{new}(v) \ :- \ !(v),\ \mathbf{int}(TRUE).\\ \mathbf{new}(v) \ :- \ \mathbf{int}((TRUE,TRUE)).\end{array}$$

Now the ancestral cut execution rule applies and the second and third **new** clauses can be pruned, avoiding redundant clauses.

Chapter 4

Transformation strategy for full Sepia

In this chapter we superimpose the transformation strategy for pure Prolog (Chapter 2) onto the unfolding tactics for full Sepia (Chapter 3) to obtain a transformation strategy for full Sepia. We also add a simple static analysis before transformation, and some extra optimisations after transformation. The complete transformation system consists of 5 phases, which are described in this chapter:



4.1 Phase 1: simplification

This is the simplification described in Section 3.2. The advanced Sepia features are left unchanged in the program, but control structures **not**/1, **once**, ! and $\rightarrow$ and ';' are removed, leaving only ',' and the ancestral cut. This makes transformation simpler and more powerful by avoiding various unfolding restrictions.

4.2 Phase 2: static analysis

The static analysis classifies predicates using the scheme described in Section 3.3.1. PADDY has the failure and binding sensitivities of all system calls built in. The classification of all other (user defined) predicates can be deduced by the rule: *any atom matching a clause head whose clause body contains a binding [failure] sensitive atom is itself classed as binding [failure] sensitive.* This rule is applied repeatedly until no new binding [failure] sensitive predicates are found. When in doubt about an atom it is always safe to classify an atom as failure sensitive (and therefore also binding sensitive). This applies to atoms such as **call**(v) because v may become bound to a binding [failure] sensitive atoms, also to atoms whose predicates are undefined. A predicate such as **loop** defined by (**loop** : − **loop**) should strictly be classed as failure sensitive, but we shall class it as pure because non-productive infinite loops are rarely used.

4.3 Phase 3: partial deduction

This is the main part of the transformation. The refined transformation strategy of Chapter 2 is superimposed upon the unfold rule of Chapter 3.

The transformation strategy already creates auxiliary predicates as required by the unfold rule, though for different reasons. All that remains is to modify the transformation strategy to correctly handle propagation of failure and variable bindings in full Sepia. This is done by modifying the transformation strategy at B2c (Figure 2.3.1, page 6) as follows. Recall that at this point of the transformation, we have a clause

$$(head,\ atom.aux.NIL)$$

where $atom$ has not been unfolded and the auxiliary predicate aux has been completely transformed. Now if aux has exactly one clause $(h,\ t)$ then instead of unfolding it immediately as in the pure Prolog case, we follow the unfold rule for full Sepia to get

$$(head,\ atom.(aux = h).t.NIL)$$

and propagate $aux = h$ back through $atom$. Alternatively, if aux has no clauses then instead of doing nothing we replace aux by $fail$ to get

$$(head,\ atom.fail.NIL)$$

and propagate $fail$ back through $atom$.

4.4 Phase 4: reduction

The output file contains one extra feature: calls to import the ancestral cut predicates, which are not part of standard Sepia. These calls always appear, although they are only necessary if the program contains calls to the ancestral cuts.

4.5 Phase 5: optimisation

This is the optimisation phase described in Section 3.4, which tidies up some of the most common redundancies generated by transforming simplified programs.

Chapter 5

Comparison with other approaches

In this chapter we assess the PADDY system and compare it with others in the literature using several criteria: quality, speed, correctness, termination, handling of full Prolog and some of the techniques used.

5.1 Quality of transformation

It is easy to design fast, correct, terminating partial deduction strategies which make little improvement, but if a system gives trivial improvements it is of small interest because the ability of a transformation strategy to improve programs is its *raison d'être*.

It is difficult to compare the quality of different systems analytically, and so we applied PADDY to the 7 benchmark programs given in the survey paper of Lam & Kusalik [24]. In that paper these programs were used to compare 5 systems: Mult (Levi & Sardu [25]), Constraints (Fujita [9]), Pure (Kursawe [21]), ProMiX† (Lakhotia & Sterling [23]) and Peval† (Takeuchi & Furukawa [42]). [1] To these we added Mixtus (Sahlin [36]) and PADDY. The examples are all pure Prolog, selected for their differing characteristics (degree of determinism, depth of recursion etc.). They are:

- relative: the well-known "relative" program
- match: a simple string matching problem
- contains: a complicated string matching problem
- transpose: a deterministic, deeply recursive matrix transposition problem
- ssupply: a nondeterministic, solution-intensive database problem
- depth: a simple meta interpreter
- grammar: a definite clause grammar translation

Each example was specialised with respect to a query and the search tree sizes of the original and transformed programs were compared using more specific queries. The results are shown in Figure 5.1.1.

[1] Only those names marked † were given by their respective authors.

				automatic systems				
program	original	Peval	ProMiX	Constraints	Pure	Mult	Mixtus	PADDY
relative	112	2	2	88	2	73	4	2
match	59	46	59	56	50	57	52	54
contains	109	9	109	109	30	57	22	21
transpose	69	2	2	63	2	2	2	2
ssupply	25	2	2	19	2	8	2	2
depth	38	2	13	38	2	2	4	2
grammar	160	6	77	17	146	77	6	6

Figure 5.1.1: *Search tree sizes for residuals*

The best results were given by Peval, as might be expected from a non-automatic system. If there is an optimum transformation strategy for a given program, a user might find it where a general purpose automatic system would not. This optimum strategy can be communicated to a non-automatic partial deducer via a set of declarations, which say which literals should or should not be unfolded.

For the automatic strategies, the best result for each example (ignoring small differences in the figures) was given by PADDY and Mixtus together. In most examples at least one other system gave the same result, but no other system did so for all examples. Hence PADDY and Mixtus gave equal best results overall, but PADDY used a less expensive strategy.

