NORTH-HOLLAND

TECHNICAL NOTE

ON DELPHI LEMMAS AND OTHER MEMOING TECHNIQUES FOR DETERMINISTIC LOGIC PROGRAMS*

PAUL TARAU, KOEN DE BOSSCHERE, AND BART DEMOEN

▷ Three orthogonal memoing techniques for deterministic logic programs are introduced and evaluated on the basis of their empirical performance. They share the same basic idea: efficient memoing is achieved by losing information gracefully, i.e., memoing benefits from a form of *abstraction*.

Abstract answers are most general computed answers of deterministic logic programs obtained through repeated applications of a simple *clause composition operator*. After describing a meta-interpreter returning abstract answers, we derive a class of program transformations that compute abstract answers more efficiently: they are ideal lemmas due to their *goal-independent* nature. For this reason, their "hit rate" is usually higher than in the case of conventional memoing.

Indexing by structural properties of terms is an effective way to speed up the retrieval of lemmas, especially in the case of simple programs using linear recursion.

Delphi lemmas add a self-adjusting control mechanism on the amount of memoing. Answers are memoized only by acquiescence of an oracle. We show that random oracles perform surprisingly well as Delphi lemmas tend naturally to cover the "hot spots" of the program.

*This paper is a revised version of the paper "Memoing with Abstract Answers and Delphi Lemmas" presented at the LOPSTR'93 Workshop [26].

Address correspondence to Paul Tarau, Département d'Informatique, Université de Moncton, Moncton, N.B. EIA-2E9, Canada, E-mail: tarau@info.umoncton.ca; Koen De Bosschere, Vakgroep Elektronica en Informatiesystemen, Universiteit Gent, Gent, Belgium, E-mail: kdb@elis.rug.ac.be; Bart Demoen, Department of Computer Science, Katholieke Universiteit Leuven, Leuven, Belgium, E-mail: bimbart@cs.kuleuven.ac.be.

Received November 1995; accepted August 1996.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010

0743-1066/97/$17.00

PII S0743-1066(96)00105-7

A subset of our memoing techniques has been incorporated in BinProlog 5.00 with a declaration-based preprocessor. ◁

1. INTRODUCTION

Memoing techniques have been an important research topic in logic programming and deductive databases (see [21, 27]). Various practical tools for memoing exist from programmer-defined `assert`-based mechanisms (see [17]) and extension tables as in XSB-Prolog to dedicated interpreters like [19, 20, 28]. Without minimizing the merits of the memoing facilities previously mentioned (which support execution of left-recursive rules and ensure termination[1] of Datalog programs [16, 27]), we have noticed that some of their weaknesses lead to the following symptoms/problems:

- static configuration: a fixed memoing algorithm is used instead of a self-adjusting mechanism;
- overmemoing: they do not try to get a small set of high "hit rate" lemmas with minimal computational resources;
- lack of abstraction: atoms obtained in the course of resolution are goal-dependent.

This paper tries to solve these problems in a simple and radical way. First, instead of memoing actual instances of answers created during the resolution process, we memo their more general instances such that, while preserving soundness, we can ensure the best possible future reuse. We show that most of the overhead of abstract answer computation can be compiled away by using program transformations.

Second, we abolish the usual predictability of *what* is memoized and *when* by delegating it to an oracle external to the resolution process. Due to statistical properties of execution traces, this is surprisingly better with a random—but tunable—oracle than with conventional fixed algorithm memoing approaches, especially when the programmer has explicit control of the amount of memoing. The use of such an oracle will henceforth be called the *Delphi principle*.

Finally, in order to limit the amount of searching in the list of lemmas, we propose indexing by structural properties. For programs using linear recursion, it is fairly easy to find a property that can be used for indexing. For list processing predicates, the most evident property is the list length. Indexing allows us to access any lemma in constant time.

The paper is organized as follows. Section 2 proposes abstract answers, and a number of ways of computing them. Section 3 proposes the Delphi principle, gives some examples, and proposes performance results. The paper ends with some directions of future work and a conclusion.

[1] Improving termination properties is an important use of memoing in the deductive database and nonmonotonic reasoning communities. In contrast, in this paper, memoing is seen as a means to optimize execution time with a minimal impact on space consumption.

2. MEMOING WITH ABSTRACT ANSWERS

2.1. Derivations with Clause Compositions

Although classical texts on SLD and SLDNF resolution (see [1] and [13]) do explain well the basic logical mechanisms behind Prolog engines, one aspect is neglected in all but some partial evaluation-oriented papers like [9] and [14][2]: the resolvent is seen as a conjunction of literals instead of being seen as a logical implication. For instance, in the case of SLD resolution, starting from a goal `?-G` hides the fact that we are actually looking for a computed answer starting from something like `answer(X):-G`. In that case, we do not have to think about resolution as a "refutation" procedure. Clearly, we can start from the tautologically true clause `G:-G`, and simply unfold clauses until a fact `A:-true` is eventually reached.

This reformulation of resolution theory is an instance of *resultant*-based descriptions as found in [9, 14, 24, 25] and *S-semantics* [8] and is beyond the scope of this paper. We will specialize these results to Prolog's computation rule by introducing a *composition* operation $\oplus$ that combines clauses by unfolding the leftmost body-goal of the first argument with the second clause.

Definition 2.1. Let $A_0 :- A_1, A_2, \ldots, A_n$ and $B_0 :- B_1, \ldots, B_m$ be two clauses (suppose $n > 0$, $m \geq 0$). We define

$$(A_0 :- A_1, A_2, \ldots, A_n) \oplus (B_0 :- B_1, \ldots, B_m) = (A_0 :- B_1, \ldots, B_m, A_2, \ldots, A_n)\theta$$

with $\theta = \text{mgu}(A_1, B_0)$. If the atoms A_1 and B_0 do not unify, the result of the composition is denoted as $\bot$.[3]

Furthermore, as usual, we consider $A_0 :- \text{true}, A_2, \ldots, A_n$ to be equivalent to $A_0 :- A_2, \ldots, A_n$ and for any clause C, $\bot \oplus C = C \oplus \bot = \bot$. We suppose that the two operands are standardized apart, and that a mgu is selected unambiguously (as in Prolog) through variable ordering or similar means.

