Memoing with Abstract Answers and Delphi Lemmas

Paul Tarau
Département d'Informatique
Université de Moncton
Moncton, Canada, E1A 3E9
tarau@info.umoncton.ca

Koen De Bosschere
Vakgroep Elektronica en Informatiesystemen
Universiteit Gent
B-9000 Gent, Belgium
kdb@elis.rug.ac.be

April 8, 1993

Abstract

Abstract answers are most general computed answers of logic programs. For each derivation leading to a computed answer there is a unique abstract answer obtained by replicating the steps of the derivation, except the first one. After describing a meta-interpreter returning abstract answers we derive a class of program transformations that compute them more efficiently. Abstract answers are ideal lemmas as their 'hit rate' is much higher than in the case of naive memoing. Experimental evidence to this claim is presented by showing an order of magnitude speed-up for the naive reverse benchmark through an automatic program transformation using abstract answers as lemmas. To ensure tight control on the amount of memoing a simple but very effective technique is introduced: Delphi lemmas are abstract answers 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.

Keywords: abstract and conditional answers, program transformations, memoing, lemmas

1 Introduction

Memoing techniques have been an important research topic in logic programming and deductive databases (see [War92a], [TS86]). Various practical tools for memoing exist from programmer defined assert based mechanisms (see [SS86]) and extension tables as in SB-Prolog to dedicated interpreters like [War92b]. However, they all share the following basic ideas:

- atoms obtained in the course of resolution are memoized as such;

- there is no mechanism to improve the 'hit rate' of the lemmas;
- a fixed memoing algorithm is used;
- memoing is seen as a tool but not as a resource.

As a practical consequence, the existing memoing facilities are not 'smarter than programmers' who therefore tend to code their lemmas directly with `assert` and `retract` or by carrying around clumsy and difficult to index data structures. This is especially true with theorem provers that suffer in terms of performances and readability from the absence of a practical higher level memoing mechanism provided by the underlying Prolog system.

This paper tries to solve these problems in a simple but radical way. First instead of memoing actual instances of answers created during the resolution process we will replace them with their more general instances, such that while preserving soundness we can ensure the best possible future reuse. We show that the overhead can be compiled away by using program transformations. We consider memoing as a possibly expensive resource that is not a priori better than recomputation. To ensure tight control on memoing, we abolish the 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 will be surprisingly better with a random oracle[1] than with naive fixed algorithm memoing approaches, especially when we have the possibility to control the amount of copying involved in the memoing process.

2 Resolution revisited

Although classical texts on SLD and SLDNF resolution (see [Llo87] and [Apt90]) do explain well the basic logical mechanisms behind Prolog engines one aspect is neglected in all but some partial evaluation oriented papers like [LS91]: the resolvent is seen as an 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 the tautologically true clause `G:-G`. Even worse, resolution is seen as a 'refutation' procedure although everyone uses it intuitively as a constructive entailment process where we simply unfold clauses of a program until a fact `A:-true` is eventually reached[2].

Without going into the details of such a reconstruction we now define the *composition* operator $\oplus$ that combines clauses by unfolding the leftmost body-goal of the first argument.

Definition 1 *Let* $\mathtt{A}_0\mathtt{:-A}_1\mathtt{,A}_2\mathtt{,\ldots,A}_n$ *and* $\mathtt{B}_0\mathtt{:-B}_1\mathtt{,\ldots,B}_m$ *be two clauses* ($n > 0, m \geq 0$). *We define*

$$(\mathtt{A}_0\mathtt{:-A}_1\mathtt{,A}_2\mathtt{,\ldots,A}_n) \oplus (\mathtt{B}_0\mathtt{:-B}_1\mathtt{,\ldots,B}_m) = (\mathtt{A}_0\mathtt{:-B}_1\mathtt{,\ldots,B}_m\mathtt{,A}_2\mathtt{,\ldots,A}_n)\theta$$

with $\theta = mgu(\mathtt{A}_1,\mathtt{B}_0)$. *If the atoms* $\mathtt{A}_1$ *and* $\mathtt{B}_0$ *do not unify, the result of the composition is denoted as* $\perp$. *Furthermore, as usual, we consider* $\mathtt{A}_0\mathtt{:-true,A}_2\mathtt{,\ldots,A}_n$ *to be equivalent to*

[1]This random oracle has however a 'tunable' probability.

[2]Redoing the theory with this supposition is easy but tedious, unless it is left as an exercise to the reader.

