In: Proceedings of the 34th Annual Meeting of the ACL. Santa Cruz, USA. 1996.

Magic for Filter Optimization in Dynamic Bottom-up Processing

Guido Minnen*
SFB 340, University of Tübingen
Kleine Wilhelmstraße. 113
D-72074 Tübingen,
Germany
e-mail:minnen@sfs.nphil.uni-tuebingen.de

Abstract

Off-line compilation of logic grammars using *Magic* allows an incorporation of filtering into the logic underlying the grammar. The explicit definite clause characterization of filtering resulting from Magic compilation allows processor independent and logically clean optimizations of *dynamic bottom-up processing* with respect to goal-directedness. Two *filter optimization*s based on the program transformation technique of Unfolding are discussed which are of practical and theoretical interest.

1 Introduction

In natural language processing *filtering* is used to weed out those search paths that are redundant, i.e., are not going to be used in the proof tree corresponding to the natural language expression to be generated or parsed. Filter optimization often comprises an extension of a specific processing strategy such that it exploits specific knowledge about grammars and/or the computational task(s) that one is using them for. At the same time it often remains unclear how these optimizations relate to each other and what they actually mean. In this paper I show how starting from a definite clause characterization of filtering derived automatically from a logic grammar using Magic compilation, filter optimizations can be performed in a processor independent and logically clean fashion.

Magic (templates) is a general compilation technique for efficient bottom-up evaluation of logic programs developed in the deductive database community (Ramakrishnan et al., 1992). Given a logic program, Magic produces a new program in which the filtering as normally resulting from top-down evaluation is explicitly characterized through, so-called, *magic predicates*, which produce variable bindings for filtering when evaluated bottom-up. The original rules of the program are extended such that these bindings can be made effective.

As a result of the definite clause characterization of filtering, Magic brings filtering into the logic underlying the grammar. I discuss two filter optimizations. These optimizations are direction independent in the sense that they are useful for both generation and parsing. For expository reasons, though, they are presented merely on the basis of examples of generation.

Magic compilation does not limit the information that can be used for filtering. This can lead to nontermination as the tree fragments enumerated in bottom-up evaluation of magic compiled grammars are *connected* (Johnson, forthcoming). More specifically, 'magic generation' falls prey to non-termination in the face of head recursion, i.e., the generation analog of left recursion in parsing. This necessitates a *dynamic* processing strategy, i.e., memoization, extended with an abstraction function like, e.g., restriction (Shieber, 1985), to weaken filtering and a subsumption check to discard redundant results. It is shown that for a large class of grammars the subsumption check which often influences processing efficiency rather dramatically can be eliminated through fine-tuning of the magic predicates derived for a particular grammar after applying an abstraction function in an off-line fashion.

Unfolding can be used to eliminate superfluous filtering steps. Given an off-line optimization of the order in which the right-hand side categories in the rules of a logic grammar are processed (Minnen et al., 1996) the resulting processing behavior can be considered a generalization of the head corner generation approach (Shieber et al., 1990): Without the need to rely on notions such as *semantic head* and *chain rule*, a head corner behavior can be mimicked in a strict bottom-up fashion.

* url: http://www.sfs.nphil.uni-tuebingen/sfb/b4

2 Definite Clause Characterization of Filtering

Many approaches focus on exploiting specific knowledge about grammars and/or the computational task(s) that one is using them for by making filtering explicit and extending the processing strategy such that this information can be made effective. In generation, examples of such extended processing strategies are head corner generation with its semantic linking (Shieber et al., 1990) or bottom-up (Earley) generation with a semantic filter (Shieber, 1988). Even though these approaches often accomplish considerable improvements with respect to efficiency or termination behavior, it remains unclear how these optimizations relate to each other and what comprises the logic behind these specialized forms of filtering. By bringing filtering into the logic underlying the grammar it is possible to show in a perspicuous and logically clean way how and why filtering can be optimized in a particular fashion and how various approaches relate to each other.