Repeated clause compositions can be used to describe a "Prolog-like" inference rule called LD resolution. Note also that clause composition is an associative[4] operation, and therefore a "sequence" of such compositions is well defined.

Definition 2.2. An LD derivation is a sequence of *clauses* $C_1, \ldots, C_n$ such that the result of their composition $C_1 \oplus \cdots \oplus C_n$ is different from $\bot$.

Let P be a definite program, and let `G` be an atomic definite goal. We can then construct the clause `G:-G`. which is a logical tautology. Derivations starting with `G:-G` are of special interest as they can be used to produce computed answers for `G` and P by applying $\oplus$ to it with clauses from P.

[2] In [9], a definition of most general *resultants* is given, which are seen as logical implications between conjunctions.

[3] We suppose that an (implementation defined) choice is made for the mgu. To keep things simple, we will also work with terms instead of equivalence classes of terms up to a renaming, and refer to [2, 11, 14] for a more precise formal description. In terms of [14], this can be seen as an instance of GSLD resolution (see Lemmas 4.1 and 4.2).

[4] This follows from the associativity application [13], and in particular, that of mgus.

2.2. *Standard versus Abstract Answers*

We will now describe two useful instances of derivations and answers: *standard* and *abstract* answers.

Definition 2.3. Let `P` be a program of `G` an atomic goal. A derivation starting with `G:-G` and resulting from a *fact* `A:-true` occurring in `P` is called a *standard LD derivation*. A derivation starting with a clause of `P` and resulting in a fact is called an *abstract LD derivation*. A *standard resp. abstract* answer is the result of the composition of the clauses occurring in a *standard resp. abstract* LD derivation.

Composing arbitrary clauses of a program is not practical without some guidance from goal-dependent computations. Fortunately, for each standard derivation, there is a corresponding abstract deviation obtained by dropping its first element `G:-G`. Clearly, such a derivation still gives a resultant of the form `G':-true`, i.e., it computes an *abstract answer* of the program. Due to the associativity of clause composition, the original goal-dependent standard answer is obtainable in one step from this abstract answer as `T:-G` $\oplus$ `G':-true`.

Also, every abstract answer can be obtained as a standard answer by starting the LD derivation with `G:-G` where the arguments of `G` are independent variables. This observation will be used when implementing the computation of an abstract answer guided by the computation of a standard answer.

The following example shows how an abstract answer can be composed from clauses of the program on top of a standard derivation which works as a "path-finder" for the corresponding abstract derivation.

Example 2.1. Let us consider the program

(C1) `plus(0,Y,Y):-true.`

(C2) `plus(s(X),Y,s(Z)):-plus(X,Y,Z).`

and `G:-G=`

`plus(s(0),s(0),R):-plus(s(0),s(0),R).`

We obtain, as the result of `(G:-G)` $\oplus C_2$,

`plus(s(0),s(0),s(Z1)):-plus(0,s(0),Z1).`

Then, by computing with (C1), the expression `(G:-G)` $\oplus C_2 \oplus C_1$ is equal to the standard answer:

`plus(s(0),s(0),s(s(0))):-true.`

The corresponding abstract answer ($C_1 \oplus C_1$), obtained by omitting the first composition

`plus(s(0),A,S(A)):-true.`

contains the useful generalization that "(the successor of `0`) plus `A` is the successor of `A`."

Note that working with clause compositions ensures that each step of a derivation corresponds to a logical consequence of the program. For a detailed discussion of this topic, we refer to [2, 4, 11].

In the examples that follow, we will often use `G` instead of `G:-true` for convenience.

2.3. *A Meta-Interpreter for Abstract Answers*

The following code (working on *definite programs*) is obtained from the "vanilla" meta-interpreter, but its extension to more sophisticated meta-interpreters dealing with *cut*, negation, and system predicates can be done with well-known meta-interpretation techniques.

```
% demo(G,AbsG) is true if AbsG is an abstract answer of P
% the clauses of P being given by ''clause/3''
demo(true,true):-!.
demo((A,B),(AbsA,AbsB)):-!,demo(A,AbsA),demo(B,AbsB).
demo(H,AbsH):-
        clause(H,B,Ref),
        demo(B,AbsB),
        instance(Ref,(AbsH:-AbsB)).
```

The interpreter uses two system predicates available on Prolog systems `clause/3` which returns a database reference `Ref` and `instance/2` which returns a new variant of the clause referred to by `Ref`. Note that instance/2 is only called after demo/2, so that its expensive copy operation is only performed for successful branches.

Example 2.2. If $P =$

```
app([],Ys,Ys)
app([A|Xa],Ys,[A|Zs]):-app(Xs,Ys,Zs).
nrev([],[]).
nrev([X|Xs],R):-nrev(Xs,T), app(T,[X],R).
```

and

```
G=nrev([a,b,c],X)
```

we obtain in the query

```
?-G=nrev([a,b,c],X),demo(G,Abstract).
```

the following abstract answer:

```
Abstract=nrev([A,B,C],[C,B,A]).
```

2.4. *Computing Abstract Answers with a Program Transformation*

We can obtain the same result without meta-interpretation by constructing a transformed program. We will define a general program transformation scheme, and then study three of its instances, each specified by a different transformation ϕ.

For simplicity, we will restrict ourselves to the case of definite programs. However, some built-ins and *cut* can be accommodated easily in this framework by adding rules to the transformation ϕ.

Definition 2.4. Let $ABS(\phi, P)$ be the program obtained by replacing each clause $A_0 :- A_1, \ldots, A_n$ of the program P by $\phi(A_0, A'^0) :- \phi(A_1, A_{1'}), \ldots, \phi(A_n, A'_n)$, where $A'_0 :- A_1, \ldots, A'_n$ is a standardized apart variant of $A_0 :- A_1, \ldots, A_n$.

Note that this class of transformations can be easily derived by partial evaluation of the meta-interpreter presented before. Using the program transformation $ABS(\phi, P)$ is not only an order of magnitude more efficient than meta-interpretation of P, but also conserves the operational meaning of programs containing *cut* or system predicates provided they are left unchanged by $ABS(\phi, P)$.