5.2 Speed of transformation

Strategies which compare atoms with their ancestors in an SLD-tree give good results but have a large analysis overhead. The time spent searching for the ancestors of each atom in an SLD-tree is proportional to its depth in the tree, giving a total analysis time which is quadratic in the depth of the tree. Examples are the strategies of Benkerimi & Lloyd [1], Bruynooghe, de Schreye and Martens [2], Fuller & Abramsky [11], Levi & Sardu [25], Proietti & Pettorossi [32], Sahlin [36] and Sterling & Beer [41].

The strategy of Fujita [9] does not compare ancestors. Instead it stores each atom in an unstructured set which must be searched for folding purposes. The total time spent comparing atoms in this strategy is therefore quadratic in the number of nodes in the tree.

The strategy of Gallagher [13] also does not compare ancestors. It has a phase called *flow analysis* which collects a set of atoms subsuming all the atoms occurring while unfolding. This set is then used to construct a specialised program. The flow analysis takes the atoms in the query as the initial set and iteratively constructs the final set. At each iteration the atoms in the current set are unfolded, new atoms are added from the resulting clause bodies, and the set is reduced by taking generalisations of similar atoms. Since the set reduction step must scan the current set, this seems to indicate a worst case behaviour which again depends quadratically upon the number of nodes in the SLD-tree.

Non-adaptive strategies use static analysis to plan the transformation in advance, and do not analyse a growing set of data while unfolding. Their transformation times should therefore be linear in the number of nodes in the SLD-tree. Examples are Kursawe's system [21] and the Logimix system of the DIKU group in Copenhagen (not yet available to our knowledge). However, we expect these to be more conservative than ours, because static analysis cannot

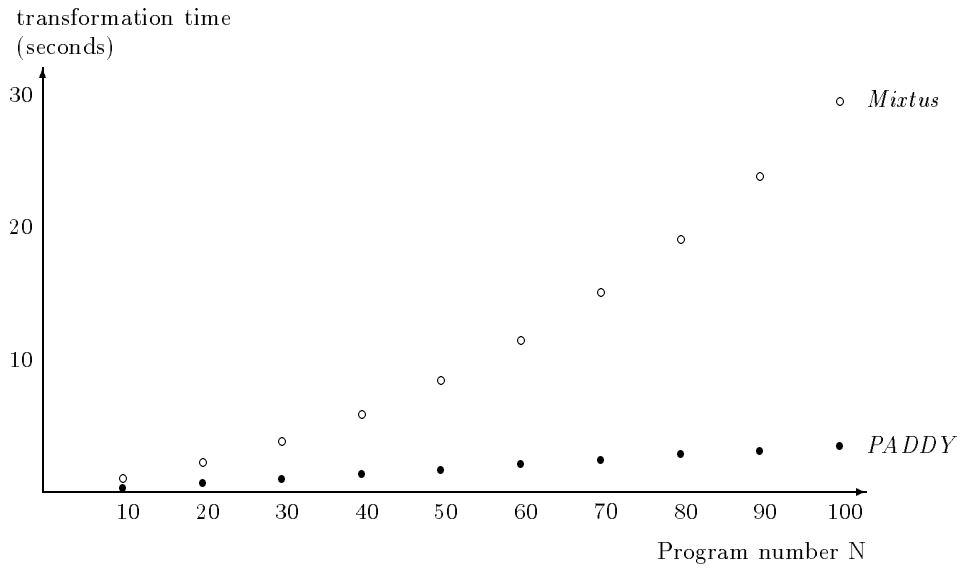


Figure 5.2.1: *Comparison of transformation times*

predict everything which can be discovered during the transformation, as noted by Fuller and Abramsky [11].

Although our strategy analyses a growing set of data while unfolding, the analysis time it spends on each atom is independent of the size of the set. Hence the total analysis time is linear in the number of nodes in the tree. This makes it an order of magnitude faster than other known adaptive strategies. However, we expect it to be somewhat slower than non-adaptive strategies because it may make a definition for an atom, fold it, transform it and then unfold it back if it is non-recursive. A non-adaptive strategy would simply unfold the atom in the first place, assuming it is clever enough to know that it will be non-recursive.

To illustrate our point about transformation times, consider the following family of recursive programs:

$$\text{program } N = \left\{ \begin{array}{lcl} \mathbf{p}_0 & \leftarrow & \mathbf{p}_1 \\ \mathbf{p}_1 & \leftarrow & \mathbf{p}_2 \\ & \vdots & \\ \mathbf{p}_N & \leftarrow & \mathbf{p}_0 \end{array} \right.$$

where $\mathbf{p}_i$ $(i = 0 \ldots N)$ are atoms of arity 0. This example is designed to be a simple test of transformation time as a function of search space size, without irrelevant complications (such as unification overhead of arguments) which should be the same for any PD system.

Say a PD system is unfolding program N. If it uses loop detection based on ancestors in the SLD-tree, then at the i^{th} atom $\mathbf{p}_i$ it must examine $i-1$ ancestors. Thus the number of comparisons will be $\sum_{i=0}^{N-1} i = N(N-1)/2$. Our strategy, on the other hand, only tries to examine at most one other atom at each $\mathbf{p}_i$, and in fact only finds an atom with the same pattern when it reaches the last clause $\mathbf{p}_N \leftarrow \mathbf{p}_0$. Thus the number of comparisons is at most $N-1$.

To demonstrate this, we compared PADDY with Sahlin's Mixtus system [36] on this family of programs. We ran both partial deducers on a Sun 3 workstation under Sepia, and the transformation times are shown in Figure 5.2.1. As can be seen, the transformation times of PADDY are roughly linear in N whereas those of Mixtus are roughly quadratic in N. This is not

a criticism of Mixtus, but a comparison of the complexities of the two transformation methods.

5.3 Treatment of Prolog control structures

We identify three basic approaches for handling control structures when unfolding full Prolog programs.