`A0:-A2,...,An`, *and for any clause* C, $\perp\oplus$ `C` $=$ `C` $\oplus\ \perp = \perp$. *We suppose (when necessary) that at least one operand has been renamed to a variant with fresh variables.*

Let us call this Prolog-like inference rule LF-SLD resolution (LF for 'left-first'). Remark that by working on the program P' obtained from P by replacing each clause with the set of clauses obtained by all possible permutations of atoms occurring in the clause's body every SLD-derivation on P can be mapped to an LF-SLD derivation on P'. Extension of this inference mechanism to SLDNF-resolution is easy.

3 Abstract and conditional answers

Definition 2 *Let* `G` *be an atomic definite goal and let us denote* `q(G)` *the clause* `G:-G`. *An LF-SLD derivation is a sequence of* clauses $C_1, \ldots, C_n$ *such that the result of their composition* $C_1 \oplus \ldots \oplus C_n$ *is different from* $\perp$.

Remark that clause composition is an associative operation, and thus it makes sense to remove the first step from a derivation `q(G)` and then compose a new `q(G')` with the rest of it.

Definition 3 *A derivation starting with* `q(G)` *for an atomic goal* `G` *and terminating with a* fact *of the form* `A:-true` *is called a* standard LF-SLD derivation. *A derivation starting with a clause of the program instead of* `q(G)` *is called an* abstract LF-SLD derivation *and a derivation not terminated by a fact is called a* conditional LF-SLD derivation. *A (standard,abstract,conditional) answer is the result of the composition of the clauses occurring in a (standard,abstract,conditional) LF-SLD derivation. By combining them we can talk about* abstract conditional answers.

Abstract answers are composed exclusively from clauses of the program P, that 'shadow' more specific standard answers obtained starting from `q(G)` and using it as a 'path-finder' for a derivation. Conditional answers are very useful in partial evaluation[3] (see [MHMC88]), type inference (see [vE88]) and in expressing partial computations similar to those occurring in higher order functional programs (see [CvER90]).

Example 1 Let us consider the program:

```
(C1)    plus(0,Y,Y):-true.
(C2)    plus(s(X),Y,s(Z)):-plus(X,Y,Z).
```

and `q(G)` =

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

We obtain, as the result of `q(G)`$\oplus C_2$, the conditional answer

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

[3]Introduced in logic programming by [Kom82]; also known as partial deduction.

Then, by composing with (C_1), the expression `q(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 ($C_2 \oplus C_1$), 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`'. ◇

Remark that working with clause compositions ensures that each step of a derivation corresponds to a logical consequence of the program. This incremental logical soundness contrasts with standard SLD-derivation where only the (final) computed answer has this property.

Theorem 4 *If S is a (standard,abstract,conditional) answer of P, then S is a logical consequence of P.*

Proof This theorem as a consequence of more general results of [MHMC88], [KR90] or [BL90]. A short direct proof in the case of binary logic programs is given in [Tar90]. ■

In the examples that follow, we often use `S` instead of `S:-true` for efficiency and readability.

4 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 systems like Quintus, ALS or C-Prolog: `clause/3` which returns a database reference `Ref` and `instance/2` which returns a new copy of the clause referred to by `Ref`.

Example 2 If P =

```
app([],Ys,Ys).
app([A|Xs],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([1,2,3],X),demo(G,Abstract).
```

the following abstract answer

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

◇

5 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.

Definition 5 *Let $ABS(\phi, P)$ be the program obtained by replacing each clause* $\texttt{A}_0\texttt{:-A}_1\texttt{,...,A}_n$ *of the program P by* $\phi(\texttt{A}_0\texttt{,A}_0^{'})\texttt{:-}\phi(\texttt{A}_1\texttt{,A}_1^{'})\texttt{,...,}\ \phi(\texttt{A}_n\texttt{,A}_n^{'})$*, where* $\texttt{A}_0^{'}\texttt{:-A}_1^{'}\texttt{,...,A}_n^{'}$ *is a fresh copy of* $\texttt{A}_0\texttt{:-A}_1\texttt{,...,A}_n$*.*

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

5.1 Naive

We define $\phi(\texttt{a(X}_1\texttt{,...,X}_n\texttt{),a(Y}_1\texttt{,...,Y}_n\texttt{)})$ as $\texttt{naive(a(X}_1\texttt{,...,X}_n\texttt{),a(Y}_1\texttt{,...,Y}_n\texttt{))}$, where `naive/2` is a knew predicate symbol.