We will now present three possible instances of the transformation ϕ.

2.4.1. Basic Transformation. Let us define $\phi(a(X_1, \ldots, X_n), a(Y_1, \ldots, Y_n))$ as the atom $p(a(X_1, \ldots, X_n), a(Y_1, \ldots, Y_n))$, where $p/2$ is a new fixed predicate symbol.

Example 2.3. $ABS(basic, P) =$

```
p(append([],Ys,Ys),append([],Ys1,Ys1)).
p(append([A|Xs],Ys,[A|Zs]),append([A1|Xs1],Ys1,
[A1|Zs1])):-
  p(append(Xs,Ys,Zs),append(Xs1,Ys1,Zs1)).
p(nrev([],[]),nrev([],[])).
p(nrev([X|Xs],R),nrev([X1|Xs1],R1)):-
  p(nrev(Xs,T),nrev(Xs1,T1)),
  p(append(T,[X],R),append(T1,[X1],R1)).
```

The query

```
?-p(nrev([a,b,c],_),Abstract).
```

returns the abstract answer

```
Abstract-nrev([A,B,C],[C,B,A]).
```

2.4.2. Indexed Transformation. We can define $\phi(a(X_1, \ldots, X_n), a(Y_1, \ldots, Y_n))$ as $a(X_1, \ldots, X_n, a(Y_1, \ldots, Y_n))$ preserving the usual first argument indexing of the original program.

Example 2.4. $ABS(indexed, P) =$

```
append([],Ys,Ys,append([],Ys1,Ys1)).
append([A|Xs],Ys,[A|Zs],append([A1|Xs1],Ys1,
[A1|Zs1])):-
  append(Xs,Ys,Zs,append(Xs1,Ys1,Zs1)).
nrev([],[],nrev([],[])).
nrev([X|Xs],R,nrev([X1|Xs1],R1)):-
  nrev(Xs,T,nrev(Xs1,T1)),
  append(T,[X],R,append(T1,[X1],R1)).
```

The query

```
?-nrev([a,b,c],_,Abstract).
```

returns the abstract answer

```
Abstract=nrev([A,B,C],[C,B,A]).
```

2.4.3. Flat Transformation. Finally, by defining ϕ(a($X_1, \ldots, X_n$), a($Y_1, \ldots, Y_n$)) as being simply a($X_1, \ldots, X_n, Y_1, \ldots, Y_n$), we can avoid the construction of useless structures.

Example 2.5. ABS(flat, P) =

```
append([],Ys,Ys,[],Ys1,Ys1).
append([A|Xs],Ys,[A|Zs],[A1|Xs1],Ys1,[A1|Zs1]):-
  append(Xs,Ys,Zs,Xs1,Ys1,Zs1).
nrev([],[],[],[]).
nrev([X|Xs],R,[X1|Xs1],R1):-
  nrev(Xs,T,Xs1,T1),
  append(T,[X],R,T1,[X1],R1).
```

The query

```
?-nrev([a,b,c],_,X,Y), Abstract=nrev(X,Y).
```

returns the abstract answer

```
Abstract=nrev([A,B,C],[C,B,A]).
```

The transformed program generates precisely the same standard answers as the original program while computing the associated abstract answers too.

Note that the *basic*, *indexed*, and *flat* transformations compute abstract answers together with standard answers within a constant factor from computing only the standard answers for *P*. This follows immediately from the fact that terms involved in the computation of standard answers are always more specific than those used to compute abstract answers, for each derivation step. Therefore, the computational effort for an abstract solution is at most as large as the effort spent in the computation of the standard solution.[5]

The execution speeds are given in Table 1. It shows that the program transformation is much faster than the meta-interpreter, and that in the case of naive reverse, the overheads of the computation of an abstract answer using our best transformation `flat` is limited to 57% w.r.t. direct execution.

[5] In practice, some low-level issues (register availability, locality of data and code, etc.) can influence the constant factor adversely.

TABLE 1. Execution speeds for the program transformations on a Sparcstation ELC with SICStus Prolog 2.1 compact code

Version	Runtime (μs)	Slow Down w.r.t. `nrev`
Original `nrev`	105	1.00
Demo	3174	30.22
Basic	374	3.56
Index	208	1.98
Flat	165	1.57

2.5. *Abstract Answers as Lemmas*

Abstract answers are good candidates for reusable lemmmas. By accumulating an abstract answer `S:-true`, the equivalent of the search for an entire LD refutation can be replaced with the composition of the clause `G:-G` for a given atomic goal `G` and a memoized abstract answer `S:-true`.[6]

Notice the nontrivial nature of lemmas obtained from abstract answers as they replace possibly infinite sets of standard answers. Compared to memoing of standard answers, memoing of abstract answers is more appropriate as the generality of the saved computation allows a better hit rate. Soundness is ensured as they are logical consequences of the program.

The computational overhead of this kind of lemma generation is minimized as the computation of abstract answers can be "compiled" through the program transformation $P \rightarrow ABS(\phi, P)$.

However, as is also the case with usual lemma generation, the actual gain in efficiency depends on the indexing of dynamic code and programmer-defined "pragmas" specifying what is worth being memoized. More precisely, lemmas will turn out to be useful only if

One-execution > lemma-use
+ (one-execution-with-overhead + lemma-creation) /
number-of-uses.

Note that this is an approximation of what actually can happen due to interaction with contextual factors. For instance, generating an abstract answer has the same or lower cost than the standard one; but the extra cost of the transformed program —which is supported to use the lemma—is more than in the generation of the abstract answer; we also have to take account of the fact that every predicate is preceded by a test as to whether a lemma can be used. Also, in the case of abstract lemmas, we have to put in the right side execution times for the transformed program, which are, as we noted earlier, at most twice as long as the execution times for the original program (i.e., one-execution-with-overhead < 2*one-execution).

The effect of memoing abstract answers instead of standard answers can be seen nicely from a benchmark which executes many times `nrev/2` with lists of the same length, but with random values in the list: a Prolog system that memos standard answers will memo, e.g., `nrev([12,13,1,5,9],[9,5,1,13,12])`, and will not be able to use this lemma for a later call like `nrev([1,2,3,4,5],X)`. On the other hand, memoing the abstract answer `nrev([A,B,C,D,E],[E,D,C,B,A])` allows us to reuse the lemma many times.