Firstly, the problems of unfolding full Prolog can be minimised by a variety of techniques. This has the advantage of operating directly on standard Prolog programs, but the disadvantages that unfolding requires complicated rules and is not always applicable. The partial deduction systems of Bugliesi & Russo [3], Levi & Sardu [25], Owen [31] and Sahlin [36] follow this approach. They show that the restriction on unfolding atoms which call cuts can be lifted under certain circumstances; also that even when an atom cannot be unfolded, variable bindings can sometimes be propagated by taking the least common generalisation of the matching clause heads. The latter technique recovers some of the benefits of unfolding, but it is only applicable if the bindings on the clause heads are mutually exclusive [3]. The unfolding restriction can also be avoided by removing cuts, and Debray [6] shows that many common uses of the cut can be removed by static analysis.

Secondly, the programmer can be forced to write programs in a more easily unfoldable style. The partial deduction system of Takeuchi & Furukawa [42] is restricted to programs written in if-then-else style. O'Keefe [29] advocates replacing most uses of the cut by if-then-else which can be unfolded easily, and the few cases which cannot be replaced will not greatly restrict the application of unfolding. However, it sometimes takes considerable skill to reorganise a program so that cuts can be replaced by if-then-else. In particular, disjunctions containing cuts cannot be directly mapped to the if-then-else form.

Thirdly, new pruning operators can be introduced, moving away from standard Prolog to languages which behave better under unfolding. This is the approach we follow, by eliminating all Prolog control constructs leaving only conjunction and ancestral cuts. Full Prolog can also be *augmented* with ancestral cuts, as done by Venken [49], which removes an unfolding restriction but not the complications of unfolding full Prolog. Part of Van Roy's Aquarius Prolog system [35] transforms Prolog into a simpler form called Kernel Prolog with ancestral cuts, eliminating if-then-else and cut in the same way as our approach. Kernel Prolog programs have a slightly different form to our simplified programs, using disjunctions and unification made explicit by introducing equalities. The Prolog implementation of Taylor [48] also has a mapping to a simpler form called normalised Prolog. Again, normalised Prolog programs are slightly different to simplified programs. They contain disjunctions, explicit unification and a version of the cut localised to disjunctions. Both these systems map programs to a simpler form to obtain efficient Prolog compilation. The Gödel language has a new pruning operator described by Hill, Lloyd and Shepherdson [16] which, like the ancestral cuts, has the property that unfolding does not change the meaning of programs.

There are two minor disadvantages with our approach. Firstly, it relies upon the use of ancestral cuts, which are not available in all Prologs. This is not really a problem because they are available in some well known implementations, for example Sepia and BIM Prolog. Secondly, the efficiency of if-then-else, negation-as-failure and once-only calls over cut is lost when simplifying programs. Recent work on high-performance Prolog implementations [35, 48] does not rely upon these control constructs, but to make our method useful for current Prologs we intend to investigate better post-unfolding optimisations, such as reintroduction of if-then-else. An

alternative approach would be to retain if-then-else when simplifying, then cuts could be replaced by ancestral cut, while negation-as-failure and once-only calls could be replaced by if-then-else. Although complicating the language somewhat, the control structure of the original program could be preserved more faithfully during transformation, giving the programmer more control over the result.

5.4 Correctness

Most strategies in the literature are correct, in the sense of preserving some program semantics, for example the success set, the finite failure set or the minimum Herbrand model. Ours is superimposed onto an unfold/fold system by Proietti & Pettorossi [33] which preserves the sequence of answer substitution semantics. Moreover, it always terminates whereas some do not, for example [1, 9, 25, 41].

5.5 Folding

Most partial deduction strategies use only the unfold rule. Owen [31] argues that folding is a useful rule, necessary to transform certain programs satisfactorily, whereas Lakhotia [22] argues that adding extra rules such as folding complicates PD unnecessarily. Our view is that unfold/fold transformation is an elegant method. However, a common problem with the unfold/fold method is that it is hard to automate. In particular, the introduction of useful definitions (by the definition rule) often requires user intervention, called the *Eureka* step in [4]. Our strategy automates the Eureka step and constructs a set of definitions which subsume all atoms encountered, as do those of Fujita [9] and de Schreye & Bruynooghe [38].

5.6 Self-application

Some partial deduction systems [10, 11] are self-applicable: that is, they can speed themselves up, specialise themselves with respect to interpreters to produce compilers, or even specialise themselves with respect to themselves to produce compiler-compilers. PADDY is written in Sepia and can handle Sepia, and therefore can be applied to itself, but the result is no faster than the original. This is partly because the implementation relies upon side-effects and failure-driven loops, which are useless for information propagation by unfolding. Another reason is that it is an online strategy. It is noted in [17] that the only partial evaluators so far which are successfully self-applicable use offline strategies. Self-application is a specialised research area outside the scope of this report.

5.7 Term abstraction

Many partial deduction strategies use some form of abstraction on atoms, to remove some information and hence ensure termination. We use an abstraction (the *pattern* in Section 2.2) related to *term depth abstractions* [45]. Gallagher [13] writes that this form of abstraction is generally too crude when used to remove information from atoms directly. However, we only use it indirectly as a hash function to locate an atom which may then be used to form an msg.

Appendix A

Examples

PADDY has been successfully applied to several programs, and been used for deductive database query optimisation [46] via meta-interpreter specialisation. Here we show sample programs with their transformed versions as produced by PADDY. For each example the pattern depth (Section 2.2) is set to $d = 5$. We use the usual Prolog syntax where predicate symbols and constants are written `x` and variables `X`.