2.1 Magic Compilation

Magic makes filtering explicit through characterizing it as definite clauses. Intuitively understood, filtering is reversed as binding information that normally becomes available as a result of top-down evaluation is derived by bottom-up evaluation of the definite clause characterization of filtering. The following is the basic Magic algorithm taken from Ramakrishnan et al. (1992).

> Let P be a program and $q(\overline{c})$ a query on the program. We construct a new program P^{mg}. Initially P^{mg} is empty.
>
> 1. Create a new predicate *magic_p* for each predicate p in P. The arity is that of p.
> 2. For each rule in P, add the *modified version* of the rule to P^{mg}. If rule r has head, say, $p(\overline{t})$, the modified version is obtained by adding the literal $magic\_p(\overline{t})$ to the body.
> 3. For each rule r in P with head, say, $p(\overline{t})$, and for each literal $q_i(\overline{t_i})$ in its body, add a *magic rule* to P^{mg}. The head is $magic\_q_i(\overline{t_i})$. The body contains the literal $magic\_p(\overline{t})$, and all the literals that precede q_i in the rule.
> 4. Create a *seed* fact $magic\_q(\overline{c})$ from the query.

To illustrate the algorithm I zoom in on the application of the above algorithm to one particular grammar rule. Suppose the original grammar rule looks as follows:

```
s(P0,P,VForm,SSem):-
   vp(P1,P,VForm,[CSem],SSem),
   np(P0,P1,CSem).
```

Step 2 of the algorithm results in the following modified version of the original grammar rule:

```
s(P0,P,VForm,SSem):-
   magic_s(P0,P,VForm,SSem),
   vp(P1,P,VForm,[CSem],SSem),
   np(P0,P1,CSem).
```

A magic literal is added to the right-hand side of the rule which 'guards' the application of the rule. This does not change the semantics of the original grammar as it merely serves as a way to incorporate the relevant bindings derived with the magic predicates to avoid redundant applications of a rule. Corresponding to the first right-hand side literal in the original rule step 3 derives the following magic rule:

```
magic_vp(P1,P,VForm,[CSem],SSem):-
   magic_s(P0,P,VForm,SSem).
```

It is used to derive from the guard for the original rule a guard for the rules defining the first right-hand side literal. The second right-hand side literal in the original rule leads to the following magic rule:

```
magic_np(P0,P1,CSem):-
   magic_s(P0,P,VForm,SSem),
   vp(P1,P,VForm,[CSem],SSem).
```

Finally, step 4 of the algorithm ensures that a seed is created. Assuming that the original rule is defining the start category, the query corresponding to the generation of the `s` "John buys Mary a book" leads to the following seed:

```
magic_s(P0,P,finite,buys(john,a(book),mary)).
```

The seed constitutes a representation of the initial bindings provided by the query that is used by the magic predicates to derive guards. Note that the creation of the seed can be delayed until run-time, i.e., the grammar does not need to be recompiled for every possible query.

2.2 Example

Magic compilation is illustrated on the basis of the simple logic grammar extract in figure 1. This grammar has been optimized automatically for generation (Minnen et al., 1996): The right-hand sides of the rules are reordered such that a simple left-to-right evaluation order constitutes the optimal evaluation order. With this grammar a simple top-down generation strategy does not terminate as a result of the head recursion in rule 3. It is necessary to use

```
(1) sentence(P0,P,decl(SSem)):-
      s(P0,P,finite,SSem).
(2) s(P0,P,VForm,SSem):-
      vp(P1,P,VForm,[CSem],SSem).
      np(P0,P1,CSem),
(3) vp(P0,P,VForm,Args,SSem):-
      vp(P0,P1,VForm,[CSem|Args],SSem),
      np(P1,P,CSem).
(4) vp(P0,P,VForm,Args,SSem):-
      v(P0,P,VForm,Args,SSem).
```

```
(5) np(P0,P,NPSem):-
      pn(P0,P,NPSem)
(6) np(P0,P,NPSem):-
      det(P0,P1,NSem,NPSem),
      n(P1,P,NSem).
(7) det([a|P],P,NSem,a(NSem)).
(8) v([buys|P],P,finite,[I,D,S],buys(S,D,I)).
(9) pn([mary|P],P,mary)
(10)n([book|P],P,book).
```

Figure 1: *Simple head-recursive grammar.*

memoization extended with an abstraction function and a subsumption check. Strict bottom-up generation is not attractive either as it is extremely inefficient: One is forced to generate all possible natural language expressions licensed by the grammar and subsequently check them against the start category. It is possible to make the process more efficient through excluding specific lexical entries with a semantic filter. The use of such a semantic filter in bottom-up evaluation requires the grammar to obey the *semantic monotonicity constraint* in order to ensure completeness(Shieber, 1988) (see below).