Example 3 $ABS(naive, P)$ =

```
naive(append([],Ys,Ys),append([],Ys1,Ys1)).
naive(append([A|Xs],Ys,[A|Zs]),append([A1|Xs1],Ys1,[A1|Zs1])):-
  naive(append(Xs,Ys,Zs),append(Xs1,Ys1,Zs1)).

naive(nrev([],[]),nrev([],[])).
naive(nrev([X|Xs],R),nrev([X1|Xs1],R1)):-
  naive(nrev(Xs,T),nrev(Xs1,T1)),
  naive(append(T,[X],R),append(T1,[X1],R1)).
```

The query

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

returns the abstract answer:

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

◇

5.2 Indexed

We can define ϕ(a(X$_1$,...,X$_n$),a(Y$_1$,...,Y$_n$)) as a(X$_1$,...,X$_n$,a(Y$_1$,...,Y$_n$)), hereby preserving the indexing on the first argument of the original program.

Example 4 *ABS(index, 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]).
```

◇

5.3 Flat

We can avoid the construction of useless structures by defining ϕ(a(X$_1$,...,X$_n$),a(Y$_1$,...,Y$_n$)) as a(X$_1$,...,X$_n$,Y$_1$,...,Y$_n$).

Example 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]).
```

◇

Remark that the three transformations that we have used, compute abstract answers together with standard answers within a constant factor from computing only the standard answers for P. Moreover, it is not difficult to prove that the transformed program generates precisely the same standard answers as the original program P while computing the abstract answers too. This proves that there is an abstract answer for every standard answer. Conversely, every abstract answer is obviously a standard answer.

The execution speeds are shown in table 1. It follows that the program transformation is much faster than the meta-interpreter and that the overhead of the computation of an abstract answer is limited to 50 % w.r.t. direct execution of naive reverse.

version	runtime (μs)	relative time
original nrev	105	0.64
demo	3174	19.24
naive	374	2.26
index	208	1.26
flat	165	1.00

Table 1: Execution speeds for the program transformations on a Sparcstation ELC with Sicstus Prolog 2.1 compact code.

6 Abstract answers as lemmas

Abstract answers are good candidates for reusable lemmas. By accumulating an abstract answer `S:-true`, the equivalent of the search for an entire LF-SLD-refutation can be replaced with the composition of the clause `q(G)` for a given atomic goal `G` and a memoized abstract answer `S:-true`. In the same way, a partial computation can be concentrated as an abstract conditional answer.

Remark the non-trivial nature of lemmas obtained from abstract or abstract conditional answers as they replace possibly infinite sets of standard answers.

Compared to memoing of standard answers, memoing of abstract or abstract conditional answers is more appropriate as the generality of the saved computation allows a better hit rate. Moreover, 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 it is also the case with usual lemma generation, the actual gain in efficiency depends on indexing of dynamic code and programmer defined ‘pragmas’ specifying what is worth to be memoized.

On the practical side, suppose we want count the number of LIPS using a large number of iterations of the well known `nrev/2` predicate. By using random values in the list to reverse we can prevent a clever Prolog system from using the memoing of something like `nrev([12,13,1,5,9],[9,5,1,13,12])` as an answer to `nrev([1,2,3,4,5],X)`. By using memoing of the abstract answer `nrev([A,B,C,D,E],[E,D,C,B,A])`, the only way to ensure fairness of the benchmark is to have lists of random content and random length at each iteration. We can go even further. As `nrev/2` is a recursive predicate we can use as lemmas `nrev([`A_1`,...,`A_n`],[`A_n`,...,`A_1`])` for each n. Obviously, this makes `nrev/2` linear[4] with respect to n and practically invalidates its use as a Prolog benchmark.

6.1 Naive abstract lemmas

Let P be the `nrev/2` program. Starting from $ABS(\phi, P)$, we can write a Prolog program that asserts as lemmas the abstract answers computed by $ABS(\phi, P)$ and that uses them to speed up the computation of the answers. We will compare the performances with the original `nrev/2` program.

This is the easiest way to use abstract answers as lemmas. However performances suffer due to creation of lemmas that eventually force a sequential search. Empirical tests confirm however a small speed-up with naive abstract lemmas, at least in the case of the nrev/2 predicate.