A.1 Matrix transposition

This is a simple example from [13, 24] to show that specialised queries to recursive programs may sometimes be finitely unfoldable. It is one of the examples mentioned in Section 5.1.

```
transpose(A,[]) :- nullrows(A).
transpose(A,[B|C]) :- makerow(A,B,D), transpose(D,C).

makerow([],[],[]).
makerow([[A|B]|C],[A|D],[B|E]) :- makerow(C,D,E).

nullrows([]).
nullrows([[]|A]) :- nullrows(A).
```

The query clause is

```
test(A,B,C,D,E,F,G,H,I,R1,R2,T) :-
   transpose([[A,B,C,D,E,F,G,H,I],R1,R2],T).
```

and the result of the transformation is the unit clause

```
test(A,B,C,D,E,F,G,H,I,[J,K,L,M,N,O,P,Q,R],[S,T,U,V,W,X,Y,Z,Z1],
     [[A,J,S],[B,K,T],[C,L,U],[D,M,V],[E,N,W],
      [F,O,X],[G,P,Y],[H,Q,X],[I,R,Z1]]).
```

A.2 String matching

This example has been used in [9, 13, 24], and shows how a naive string matching program can be transformed into the Knuth-Morris-Pratt algorithm by partial deduction. It is also one of the examples mentioned in Section 5.1.

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

contains(A,B) :- con(B,([],A)).

con(A,(B,[])).
con([A|B],C) :- new(A,C,D), con(B,D).

new(A,(B,[A|C]),(D,C)) :- append(B,[A],D).
new(A,(B,[C|D]),(E,F)) :- A~=C, append(B,[A],G), append(E,H,B),
                          append(I,E,G), append(H,[C|D],F).

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

The query clause is

```
contains_aab(S) :- contains([a,a,b],S).
```

and the specialised program is

```
contains_aab(A) :- con-0(A).

con-0([a|A]) :- con-9(A).
con-0([A|B]) :- A~=a, con-0(B).

con-9([a|A]) :- con-21(A).
con-9([A|B]) :- A~=a, aux-39(B, A).

aux-39(A, B) :- con-0(A).
aux-39(A, a) :- con-9(A).

con-21([b|A]).
con-21([A|B]) :- A~=b, aux-37(B, A).

aux-37(A, B) :- con-0(A).
aux-37(A, a) :- con-9(A).
aux-37(A, a) :- con-21(A).
```

A.3 Tracer specialisation

This is a simple tracing interpreter for pure Prolog

```
tracer(Goal) :- tracer(Goal,0).

tracer((A,B),Depth) :- !, tracer(A,Depth), tracer(B,Depth).
tracer(Goal,Depth) :- system(Goal), !, Goal.
tracer(Goal,Depth) :- clau(Goal,Body), writeln(Depth-Goal),
                      Depth1 is Depth+1, tracer(Body,Depth1).

system(true).
system(_ is _).
```

specialised to a Fibonnacci object program

```
clau(fib(0,1),true).
clau(fib(1,1),true).
clau(fib(A,B),(A1 is A-1, A2 is A-2, fib(A1,B1), fib(A2,B2), B is B1+B2)).
```

The query clause is

```
tracerfib(A,B) :- tracer(fib(A,B)).
```

and the specialised interpreter is

```
tracerfib(A,B) :- aux-47(A,B,0).

aux-47(0,1,A) :- writeln(A-fib(0,1)), B is A+1.
aux-47(1,1,A) :- writeln(A-fib(1,1)), B is A+1.
aux-47(A,B,C) :- writeln(C-fib(A,B)), D is C+1, E is A-1, F is A-2,
                 aux-47(E,G,D), aux-47(F,H,D), B is G+H.
```

A.4 Compiling bottom-up to top-down

Here we have an interpreter which can interpreter programs which have been adorned to express a limited form of mixed top-down/bottom-up execution. Other work on compiling such execution strategies into a top-down strategy is in [39].

```
mixcomp(G) :- mixcomp(G,[],_).

mixcomp([],Li,Li).
mixcomp([A|B],Li,Lo) :- tstep(A,Li,Lt), mixcomp(B,Lt,Lo).

tstep(add_lemma(G),Li,Lo) :- tstep(G,Li,Lt), append(Lt,[G],Lo).
tstep(drop_first_lemma(G),[G|Li],Li).
tstep(drop_lemma(G),Li,Lo) :- delete(G,Li,Lo).
tstep(is_lemma(G),Li,Li) :- member(G,Li).
tstep(bottomup(G),Li,Li) :- member(G,Li).
tstep(bottomup(G),Li,Lo) :- brule(H,T), mixcomp(T,Li,Lt),
                            append(Lt,[H],Ls), bstep(G,H,Lo,Ls).
```

```
tstep(G,Li,Li) :- system(G), G.
tstep(G,Li,Lo) :- trule(G,T), mixcomp(T,Li,Lo).

bstep(G,G,Ls,Ls).
bstep(G,_,Lo,Ls) :- brule(H,T), mixcomp(T,Ls,Lt),
                    append(Lt,[H],Lt1), bstep(G,H,Lo,Lt1).

append([],G,G).
append([H|T],G,[H|S]) :- append(T,G,S).

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

delete(G,[G|L],L).
delete(G,[H|L],[H|X]) :- delete(G,L,X).

system(true).
system(_ is _).
```

This is to be specialised with respect to an adorned Fibonnacci program which generates lemmas in a bottom-up way:

```
trule(fib(0,1),[]).
trule(fib(1,1),[]).
brule(fib(N,X),[drop_first_lemma(fib(P,Z)),
                is_lemma(fib(Q,Y)),
                drop_lemma(Q is P+1),
                add_lemma(N is Q+1),
                X is Y+Z]).
```

The interpreter maintains a list of lemmas which may be added to by `add_lemma` (in which case the added lemma must first be proved by interpreting it), subtracted from by `drop_lemma` (non-deterministic) or `drop_first_lemma` (deterministic), or examined by `is_lemma` (non-deterministic).