2.6. *Indexing*

Asserting abstract answers is the easiest way to use abstract answers as lemmas. However, the performance may suffer from the creation of too many lemmas that eventually force a sequential search. This performance problem can be alleviated by using indexing on a structural property. The basic idea is to use some simple

[6] Clearly, this is allowed because clause composition is associative.

measure defined on a structural property of a class of expressions (like the length of a list or the depth/path of a tree) as a key for a hashing-based indexing scheme or by mapping it directly to the underlying (first/multi)-argument indexing of the Prolog system. An example of this technique for `nrev/2` is given in Section 3.3.

It ensures constant time access to the only relevant lemma, and is enough to achieve $O(N)$ performances on the benchmark. Note that such size-related parameters can be found relatively easily as shown, for instance, by research on the automatic inference of norms for termination analysis [7]. However, this step cannot be automated in the general case. Alternatively, indexing methods like matching trees or decision graphs [12, 29] can be used if provided by the underlying Prolog implementation.

3. DELPHI LEMMAS

Delphi lemmas are intended to reduce the time and the space spent for relatively useless memoing. The basic idea is that, very often, lemmas are only useful for the hot spots of the execution trace of the program.

3.1. Straightforward Delphi Lemmas

Suppose we consult an oracle before each decision to generate a lemma. A very smart (say human) oracle can decide for the naive reverse program for a list of length 100 iterated 50 times that the only lemma that is really worth generating is

```
nrev([A1,...,100], [A100,...,A1]).
```

This ensures a hit for each call and no search. How can we get close to this automatically? A surprisingly simple answer is to use a random oracle with a sufficiently low probability of answering `yes` to the question

Should this goal create a lemma after finishing its computation?

This means that the probability of generating lemmas will only be high for the "hot spots" of the program. However, a programmer's hint on which predicates are subject to Delphi memoing (quadratic `nrev/2` in this case instead of the already linear `append/3`) can definitely help. As we will show in the next section, the key property that makes Delphi lemmas effective is the presence of *attractors*: entry points to spots in the program where long sequences of "regular" repetitive computations start. In practice, the choice of the *predicates* and *probabilities* will give the programmer enough control to empirically fine-tune lemma generation and focus the action of the Delphi principle on the entry points of the actual hot spots of the program.

Here is the code (SICStus Prolog 2.1):

```
nrev_lemma(Xs,Ys,Xs1,Ys1):-
    nrev_fact(Xs,Ys,Xs1,Ys1), !.
nrev_lemma(Xs,Ys,Xs1,Ys1):-
    nrev(Xs,Ys,Xs1,Ys1),
    make_nrev_lemma(Xs1,Ys1).
make_nrev_lemma(Xs1,Ys1):-random(X), X>0.04,!.
make_nrev_lemma(Xs1,Ys1);-
    copy_term(Xs+Ys1,Xs2+Ys2),
    assert(nrev_fact(Xs1,Ys1,Xs2,Ys2)).
```

```
app([],Ys,Ys,[],Ys1,Ys1).
app([A|Xs],Ys,[A|Zs],[A1|Xs1],Ys1,[A1|Zs1]):-
   app(Xs,Ys,Zs,Xs1,Ys1,Zs1).
nrev([],[],[],[]).
nrev([X|Xs],R,[X1|Xs1],R1):-
  enrev_lemma(Xs,T,Xs1,T1),
app(T,[X],R,T1,[X1],R1).
nrev(Xs,Ys):-
   nrev(Xs,"Ys,_Xs1,_Ys1).
```

Note that this example exhibits a combination of the Delphi principle and abstract answers. `copy_term` is used to standardize apart the component which computes abstract answers. For length 2, the `nrev_fact/4` to be memoized looks like

```
nrev_fact([A0,A1],[A1,A0],[A2,A3],[A3,A2]).
```

When the lemma is used, the first two arguments will reverse the actual goal-dependent list, while the last two arguments will perform a resolution step which incrementally computes the next abstract answer.

Note also that `random/1` (see `library(random)` in SICStus 2.x) generates a uniformly distributed float value between 0 and 1.[7] Clearly, this transformation is correct only for deterministic programs. The cut is needed to choose between using a lemma and doing a computation.

The execution speed of the `nrev/2` programs using Delphi lemmas w.r.t. the probability is depicted in Figure 1. Although LIPS are inappropriate as they count inferences we actually spare, we keep using them because, as a popular metric for `nrev/2`, they give a familiar intuition about the amount of performance increase. There is a clear performance peak for small Delphi probabilities, followed by a gradual performance decrease for increasing Delphi probabilities. As we do not use indexing to search for lemmas, the cost of finding a lemma is proportional to the number of lemmas in the database. As can be seen from Figure 2, the number of lemmas rapidly increases with the Delphi probability. That is why the execution speed decreases nearly linearly.

[7]This has been speeded up in the BinProlog 5.00 built-in Delphi lemma preprocessor by transforming it to equivalent but faster integer-only arithmetics.

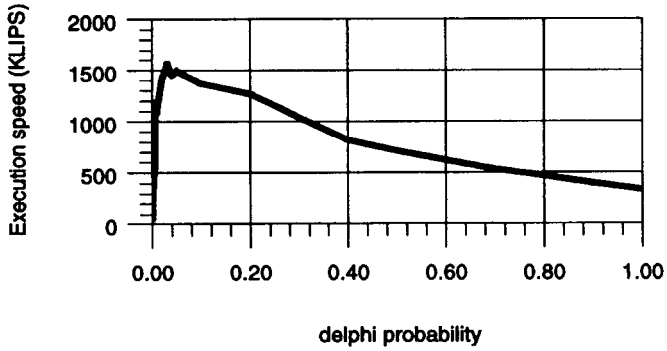

FIGURE 1. Execution speed w.r.t. the Delphi probability for `nrev/2`, length = 100.