6.2 Length-indexed abstract lemmas

An easy way to overcome this is by using the a parameter that uniquely determines the appropriate lemma as a key. In the case of a program like naive reverse this is the length of the list. This ensures constant time access to the only relevant lemma and is enough to achieve $O(N)$ performances on the benchmark. Remark that such size-related parameters can be found relatively easily. However this step cannot be automated in the general case. On the other hand, indexing methods like switching trees or switching graphs can be used ‘automatically’ if provided by the underlying Prolog implementation.

6.3 Delphi lemmas

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

Suppose we consult an oracle before each decision to generating 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 to be generated is

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

[4]Constant in terms of LIPS but linear in terms of real work due to the unification of two lists of size n

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`. This means that the probability of generating lemmas will only be high for the hot spots of the program. In practice, the probability should be a parameter allowing the programmer to empirically fine-tune lemma generation. Here is the code (Sicstus Prolog 2.1):

```
:- ensure_loaded(library(random)).

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(Xs1+Ys1,Xs2+Ys2),
   asserta(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):-
   nrev_lemma(Xs,T,Xs1,T1),
   app(T,[X],R,T1,[X1],R1).

nrev(Xs,Ys=nrev(Xs1,Ys1)):-
   nrev(Xs,Ys,Xs1,Ys1).
```

Remark that this code can be obtained mechanically from the original nrev/2 program by simply specifying which predicates have to be memoized.

The execution speed of the nrev/2 programs using Delphi lemmas w.r.t. the probability is depicted in figure 1.

Remark that obtaining a variant with Delphi lemmas from a program P is a fully automatic operation if for instance the programmer declares something like:

```
:- delphi(nrev/2, 0.04).
```

specifying the probability of lemma generation for `nrev/2`.

6.4 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 extra effort for the

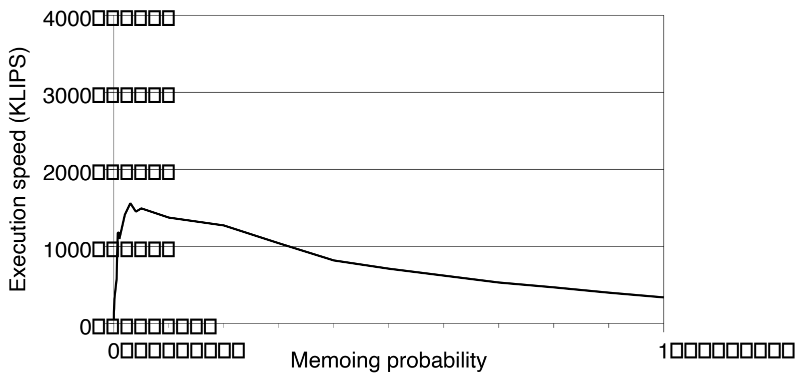

Figure 1: Execution speed of the Delphi lemmas w.r.t. the probability.

host predicate. Here is the code obtained for nrev/2.

```
:- ensure_loaded(library(random)).

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),
   asserta(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=L+nrev(Xs1,Ys1)):-
   length(Xs,L),
   nrev(Xs,Ys,Xs1,Ys1,L).
```

6.5 Non-copying and lazy-copying techniques

We have also tried to combine Delphi lemmas with BinProlog 1.71's non-copying and lazy-copying blackboard primitives. In BinProlog[5] the programmer has a tighter control on on the copying to the global data area called blackboard.

In the case of `nrev/2`, lemmas are of type 'memoize once, use N times' and are therefore not 'disposable'. However, in the case of more volatile data it is interesting to have lemmas which are used only once. Linear-logic based logic programming languages have the ability to specify this behaviour explicitly.

Remark that in the absence of backtracking, copying can be avoided if the lemmas are either ground or disposable. In the presence of backtracking, these lemmas must be copied (once) from the heap to the permanent data area (blackboard). Even in this case, structures already on the blackboard are reused by BinProlog's smart copying algorithm.

Hence, in the case of a program that does not backtrack it makes sense to represent lemmas directly as named heap-based data-structures if they are either 'ground' or 'disposable'. This is done easily in BinProlog which replaces `assert` and `retract` by a set of efficient hashing-based 'naming' primitives and a separate set of tunable 'copying' primitives (see [Tar93]).

6.6 Performance results

The execution speeds for the various lemma generation strategies and 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.

type of lemmas	speed (KLIPS)	speed-up
original `nrev`	145	1.0
naive 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

Table 2: Execution speeds for the abstract lemma techniques on a Sparcstation ELC with Sicstus Prolog 2.1 compact code.

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.

[5] a program transformation based continuation passing Prolog compiler available by ftp from `clement.info.umoncton.ca`.