The 'magic-compiled grammar' in figure 2 is the result of applying the algorithm in the previous section to the head-recursive example grammar and subsequently performing two optimizations (Beeri and Ramakrishnan, 1991): All (calls to) magic predicates corresponding to lexical entries are removed. Furthermore, data-flow analysis is used to fine-tune the magic predicates for the specific processing task at hand, i.e., generation.[1] Given a user-specified *abstract query*, i.e., a specification of the intended input (Beeri and Ramakrishnan, 1991) those arguments which are not bound and which therefore serve no filtering purpose are removed. The modified versions of the original rules in the grammar are adapted accordingly. The effect of taking data-flow into account can be observed by comparing the rules for `magic_vp` and `magic_np` in the previous section with rule 12 and 14 in figure 2, respectively.

Figure 3 shows the results from generation of the `sentence` "John buys Mary a book". In the case of this example the seed looks as follows:

```
magic_sentence(decl(buys(john,a(book),mary))).
```

The *facts*, i.e., passive edges/items, in figure 3 resulted from *semi-naive* bottom-up evaluation (Ramakrishnan et al., 1992) which constitutes a dynamic bottom-up evaluation, where repeated derivation of facts from the same earlier derived facts (as in *naive* evaluation; Bancilhon, 1985) is blocked. (Active edges are not memoized.) The figure[2] consist of two tree structures (connected through dotted lines) of which the left one corresponds to the filtering part of the derivation. The filtering tree is reversed and derives magic facts starting from the seed in a bottom-up fashion. The tree on the right is the proof tree for the example sentence which is built up as a result of unifying in the derived magic facts when applying a particular rule. E.g., in order to derive fact 13, magic fact 2 is unified with the magic literal in the modified version of rule 2 (in addition to the facts 12 and 10). This, however, is not represented in order to keep the figure clear. Dotted lines are used to represent when 'normal' facts are combined with magic facts to derive new magic facts.

As can be reconstructed from the numbering of the facts in figure 3 the resulting processing behavior is identical to the behavior that would result from Earley generation as in Gerdemann (1991) except that the different filtering steps are performed in a bottom-up fashion. In order to obtain a generator similar to the bottom-up generator as described in Shieber (1988) the compilation process can be modified such that only lexical entries are extended with magic literals. Just like in case of Shieber's bottom-up generator, bottom-up evaluation of magic-compiled grammars produced with this Magic variant is only guaranteed to be complete in case the original grammar obeys the semantic monotonicity constraint.

[1]For expository reasons some data-flow information that does restrict processing is not taken into account. E.g., the fact that the `vp` literal in rule 2 is always called with a one-element list is ignored here, but see section 3.1.