In the recursive `fib` clause the first recursive `fib` call is taken from the lemmas and the second recursive call is also a lemma but is not deleted. The two calls to `is` which add 1 can also be treated as lemmas. First the last such lemma is deleted from the list, and then a new one is proved and added. The results of the two recursive calls are added, and then because the clause is a `brule` instead of a `trule` it is to be interpreted bottom-up, and so its head is added to the lemmas.

The query clause is

```
fib(I,F) :- mixcomp([add_lemma(fib(0,1)),
                     add_lemma(1 is 0+1),
                     add_lemma(fib(1,1)),
                     bottomup(fib(I,F))]).
```

The list of lemmas is initialised to contain two `fib` lemmas and one `is` lemma, and `fib` is called in bottom-up mode.

The specialised interpreter is

```
bfib(0,1).
bfib(1,1).
bfib(2,2).
bfib(3,3).
bfib(4,5).
bfib(A,B) :- bstep-152(A,B,C,5,4,5,8).

bstep-152(A,B,[bfib(C,D),A is C+1,bfib(A,B)],D,C,A,B).
bstep-152(A,B,C,D,E,F,G) :- H is F+1, I is G+D,
                            bstep-152(A,B,C,G,F,H,I).
```

The 3rd argument of `fib` is the lemma list, and plays no part in the computation, and could be removed by a more sophisticated redundant argument removal than PADDY currently has.

A.5 Lemma generation

Here we take the same interpreter with a differently adorned Fibonnacci program, which generates lemmas in a top-down way and reuses them. The object program is:

```
trule(fib(0,1),[]).
trule(fib(1,1),[]).
trule(fib(N,X),[Q is N-1,
                add_lemma(fib(Q,Y)),
                P is Q-1,
                is_lemma(fib(P,Z)),
                X is Y+Z]).
```

The strategy here is to add the first recursive `fib` call for reuse and to take the second `fib` call from the lemmas (it will have been added during interpretation of the first call). No lemmas are deleted in this program.

the query clause is

```
fib(I,F) :- mixcomp([add_lemma(fib(0,1)),fib(I,F)]).
```

and the specialised interpreter is

```
tfib(A,B) :- tstep-8(0,1,C), app-31(C,0,1,D), tstep-40(A,B,D,E).

tstep-8(0,1,[]).
tstep-8(1,1,[]).
tstep-8(A,B,C) :- D is A-1, tstep-8(D,E,F), app-31(F,D,E,C),
                  I is D-1, mem-20(I,J,C), B is E+J.

app-31([],A,B,[tfib(A,B)]).
```

```
app-31([A|B],C,D,[A|E]) :- app-31(B,C,D,E).

mem-20(A,B,[tfib(A,B)|C]).
mem-20(A,B,[C|D]) :- mem-20(A,B,D).

tstep-40(0,1,A,A).
tstep-40(1,1,A,A).
tstep-40(A,B,C,D) :- E is A-1, tstep-40(E,F,C,G), app-31(G,E,F,D),
                     H is E-1, mem-20(H,I,D), B is F+I.
```

This time the list of lemmas plays a part in the computation, because the length of the list is unbounded: lemmas are never deleted as they were in the previous example. Nevertheless, the specialised program is faster than the interpreted one.

Bibliography

[1] K.Benkerimi, J.W.Lloyd. A Procedure for the Partial Evaluation of Logic Programs. [53] pp.343-358.

[2] M.Bruynooghe, D. de Schreye, B.Martens. A General Criterion for Avoiding Infinite Unfolding During Partial Deduction of Logic Programs. *Proceedings of ILPS'91*.

[3] M.Bugliesi, F.Russo. Partial Evaluation in Prolog: Some Improvements About Cut. *Proceedings of the North American Conference on Logic Programming 1989.*

[4] R.Burstall, J.Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM* **24**(1), January 1977, pp.44-67.

[5] M.H.M.Cheng, M.H.van Emden, P.A.Strooper. Complete Sets of Frontiers in Logic-Based Program Transformation. *Proc. of the Workshop on Meta-Programming in Logic Programming,* 1988, pp.213-226.

[6] S.K.Debray. Towards Banishing the Cut from Prolog. *Proceedings of the International Conference on Computer Languages*, Miami, October 1986.

[7] S.K.Debray, D.S.Warren. Detection and Optimization of Functional Computations in Prolog. *Proceedings of the International Conference on Logic Programming,* 1986, pp.490-504.

[8] A.P.Ershov. Mixed Computation: Potential Applications and Problems for Study. Theoretical Computer Science **18**, 1982, pp.41-67.

[9] H.Fujita. An Algorithm for Partial Evaluation with Constraints. ICOT Technical Memorandum TM-0367, August 1987.

[10] H.Fujita, K.Furukawa. A Self-Applicable Partial Evaluator and Its Use in Incremental Compilation. [51] pp.91-118.

[11] D.A.Fuller, S.Abramsky. Mixed Computation of Prolog Programs. [51] pp.119-141.

[12] Y.Futamura. Partial Computation of Programs. In *E.Goto et al., editor, RIMS Symposia on Software Science and Engineering, Kyoto, Japan 1982,* Lecture Notes in Computer Science **147**, Springer-Verlag 1982, pp.1-35.

[13] J.Gallagher. A System for Specialising Logic Programs. TR-91-32, University of Bristol, Computer Science Department.

[14] J.Gallagher, M.Bruynooghe. Some Low-Level Source Transformations for Logic Programs. [52] pp.229-244.

[15] P.Gardner, J.C.Shepherdson. Unfold-Fold Transformations of Logic Programs. Report No. PM-89-01, School of Mathematics, University of Bristol 1989.

[16] P.M.Hill, J.W.Lloyd. J.C.Shepherdson. Properties of a Pruning Operator. *Journal of Logic and Computation* **1**(1) pp.99-143, 1990.