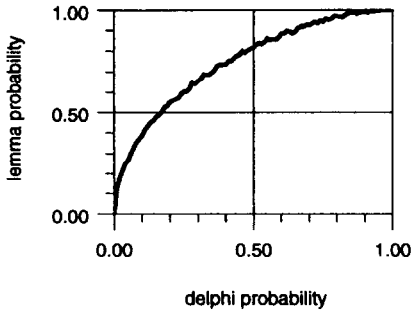

FIGURE 2. Lemma probability versus Delphi probability.

Note that obtaining a variant with Delphi lemmas from a program P is a fully automatic operation after inserting a Delphi declaration in the program

```
:-delphi nrev/2-4.
```

specifying the predicates to be memoized and the probability (4/100) of lemma generation for each of them.

3.2. *Optimal Delphi Probability*

Figure 2 shows the probability that a particular lemma is memoized w.r.t. the Delphi probability for repeated executions of `nrev/2` with list length = 100. It turns out that the number of lemmas generated rapidly increases from small Delphi values, e.g., for Delphi = 0.1, 40% of all possible lemmas is already generated.

Given the relationship of Figure 2, we can find the optimal value for the Delphi probability for `nrev/2` and for a given list length. We assume that there is no lemma indexing such that adding one extra lemma increases the cost of using a lemma. For list lengths varying between 0 and 100, the optimal Delphi probability turns out to lie between 0 and 0.08 as depicted in Figure 3.

3.3. *Indexed Delphi Lemmas*

By combining Delphi lemmas with length indexing, we can see a cumulative positive speed-up. Computing the length can be partially evaluated away as a small

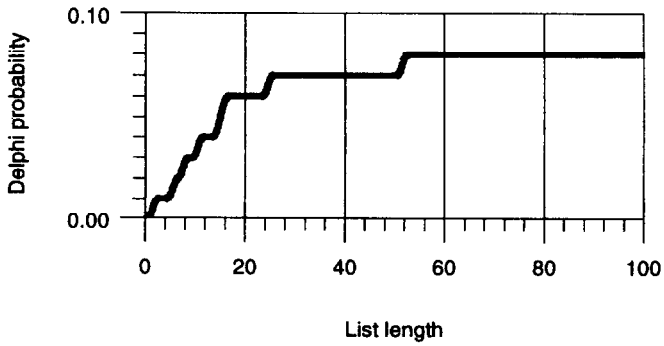

FIGURE 3. Optimal Delphi probability versus list length in `nrev/2`.

extra effort for the host predicate. Here the code obtained for `nrev/2`:

```
nrev_lemma(Xs,Ys,Xs1,Ys1,L):-
   nrev_fact(L,Xs,Ys,Xs1,Ys1), !.
nrev_lemma(Xs,Ys,Xs1,Ys1,L):-
   nrev(Xs,Ys,Xs1,Ys1,L),
   make_nrev_lemma(L,Xs1,Ys1).
make_nrev_lemma(L,Xs1,Ys1):-random(X), X>0.04,!.
make_nrev_lemma(L,Xs1,Ys1):-
   copy_term(Xs1+Ys1,Xs2+Ys2),
   assert(nrev_fact(L,Xs1,Ys1,Xs2,Ys2)).
app([],Ys,Ys,[],Ys1,Ys1).
app([A|Xs],Ys,[A|Zs],[A1|Xs1],Ys1,[A1|Zs1]):-
   app(Xs,Ys,Zs,Xs1,Ys1,Zs1).
nrev([],[],[],[],0).
nrev([X|Xs],R,[X1|Xs1],R1,N):-N1 is N-1,
   nrev_lemma(Xs,T,Xs1,T1,N1),
   app(T,[X],R,T1,[X1],R1).
nrev(Xs,Ys):-
   length(Xs,L),
   nrev(Xs,Ys,_Xs1,_Ys1,L).
```

Note that by using a *generic* `assert` operation in the implementation instead of `asserta` or `assertz`, we can count on the implementation to use a more sophisticated indexing mechanism which may not be *order preserving* (for instance, some form of hashing using structural information on ground/nonground terms).

3.4. *Performance of Various Memoing Techniques*

The execution speeds for the various lemma generation techniques are given in Table 2 normalized from 200 executions on random lists of length 100 with probability 0.04 of Delphi-lemma generation. Although LIPS have no meaning in terms of counting logical inferences (mostly avoided by using the lemmas), we have kept them simply to express the speed-up in familiar terms.

TABLE 2. Execution speeds for the abstract lemma techniques on a Sparcstation ELC with SICStus Prolog 2.1 compact code

Type of lemmas	Speed (KLIPS)	Speed-Up
Original `nrev`	145	1.0
Straightforward abstract	358	2.5
Length-indexed	1272	8.8
Delphi	1515	10.4
Indexed Delphi	1689	11.6
Delphi oracle static code	2525	17.4
Human oracle static code	11841	81.7

The last two lines are obtained by writing the lemmas to a file and then adding them to the program as static code. They give an idea of the maximum speed-up that can be obtained. This is, in a sense, comparable to profile-based optimizations that are common in contemporary state-of-the art optimizing compilers.

3.5. *Distribution of the Generated Delphi Lemmas*

It turns out that the Delphi algorithm tends to generate the most efficient lemmas, i.e., the more "complex" ones in our case, with a very high probability. The explanation is basically that a lemma of complexity $\mathscr{C}$ ensures that no simpler lemmas are generated, while it cannot prevent the generation of more complex ones.

Applied to the case of `nrev/2`, this means that whenever we have a lemma for a list of length `N`, a goal with list length greater than `N` will not generate lemmas with list lengths smaller than `N`. Hence, after a sufficiently large number of runs, the lemma for the initial goal will be generated. This effect can be experimentally observed, and can also be theoretically modeled.

The experimental distribution is compared with the theoretical distribution $p(n)$ that denotes the probability of having lemma n $(0 < n < M)$ in Figure 4. To compute $p(n)$, we start by defining an auxiliary distribution $p_i(n)$ denoting the probability of having lemma n after the ith run. It is expressed as being the sum of the probability of having a lemma in the previous run plus the probability of generating a lemma in the current run. The latter probability is given by p times the probability that no lemma was hit while recursing down to n. Initially, $p_0(n) = p$, e.g., all goals have the same probability of being memoized. The distribution $p(n)$ is the limit for i going to ∞:

$$p_0(n) = p$$

$$p_i(n) = p_{i-1}(n) + p \prod_{j=n}^{M} (1 - p_{i-1}(j))$$

$$p(n) = \lim_{i \to \infty} p_i(n).$$

The empirical and theoretical distributions are given in Figure 4. The darker curve is the empirical distribution.