6.7 Distribution of the generated Delphi lemmas

The probability[6] that a particular lemma is generated is depicted in figure 2. It is obtained

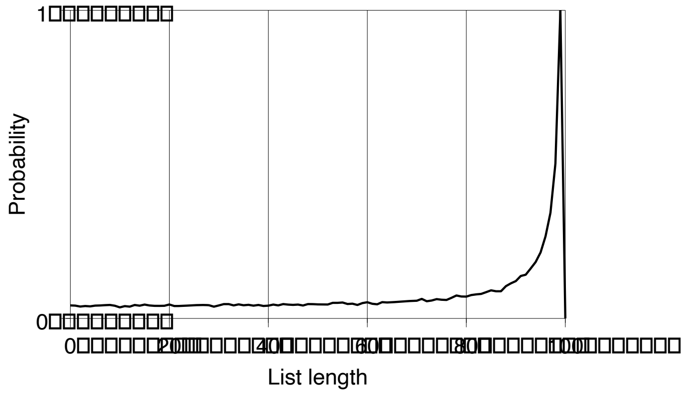

Figure 2: Probability that a lemma of list-length N is generated by the Delphi algorithm.

with a program running 2 hours and counting lemmas by length. It follows that the Delphi algorithm tends to generate the most efficient lemmas, i.e., the more complex ones with a very high probability. The explanation is basically that for every N a lemma of length N ensures that no smaller lemmas will be generated while it cannot prevent generation larger ones.

7 Conclusion

We have presented some new memoing techniques and showed their practicality by an empirical study. 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 that hashing or Ethernet collision avoidance, it gives very good performances for mostly the same reasons. The technique can be especially beneficial for lemma-intensive theorem proving systems. Future work will focus on integrating our program transformations in BinProlog's preprocessor in a fully automatic way.

Acknowledgement

We are grateful for support from the Canadian National Science and Engineering Research Council. Koen De Bosschere is research assistant with the Belgian National Fund for Scientific Research.

[6]more precisely: empirical relative frequency

References

[Apt90] K.R. Apt. Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 493–574. Elsevier, North-Holland, 1990.

[BL90] Kerima Benkerimi and John W. Lloyd. A partial evaluation procedure for logic programs. In Debray and Hermenegildo [DH90], pages 343–358.

[CvER90] M. H. M. Cheng, M. H. van Emden, and B. E. Richards. On Warren's Method for Functional Programming in Logic. In D. H. D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 546–560, Cambridge, Massachusetts London, England, 1990. MIT Press.

[DH90] Saumya Debray and Manuel Hermenegildo, editors. *Proceedings of the 1990 North American Conference on Logic Programming*, Cambridge, Massachusetts London, England, 1990. MIT Press.

[Kom82] H. J. Komorowski. Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Language: A Theory and Implementation in the case of Prolog. In *Proc. of the IXth ACM Symposium on Principles of Programming Languages.* Albuquerque, 1982.

[KR90] R. S. Kemp and G. A. Ringwood. An Algebraic framework for Abstract Interpretation of Definite Programs. In Debray and Hermenegildo [DH90], pages 516–530.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[LS91] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *JLP91*, 11(3,4), October/November 1991.

[MHMC88] P. A. Strooper M. H. M. Cheng, M. H. van Emden. Complete Sets of Frontiers in logic-based Program Transformation. MIT Press Series in Logic Programming, pages 213–225, Cambridge, Massachusetts London, England, 1988. MIT Press.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[Tar90] Paul Tarau. *Transformation de programmes logiques. Bases sémantiques et applications.* Phd thesis, Université de Montréal, november 1990.

[Tar93] Paul Tarau. Language issues and programming techniques in BinProlog. In Domenico Sacca, editor, *Proceeding of the GULP'93 Conference*, Gizzeria Lido, Italy, June 1993.

[TS86] Hisao Tamaki and Taisuke Sato. OLD resolution with tabulation. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic*

Programming, Lecture Notes in Computer Science, pages 84–98, London, 1986. Springer-Verlag.

[vE88] M. H. van Emden. Conditional Answers for Polymorphic Type Inference. pages 590–603, Cambridge, Massachusetts London, England, 1988. MIT Press.

[War92a] D. S. Warren. Memoing for logic programming. *CACM*, 35(3):37–48, March 1992.

[War92b] D. S. Warren. The XOLDT System. Technical report, SUNY Stony Brook, electronic document: ftp sbcs.sunysb.edu, 1992.