[17] N.D.Jones. Challenging Problems in Partial Evaluation and Mixed Computation. [51] pp.291-302.

[18] N.D.Jones. Automatic Program Specialization: a Re-examination from Basic Principles. *Proceedings of the Workshop on Partial Evaluation and Mixed Computation,* (D.Bjørner, A.P.Ershov, N.D.Jones eds.), North-Holland 1988.

[19] T.Kanamori, H.Fujita. Unfold/Fold Transformation of Logic Programs with Counters. ICOT Technical Report TR-179, May 1986.

[20] J.Komorowski. Partial Evaluation as a means for inferencing data structures in an applicative language: a theory and implementation in the case of Prolog. *Proceedings of the ACM Symposium on Principles of Programming Languages,* 1982, pp.255-267.

[21] P.Kursawe. Pure Partial Evaluation and Instantiation. *Proceedings of the IFIP TC2 Workshop On Partial Evaluation And Mixed Computation,* North Holland 1987, eds. D.Bjørner, A.P.Ershov, N.D.Jones.

[22] A.Lakhotia. To PE or not to PE. [52] pp.218-228.

[23] A.Lakhotia, L.Sterling. ProMiX: a Prolog Partial Evaluation System. The Practice of Prolog, editor L.Sterling, MIT Press 1991.

[24] J.K.K.Lam, A.J.Kusalik. A Partial Evaluation of Partial Evaluators for Pure Prolog. Technical Report 90-9, Department of Computational Science, University of Saskatchewan, Canada, 1990.

[25] G.Levi, G.Sardu. Partial Evaluation of Metaprograms in a 'Multiple Worlds" Logic Language. [51] pp.227-247.

[26] J.W.Lloyd, J.C.Shepherdson. Partial Evaluation in Logic Programming. Technical Report CS-87-09, University of Bristol 1987.

[27] Z.Manna, R.Waldinger. Synthesis: Dreams ⇒ Programs. *IEEE Trans. Soft Eng.* **5**(4), July 1979, pp.294-328.

[28] M.Meier et al. SEPIA Version 3.0 User Manual. SEP/UMS/064 ECRC June 1990.

[29] R.O'Keefe. On the Treatment of Cuts in Prolog Source-Level Tools. *Symposium on Logic Programming, IEEE* 1985.

[30] R.O'Keefe. The Craft of Prolog. MIT Press 1990.

[31] S.Owen. Issues in the Partial Evaluation of Meta-Interpreters. *Proceedings of the Workshop on Meta-Programming in Logic Programming*, 1988, pp.241-254.

[32] M.Proietti, A.Pettorossi. Construction of Efficient Logic Programs by Loop Absorption and Generalisation. [52] pp.57-81.

[33] M.Proietti, A.Pettorossi. Semantics Preserving Transformation Rules for Prolog. Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, *SIGPLAN NOTICES* **26** no.9, September 1991, pp.274-284.

[34] M.Proietti, A.Pettorossi. Unfolding-Definition-Folding, in this Order, for Avoiding Unnecessary Variables in Logic Programs. *Symposium on Program Language Implementation and Logic Programming,* 1991, pp.347-358.

[35] P.L.Van Roy. Can Logic Programming Execute as Fast as Imperative Programming. Report no.UCB/CSD 90/600 Dec.1990, Computer Science Division (EECS), University of California, Berkeley, California 94720.

[36] D.Sahlin. The Mixtus Approach to Automatic Partial Evaluation of Full Prolog. [53] pp.377-398.

[37] H.Sawamura, T.Takeshima, A.Kato. Source-Level Optimization Techniques for Prolog. ICOT Technical Report TR-91, January 1985.

[38] D. de Schreye, M.Bruynooghe. On the Transformation of Logic Programs with Instantiation Based Computation Rules. *Journal of Symbolic Computation* **7**, pp.125-154, 1989.

[39] D. de Schreye, B.Martens, G.Sablon, M.Bruynooghe. Compiling bottom-up and mixed derivations into top-down executable logic programs. [52] pp.37-56.

[40] H.Seki. Unfold/Fold Transformation of Stratified Programs. *Proc. of the Int. Conf. on Logic Programming* 1989, pp.554-568.

[41] L.Sterling, R.D.Beer. Incremental Flavor-Mixing of Meta-interpreters for Expert System Construction. *Proceedings of the Symposium on Logic Programming,* Salt Lake City, Utah 1986, pp.20-27.

[42] A.Takeuchi, K.Furukawa. Partial Evaluation of Prolog Programs and its Application to Meta Programming. ICOT Technical Report TR-126, July 1985.

[43] H.Tamaki, T.Sato. A Transformation System for Logic Programs Which Preserves Equivalence. ICOT Technical Report TR-18, August 1983.

[44] H.Tamaki, T.Sato. A Generalized Correctness Proof of the Unfold/Fold Logic Program Transformation. Department of Information Science TR 86-04, Ibaraki University, June 1986.

[45] H.Tamaki, T.Sato. OLD Resolution with Tabulation. *Proceedings of the 3rd International Conference in Logic Programming, LNCS* **225**, Springer-Verlag 1986.

[46] J.L.Träff, S.D.Prestwich. Meta-Programming for Reordering Literals in Deductive Databases. To appear in *Proceedings of META 92*, Uppsala, Sweden.

[47] V.F.Turchin. The Concept of a Supercompiler. *ACM TOPLAS* **8**(3) pp.292-325, 1986.

[48] A.Taylor. High Performance Prolog Implementation. PhD thesis, Basser Department of Computer Science, University of Sidney, June 1991.

[49] R.Venken. A Prolog Meta-Interpreter for Partial Evaluation and its Application to Source to Source Transformation and Query Optimisation, *Proceedings of the European Conference on Artificial Intelligence,* 1984.