[2]The numbering of the facts corresponds to the order in which they are derived. A number of lexical entries have been added to the example grammar. The facts corresponding to lexical entries are ignored. For expository reasons the phonology and semantics of lexical entries (except for `vs`) are abbreviated by the first letter. Furthermore the fact corresponding to the `vp` "buys Mary a book John" is not included.

```
(1) sentence(P0,P,decl(SSem)):-
      magic_sentence(decl(SSem)),
      s(P0,P,finite,SSem).
(2) s(P0,P,VForm,SSem):-
      magic_s(VForm,SSem),
      vp(P1,P,VForm,[CSem],SSem),
      np(P0,P1,CSem).
(3) vp(P0,P,VForm,Args,SSem):-
      magic_vp(VForm,SSem),
      vp(P0,P1,VForm,[CSem|Args],SSem),
      np(P1,P,CSem).
(4) vp(P0,P,VForm,Args,SSem):-
      magic_vp(VForm,SSem),
      v(P0,P,VForm,Args,SSem).
(5) np(P0,P,NPSem):-
      magic_np(NPSem),
      pn(P0,P,NPSem).
(6)  np(P0,P,NPSem):-
       magic_np(NPSem),
       det(P0,P1,NSem,NPSem),
       n(P1,P,NSem).
(7)  det([a|P],P,NSem,a(NSem)).
(8)  v([buys|P],P,finite,[I,D,S],buys(S,D,I)).
(9)  pn([mary|P],P,mary)
(10) n([book|P],P,book).
(11) magic_s(finite,SSem):-
       magic_sentence(decl(SSem)).
(12) magic_vp(VForm,SSem):-
       magic_s(VForm,SSem).
(13) magic_vp(VForm,SSem):-
       magic_vp(VForm,SSem).
(14) magic_np(CSem):-
       magic_s(VForm,SSem),
       vp(P1,P,VForm,[CSem],SSem).
(15) magic_np(CSem):-
       magic_vp(VForm,SSem),
       vp(P0,P1,VForm,[CSem|Args],SSem).
```

Figure 2: *Magic compiled version 1 of the grammar in figure 1.*

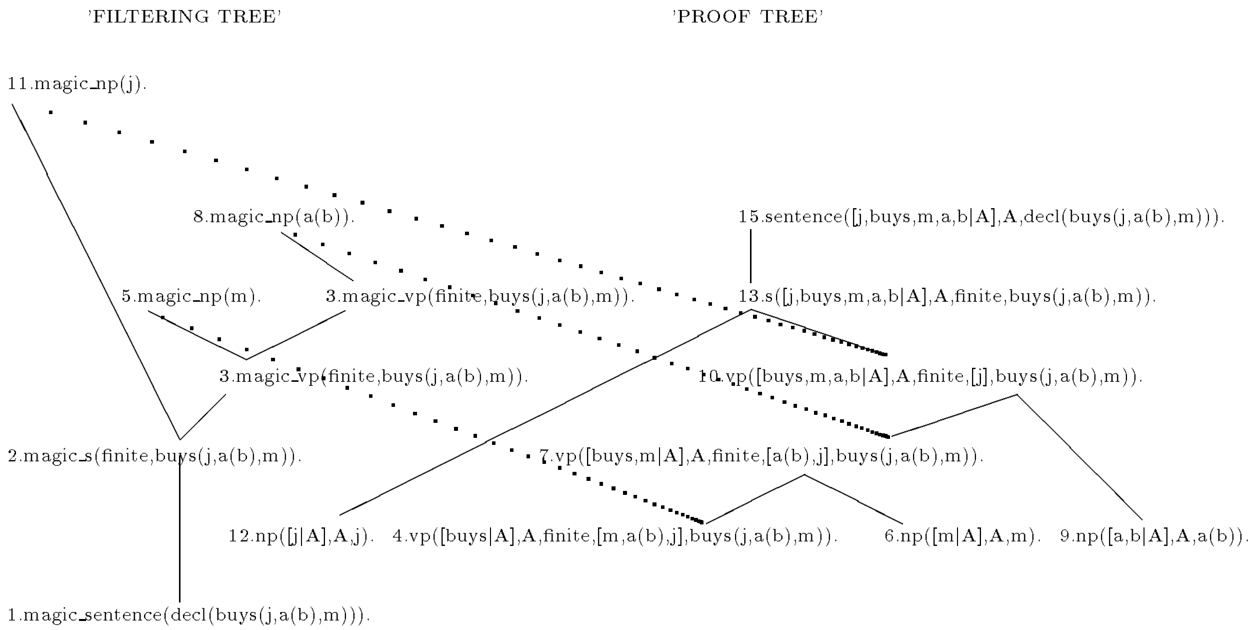

Figure 3: *'Connecting up' facts resulting from semi-naive generation of the sentence "John buys Mary a book" with the magic-compiled grammar from figure 2.*

3 Filter Optimization through Program Transformation

As a result of characterizing filtering by a definite clause representation Magic brings filtering inside of the logic underlying the grammar. This allows it to be optimized in a processor independent and logically clean fashion. I discuss two possible filter optimizations based on a program transformation technique called *unfolding* (Tamaki and Sato, 1984) also referred to as partial execution, e.g., in Pereira and Shieber (1987).