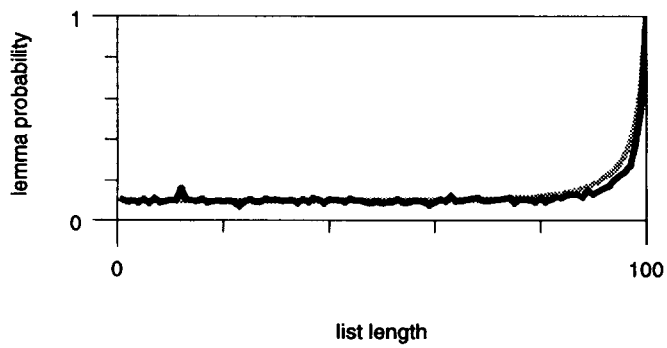

FIGURE 4. Comparison between theoretical and empirical distribution of lemmas.

3.6. *Delphi Lemmas for Predicates with Irregular Behavior*

The Delphi principle succeeds in finding the hot spots in the very regular naive reverse benchmark. We have also applied the Delphi principle to an arithmetic function known to have a highly irregular behavior. The timings show that Delphi lemmas are superior to full memoing (more than 50% faster), which in turn is superior to no memoing. The function we consider is

$$f(i) = \begin{cases} true, & \text{if } i = 1 \\ f(i/2), & \text{if } i \text{ is even} \\ f(3i+1), & \text{otherwise.} \end{cases}$$

The program after the Delphi transformation becomes

```
f_lemma(N) :-f_fact(N), !.
f_lemma(N) :- f_real(N).
f_real(1).
f_real(N) :-N mod 2 =:= 0, N1 is N//2, f(N1),
make-f-lemma(N).
f_real(N) :- N1 is 3*N+1, f(N1), make_f_lemma(N).
make_f_lemma(N) :- random (X), X>0.04, !.
make_f_lemma (N) :- asserta(f_fact(N)).
f(N) :- F_lemma(N).
```

Two experiments have been done with this function. Calling `f(100)` repeatedly, gives rise to the lemma distribution of Figure 5. This behavior is completely similar to the behavior of `nrev/2` in Figure 4 after reordering the recursive calls in the order in which they occur (100, 50, 26, 76, 38, 19, ...). The most efficient lemma is `f(100)`, and it is eventually generated with probability 1.0.

Figure 6 gives a different view. Now, `f` is evaluated for 100 different values `(f(1)...f(100))`. Due to the rather irregular behavior of the function, the distribution seems to be strange, but even in this case, the Delphi principle succeeds in finding the hot spots, namely, the function calls with small arguments. Indeed, whenever the function is defined, it will eventually enter a decreasing sequence of powers of 2, ending in `f(1)`.

The conclusion is that, even for very irregular predicates, the Delphi principle succeeds in detecting and memoing the hot spots in the program. Note that despite their irregular behavior, most programs exhibit *attractors* (the sequence 1, 2, 4, 8, ...

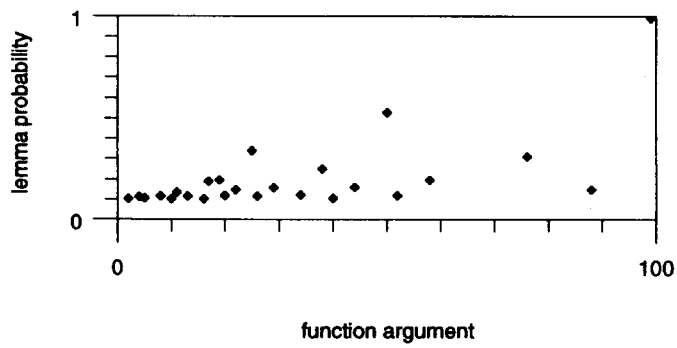

FIGURE 5. Distribution of lemmas for repeated execution of `f(100)`.

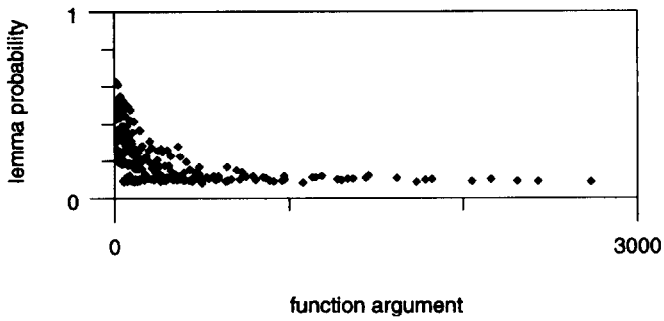

FIGURE 6. Distribution of lemmas for repeated execution of `f(1)...f(100)`.

in this example), i.e., a set of data objects which occur in most computations.[8] Delphi lemmas naturally tend to find these attractors and to speed up their computation by memoing them. An interesting feature of Delphi lemmas is that the programmer does not have to search for attractors. If they are present, the Delphi principle will find them automatically. Even if there is more than one attractor, the Delphi principle we find them all.

4. BENCHMARKS

Delphi lemmas have been added as a standard extension of BinProlog 5.00 [23] with a declaration-based preprocessor. At this time, only memoing of deterministic predicates with indexing on an arbitrary number of arguments of the top-level functor is supported. However, their impact on some well-known benchmarks is quite dramatic.