[50] R.Venken, B.Demoen. A Partial Evaluation System for Prolog: some Practical Considerations. [51] pp.279-290.

[51] *New Generation Computing* **6**(2,3) 1988.

[52] *Proceedings of the 2nd Workshop on Meta-Programming,* 1990.

[53] *Proceedings of the North American Conference on Logic Programming,* 1990.

Other Reports Available from ECRC

[ECRC–TR–LP-60] Mireille Ducasse and Anna-Maria Emde. *Opium 3.1 - User Manual A High-level Debugging Environment for Prolog.* 1991.

[ECRC–TR–LP-61] E. Yardeni, T. Frühwirth, and E. Shapiro. *Polymorphically Typed Logic Programs.* 1991.

[ECRC–TR–DPS-81] U. Baron, S. Bescos, and S. Delgado. *The ElipSys Logic Programming Language.* 17. 01. 1991.

[ECRC–TR–DPS-82] Sergio Delgado, Michel Dorochevsky, and Kees Schuerman. *A Shared Environment Parallel Logic Programming System On Distributed Memory Architectures.* 18. 01. 1991.

[ECRC–TR–DPS-83] Andre Veron, Jiyang Xu, and Kees Schuerman. *Virtual Memory Support for OR-Parallel Logic Programming Systems.* 05. 03. 1991.

[ECRC–TR–DPS-85] Michel Dorochevsky. *Garbage Collection in the OR-Parallel Logic Programming.* 15. 03. 1991.

[ECRC–TR–DPS-100] Alan Sexton. *KCM Kernel Implementation Report.* 22. 05. 1991.

[ECRC–TR–DPS-103] Michel Dorochevsky. *Key Features of a Prolog Module System.* 08. 03. 1991.

[ECRC–TR–DPS-104] Michel Dorochevsky, Kees Schuerman, and Andre Veron. *ElipSys: An Integrated Platform for Building Large Decision Support Systems.* 29. 01. 1991.

[ECRC–TR–DPS-105] Jiyang Xu and Andre Veron. *Types and Constraints in the Parallel Logic Programming System ElipSys.* 15. 03. 1991.

[ECRC–TR–DPS-107] Olivier Thibault. *Design and Evaluation of a Symbolic Processor.* 13. 06. 1991.

[ECRC–TR–DPS-112] Michel Dorochevsky, Jacques Noyé, and Olivier Thibault. *Has Dedicated Hardware for Prolog a Future ?* 14. 09. 1991.

[ECRC–91–1] Norbert Eisinger and Hans Jürgen Ohlbach. *Deduction Systems Based on Resolution.* 29. 10. 1991.

[ECRC–91–2] Michel Kuntz. *The Gist of GIUKU: Graphical Interactive Intelligent Utilities for Knowledgeable Users of Data Base Systems.* 4. 11. 1991.

[ECRC–91–3] Michel Kuntz. *An Introduction to GIUKU: Graphical Interactive Intelligent Utilities for Knowledgeable Users of Data Base Systems.* 4. 11. 1991.

[ECRC–91–4] Michel Kuntz. *Enhanced Graphical Browsing Techniques for Collections of Structured Data.* 4. 11. 1991.

[ECRC–91–5] Michel Kuntz. *A Graphical Syntax Facility for Knowledge Base Languages.* 4. 11. 1991.

[ECRC–91–6] Michel Kuntz. *A Versatile Browser-Editor for NF2 Relations.* 4. 11. 1991.

[ECRC–91–7] Norbert Eisinger, Nabiel Elshiewy, and Remo Pareschi. *Distributed Artificial Intelligence - An Overview.* 4. 11. 1991.

[ECRC–91–8] Norbert Eisinger. *An Approach to Multi-Agent Problem-Solving.* 11. 11. 1991.

[ECRC–91–9] Klaus H. Ahlers, Michael Fendt, Marc Herrmann, Isabelle Hounieu, and Philippe Marchal. *TUBE Implementor's Manual.* 21. 11. 1991.

[ECRC–91–10] Klaus H. Ahlers, Michael Fendt, Marc Herrmann, Isabelle Hounieu, and Philippe Marchal. *TUBE Programmer's Manual.* 21. 11. 1991.

[ECRC–91–11] Michael Dahmen. *A Debugger for Constraints in Prolog.* 26. 11. 1991.

[ECRC–91–12] Jean-Marc Andreoli and Remo Pareschi. *Communication as Fair Distribution of Knowledge.* 26. 11. 1991.

[ECRC–91–13] Jean-Marc Andreoli, Remo Pareschi, and Marc Bourgois. *Dynamic Programming as Multiagent Programming.* 26. 11. 1991.

[ECRC–91–14] Volker Küchenhoff. *On the Efficient Computation of the Difference Between Consecutive Database States.* 5. 12. 1991.

[ECRC–91–15] Sylvie Bescos and Michael Ratcliffe. *Secondary Structure Prediction of rRNA Molecules Using ElipSys.* 16. 12. 1991.

[ECRC–91–16] Michael Dahmen. *Abstract Debugging of Coroutines and Constraints in Prolog.* 30. 12. 1991.

[ECRC–92–1] Thierry Le Provost and Mark Wallace. *Constraint Satisfaction Over the CLP Scheme.* 30. 1. 1992.

[ECRC–92–2] Gérard Comyn, M. Jarke, and Suryanarayana M. Sripada. *Proceedings of the 1st Compulog Net meeting on Knowledge Bases (CNKBS'92).* 30. 1. 1992.

[ECRC–92–3] Jesper Larsson Traeff and Steven David Prestwich. *Meta-programming for re-ordering Literals in Deductive Databases.* 30. 1. 1992.

[ECRC–92–4] Beat Wüthrich. *Update Realizations Drawn from Knowledge Base Schemas and Executed by Dialog.* 4. 2. 1992.

[ECRC–92–5] Lone Leth. *A New Direction in Functions as Processes.* 25. 2. 1992.

[ECRC–92–6] Steven David Prestwich. *The PADDY Partial Deduction System.* 23. 3. 1992.

[ECRC–92–7] Andrei Voronkov. *Extracting Higher Order Functions from First Order Proofs.* 23. 3. 1992.

[ECRC–92–8] Andrei Voronkov. *On Computability by Logic Programs.* 23. 3. 1992.

[ECRC–92–9] Beat Wüthrich. *Towards Probabilistic Knowledge Bases.* 02. 4. 1992.

[ECRC–92–10] Petra Bayer. *Update Propagation for Integrity Checking, Materialized View Maintenance and Production Rule Triggering.* 08. 4. 1992.

[ECRC–92–11] Mireille Ducassé. *Abstract views of Prolog executions in Opium.* 15. 4. 1992.

[ECRC–92–12] Alexandre Lefebvre. *Towards an Efficient Evaluation of Recursive Aggregates in Deductive Databases.* 30. 4. 1992.

[ECRC–92–13] Udo W. Lipeck and Rainer Manthey (Hrsg.). *Kurzfassungen des 4. GI-Workshops "Grundlagen von Datenbanken", Barsinghausen, 9.-12.6.1992.* 12. 05. 1992.

[ECRC–92–14] Lone Leth and Bent Thomsen. *Some Facile Chemistry.* 26. 05. 1992.

[ECRC–92–15] Jacques Noyé (Ed.). *Proceedings of the International KCM User Group Meeting,Munich, 7 and 8 October 1991.* 03. 06. 1992.

[ECRC–92–16] Frederick Knabe. *A Distributed Protocol for Channel-Based Communication with Choice.* 10. 06. 1992.

[ECRC–92-17] Benoit Baurens, Petra Bayer, Luis Hermosilla, and Andrea Sikeler. *Publication Management: A Requirements Analysis.* 03. 07. 1992.

[ECRC–92-18] Thom Frühwirth. *Constraint Simplification Rules.* 28. 07. 1992.

[ECRC–92-19] Mark Wallace. *Compiling Integrity Checking into Update Procedures.* 29. 07. 1992.

[ECRC–92-20] Petra Bayer. *Data and Knowledge for Medical Applications: A Case Study.* 30. 07. 1992.

[ECRC–92-21] Michel Dorochevsky and André Véron. *Binding Techniques and Garbage Collection for OR-Parallel CLP Systems.* 11. 08. 1992.

[ECRC–92-22] Shan-Wen Yan. *Efficiently Estimating Relative Grain Size for Logic Programs on Basis of Abstract Interpretation.* 25. 08. 1992.

[ECRC–92-23] Jean-Marc Andreoli, Paolo Ciancarini, and Remo Pareschi. *Interaction Abstract Machines.* 25. 08. 1992.

[ECRC–92-24] Jean-Marc Andreoli and Remo Pareschi. *Associative Communication and its Optimization via Abstract Interpretation.* 25. 08. 1992.

[ECRC–92-25] Jean-Marc Andreoli, Lone Leth, Remo Pareschi, and Bent Thomsen. *On the Chemistry of Broadcasting.* 25. 08. 1992.

[ECRC–92-26] Marc Bourgois, Jean-Marc Andreoli, and Remo Pareschi. *Extending Objects with Rules, Composition and Concurrency : the LO Experience.* 25. 08. 1992.

[ECRC–92-27] Benoit Dageville and Kam-Fai Wong. *SIM: A C-based SIMulation Package.* 28. 09. 1992.

[ECRC–92-28] Beat Wüthrich. *On the Efficient Distribution-free Learning of Rule Uncertainties and their Integration into Probabilistic Knowledge Bases.* 29. 09. 1992.

[ECRC–92-29] Andrei Voronkov. *Logic Programming with Bounded Quantifiers.* 29. 09. 1992.

[ECRC–92-30] Eric Monfroy. *Gröbner Bases: Strategies and Applications.* 30. 09. 1992.

[ECRC–92-31] Eric Monfroy. *Specification of Geometrical Constraints.* 30. 09. 1992.

[ECRC–92-32] Bent Thomsen, Lone Leth, and Alessandro Giacalone. *Some Issues in the Semantics of Facile Distributed Programming.* 22. 10. 1992.

[ECRC–92-33] Mireille Ducassé. *An Extendable Trace Analyser to Support Automated Debugging.* 04. 12. 1992.

[ECRC–92-34] Jorge Bocca and Luis Hermosilla. *A Preliminary Study of the Performance of MegaLog.* 20. 12. 1992.

[ECRC–93–1] Benoit Dageville and Kam-Fai Wong. *Supporting Thousands of Threads Using a Hybrid Stack Sharing Scheme.* 18. 01. 1993.

[ECRC–93–2] Steven Prestwich. *ElipSys Programming Tutorial.* 18. 01. 1993.

[ECRC–93–3] Beat Wüthrich. *Learning Probabilistic Rules.* 28. 01. 1993.

[ECRC–93–4] Eric Monfroy. *A Survey of Non-Linear Solvers.* 02. 02. 1993.

[ECRC–93–5] Thom Frühwirth, Alexander Herold, Volker Küchenhoff, Thierry Le Provost, Pierre Lim, Eric Monfroy, and Mark Wallace. *Constraint Logic Programming - An Informal Introduction.* 02. 02. 1993.

[ECRC–93–6] *ECLiPSe User Manual.* 23. 03. 1993.

[ECRC–93–7] Petra Bayer, Alexandre Lefebvre, and Laurent Vieille. *Architecture and Design of the EKS Deductive Database System.* 29. 03. 1993.

[ECRC–93–8] Petra Bayer and John Fox. *State-Of-The-Art Report on Reactive Processing in Database Technology and in Artificial Intelligence.* 29. 03. 1993.