A directive like `:-delphi tak/4-10/[1,2,3]` (used for the tak benchmark) is interpreted as follows: use Delphi memoing with probability 10/100 while indexing by hashing on the first three arguments. A directive like `:-delphi rewrite/2-20` (used for the boyer benchmark) is interpreted as follows: use Delphi memoing with (default) indexing on first argument. In this case, in the actual benchmark, each call of the form `rewrite(A,B)` is replaced with `delphi_call_1(rewrite(A,B),20,A)`. This operation is done at compile time. The generic `delphi_call_1` predicate looks as follows:

```
delph_call_1(P,_X):-membq(X,P,P),!. % already memoed
delphi_call_1(P,Delphi,X):-
  random(R),Luck is R mod 100,Delphi>Luck,!,
  P,!,
  addq(X,P,P).                        % memoing now
delphi_call_1(P,_,_):-P,!.            % call as usual
```

[8] Intuitively, an attractor is a value such that once a specific argument of a literal has this (input) value, termination is ensured, and moreover, (many) terminating derivations will eventually hit this value. The value that matches the base case of a recursive predicate is the attractor *par excellence*. In the more complex situation of our arithmetic example, the values 2^n are attractors because they show up more in derivations than, say, 17. That is why in Delphi memoing, the facts for 2^n are memoed more likely. Since programs usually terminate in a way that is less dependent on their data than on their structure, it is reasonable to believe that most terminating programs indeed exhibit attractors.

Benchmark	No memoing	Delphi 5/100	Delphi 20/100	Full memoing
Fibo	2660	20	10	10
Tak	1130	390	180	90
Boyer	4660	880	480	420

FIGURE 7. Delphi memoing (user time in ms).

Note that `membq/3` and `addq/3` both use BinProlog's efficient blackboard [22, 23] with hasing on its first two arguments and the memoed value in the third. Further speed-up can be obtained by specializing this generic predicate to each of its uses, known at compile time, i.e., to something like `delphi_call_1_rewrite(A,B)` instead of `delphi_call_1(rewrite(A,B),20,A)`. Even without this optimization, performance turned out to be unusually good, as shown in Figures 7 and 8.

We have executed the benchmarks on a one-user Sparcstation 10-20 with 32 Mbytes of memory. Delphi results are geometric means of ten different runs. The use of lemmas is optimized by avoiding `assert`. As the generated lemmas are known to be ground, BinProlog's blackboard can and does avoid the usual copying that happens if `assert` is used for memoing.[9]

This also explains why on second runs, all benchmarks give execution times under 100 ms. The sharing of ground lemmas also contributes to the dramatic decrease of heap consumption due to memoing. We have noticed that our lemmas are particularly effective on binarization-based compilers like BinProlog which have large heap consumption and AND-intensive functional-style programs like `boyer` or `tak`. Good performance on `boyer` is explained by the fact that on a total of 95,016 calls to the memoized predicate `rewrite/2`, only 200 are different. The memoized calls function as *attractors* which avoid iterated uses of `rewrite/2`. The relatively small number of attractors for predicates like `tak/4` or `fibo/2` explains the speed-up on these two benchmarks.

Although on these benchmarks full memoing is still faster than Delphi memoing (due to the overhead of the source-level implementation of the random oracle), the ability of fine-tune the space–time tradeoff is in itself enough to make them useful in practice.

Note that the techniques are general purpose, and we have measured similar speed-ups for a modified version of ProLog-by-BIM, a native code compiler which uses a traditional WAM model. Moreover, in that case, we have observed that, e.g., `20%` Delphi-memoing is 50% faster than full memoing, due to the relatively lower computation/memoing ratio of ProLog-by-BIM.

[9]Direct unification with ground terms is safe even on failure as it will always generate "well-directed" bindings (from heap variables to the blackboard).

Benchmark	No memoing	Delphi 5/100	Delphi 20/100	Full memoing
Fibo	51132+0	96+360	96+520	96+680
Tak	1017852+0	287992+5272	128384+7336	68660+13528
Boyer	4774920+0	244112+9344	113932+15404	77092+21888

FIGURE 8. Delphi memoing (space used: heap + blackboard).

5. CONCLUSION AND FUTURE DIRECTIONS

The techniques developed in this paper are partially motivated by current Prolog systems, with their particular strengths and limitations (e.g., their built-ins and indexing techniques).

We have studied the case of a well-known definite deterministic program as being the simplest instance of this memoing technique. This case covers, however, the complete class of committed choice logic programming languages [5, 10]. Delphi lemmas, as shown in the previous examples, have the same domain of applicability and limitations as ordinary lemmas. Note, however, that the Delphi principle is itself orthogonal to the type of program to which memoing is applied. With some care, they can be used in the context of full Prolog, specified as we have suggested by a programmer-controlled directive. The practicality of the special case of ground, multiargument-indexed Delphi lemmas has convinced us to add them as a standard extension to the BinProlog compiler, starting from version 5.00 (available by ftp from clement.info.umoncton.ca).

Delphi lemmas have also served as inspiration for BinProlog 5.00's dynamic recompilation scheme, which moves "hot spots" of nonvolatile interpreted code to compiled code on the fly, based on *update* versus *call* statistics [23].

Indexing in the case of nonground answers in general can improve potential performance problems with abstract answers. The techniques described in [18] as well as the idea of using tries instead of hashing, as done in the latest version of XSB, can be adapted for this purpose.

The program transformations we used can be modified in the case of nondeterministic programs to avoid interference between lemma-related pruning and the program's control structure. More work is needed on how this can be done in general, and on what are the limitations of tradeoffs. The latest version of the XSB system successfully deals with SLG resolution, which is fairly realistic subset of Prolog including nondeterminism and a restricted form of negation as failure. We plan to apply some of the techniques of XSB (like memoing of multiple answer substitutions) to extend Delphi lemmas beyond definite programs.

Nevertheless, we consider our techniques useful for "real" programmers as it can accelerate the most critical parts of a program in a significant way, and also because they can be fully automated. Delphi lemmas can be also complement existing systems with memoing facilities having objectives that are orthogonal to ours, as [28]. Future work is planned to port Delphi lemmas to the latest version of XSB Prolog.

We do not know about something similar to Delphi lemmas in functional or procedural languages, but the concept can be easily adapted. Although it is a probabilistic concept in the same sense as hashing or Ethernet collision avoidance, it gives very good performances for mostly the same reasons, especially in combination with the use of abstract answers. As our benchmark data on *boyer* show, the technique looks particularly beneficial for lemma-intensive theorem-proving systems.

Future work will focus on the design of a set of high-level pragmas for specifying memoization and on safe conditions when lemma generation can be automated. This direction involves both static analysis techniques and trace analysis. This paper has showed the empirical utility of considering memoing as a possibly expensive computational resource, not only as an obvious space-for-time trade-in. More theoretical work, possibly on a subset of linear logic, is needed to formalize resource-conscious memoing techniques in a uniform framework.

REFERENCES

1. Apt, K., Logic Programming, in: J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Elsevier, North-Holland, 1990, Vol. B, pp. 493–574.
2. Benkerimi, K. and Lloyd, J. W., A Partial Evaluation Procedure for Logic Programs, in: [6], pp. 343–358.
3. Bruynooghe, M. (ed.), *Logic Programming—Proc. 1994 Int. Symp.*, Massachusetts Institute of Technology, MIT Press, 1994.
4. Cheng, M. H. M., van Emden, M. H., and Strooper, P. A., Complete Sets of Frontiers in Logic-Based Program Transformation, in: J. W. Lloyd (ed.), *Proc. Workshop on Meta-Programming in Logic Programming*, MIT Press Series in Logic Programming, Cambridge, MA, London, England, MIT Press, 1988, pp. 213–225.
5. De Bosschere, K., DeBray, S., Gudeman, D., and Kannan, S., Call Forwarding: A Simple Interprocedural Optimization Technique for Dynamically Typed Languages, in: *Proc. 21st ACM SIGPLAN–SIGACT Symp. on Principles of Programming Languages (POPL)*, Portland, Jan. 1994, pp. 409–420.
6. Debray, S. and Hermenegildo, M. (eds.), *Proc. 1990 North American Conf. on Logic Programming*, Cambridge, MA, London, England, MIT Press, 1990.
7. Decorte, S., DeSchreye, D., and Fabris, M., Automatic Inference of Norms: A Missing Link in Automatic Termination Analysis, in: [15], pp. 420–436.
8. Falaschi, M., Levi, G., Martelli, M., and Palamidessi, C., A New Declarative Semantics for Logic Languages, in: R. A. Kowalsksi and K. A. Bowen (eds.), *Proc. 5th Int. Conf. and Symp. on Logic Programming*, MIT Press, 1988, pp. 993–1005.
9. Gallagher, J. and Bruynooghe, M., The Derivation of an Algorithm for Program Specialisation, *New Generation Computing* 9 (1991).
10. Gudeman, D., De Bosschere, K., and Debray, S., `jc`: An Efficient and Portable Sequential Implementation of Janus, in: K. Apt (ed.), *Joint Int. Conf. and Symp. on Logic Programming*, Washington, DC, MIT Press, Nov. 1992, pp. 399–413.
11. Kemp, R. S. and Ringwood, G. A., An Algebraic Framework for Abstract Interpretation of Definite Programs, in: [6], pp. 506–530.
12. Kliger, S. and Shapiro, E., From Decision Trees to Decision Graphs, in: [6], pp. 97–116.
13. Lloyd, J., *Foundations of Logic Programming*, 2nd edition (Symbolic Computation—Artificial Intelligence), Springer-Verlag, Berlin, 1987.
14. Lloyd, J. W. and Shepherdson, J. C., Partial Evaluation in Logic Programming, *JLP91* 11(3/4) (1991).
15. Miller, D. (ed.), *Logic Programming—Proc. 1993 Int. Symp.*, Vancouver, Canada, MIT Press, 1993.
16. Ramakrishnan, I. V., Rao, P., Sagonas, K. F., Swift, T., and Warren, D. S., Efficient Tabling Mechanisms for Logic Programs, in: L. Sterling (ed.), *Logic Programming—Proc. 12th Int. Conf. on Logic Programming*, Massachusetts Institute of Technology, MIT Press, 1995, pp. 697–711.
17. Sterling, L. and Shapiro, E., *The Art of Prolog*, MIT Press, 1986.
18. Sudarshan, S. and Ramakrishnan, R., Optimizations of Bottom-Up Evaluation with Non-Ground Terms, in: [15], pp. 557–574.
19. Swift, T. and Warren, D. S., An Abstract Machine for SLG Resolution: Definite Programs, in: [3], pp. 633–652.
20. Swift, T. and Warren, D. S., Analysis of SLG-WAM Evaluation of Definite Programs, in: [3], pp. 219–235.
21. Tamaki, H. and Sato, T., OLD Resolution with Tabulation, in: E. Shapiro (ed.), *Proc. 3rd Int. Conf. on Logic Programming*, Lecture Notes in Computer Science, London, Springer-Verlag, 1986, pp. 84–98.
22. Tarau, P., Language Issues and Programming Techniques in Bin Prolog, in: D. Sacca (ed.), *Proc. GULP'93 Conf.*, Gizzeria Lido, Italy, June 1993.

23. Tarau, P., *BinProlog 5.00 User Guide*, Technical Report 96-1, Département d'Informatique, Université de Moncton, Canada, Apr. 1996. Available from http://clement.info.umoncton.ca//BinProlog.
24. Tarau, P. and Boyer, M., Elementary Logic Programs, in: P. Deransart and J. Maluszyński (eds.), *Proc. Programming Language Implementation and Logic Programming*, no. 456 in Lecture Notes in Computer Science, Springer, Aug. 1990, pp. 159–173.
25. Tarau, P. and Boyer, M., Nonstandard Answers of Elementary Logic Programs, in: J. Jacquet (ed.), *Constructing Logic Programs*, Wiley, 1993, pp. 279–300.
26. Tarau, P. and De Bosschere, K., Memoing with Abstract Answers and Delphi Lemmas, in: Y. Deville (ed.), *Logic Program Synthesis and Transformation*, Louvain-la-Neuve, Springer-Verlag, Workshops in Computing, July 1993, pp. 196–209.
27. Warren, D. S., Memoing for Logic Programming, *Commun. ACM* 35(3):37–48 (1992).
28. Warren, D. S., The XOLDT System, Technical Report, SUNY Stony Brook, NY, electronic document: ftp sbcs.sunysb.edu, 1992.
29. Zhou, N.-F., Takagi, T., and Ushijima, K., A Matching Tree Oriented Abstract Machine for Prolog, in: D. H. D. Warren and P. Szeredi (eds.), *Proc. 7th Int. Conf. on Logic Programming*, Cambridge, MA, London, England, MIT Press, 1990, pp. 159–173.