3.1 Subsumption Checking

Just like top-down evaluation of the original grammar bottom-up evaluation of its magic compiled version falls prey to non-termination in the face of head recursion. It is however possible to eliminate the subsumption check through fine-tuning the magic predicates derived for a particular grammar in an off-line fashion. In order to illustrate how the magic predicates can be adapted such that the subsumption check can be eliminated it is necessary to take a closer look at the relation between the magic predicates and the facts they derive. In figure 4 the relation between the magic predicates for the example grammar is represented by an *unfolding tree* (Pettorossi and Proietti, 1994). This, however, is not an ordinary unfolding tree as it is constructed on the basis of an *abstract seed*, i.e., a seed adorned with a specification of which arguments are to be considered bound. Note that an abstract seed can be derived from the user-specified abstract query. Only the magic part of the abstract unfolding tree is represented.

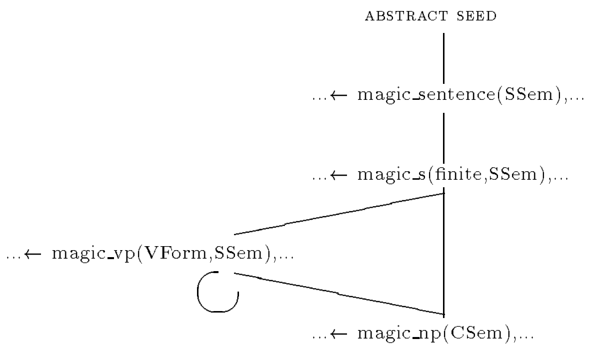

Figure 4: *Abstract unfolding tree representing the relation between the magic predicates in the compiled grammar.*

The abstract unfolding tree in figure 4 clearly shows why there exists the need for subsumption checking: Rule 13 in figure 2 produces infinitely many `magic_vp` facts. This 'cyclic' magic rule is derived from the head-recursive `vp` rule in the example grammar. There is however no reason to keep this rule in the magic-compiled grammar. It influences neither the efficiency of processing with the grammar nor the completeness of the evaluation process.

3.1.1 Off-line Abstraction

Finding these types of cycles in the magic part of the compiled grammar is in general undecidable. It is possible though to 'trim' the magic predicates by applying an abstraction function. As a result of the explicit representation of filtering we do not need to postpone abstraction until run-time, but can trim the magic predicates off-line. One can consider this as bringing abstraction into the logic as the definite clause representation of filtering is weakened such that only a mild form of connectedness results which does not affect completeness (Shieber, 1985). Consider the following magic rule:

```
magic_vp(VForm,[CSem|Args],SSem):-
   magic_vp(VForm,Args,SSem).
```

This is the rule that is derived from the head-recursive `vp` rule when the partially specified subcategorization list is considered as filtering information (cf., fn. 1). The rule builds up infinitely large subcategorization lists of which eventually only one is to be matched against the subcategorization list of, e.g., the lexical entry for "buys". Though this rule is not cyclic, it becomes cyclic upon *off-line abstraction*:

```
magic_vp(VForm,[CSem|_],SSem):-
   magic_vp(VForm,[CSem2|_],SSem).
```

Through trimming this magic rule, e.g., given a bounded term depth (Sato and Tamaki, 1984) or a restrictor (Shieber, 1985), constructing an abstract unfolding tree reveals the fact that a cycle results from the magic rule. This information can then be used to discard the culprit.

3.1.2 Indexing

Removing the direct or indirect cycles from the magic part of the compiled grammar does eliminate the necessity of subsumption checking in many cases. However, consider the magic rules 14 and 15 in figure 2. Rule 15 is more general than rule 14. Without subsumption checking this leads to spurious ambiguity: Both rules produce a magic fact with which a subject `np` can be built. A possible solution to this problem is to couple magic rules with the modified version of the original grammar rule that instigated it. To accomplish this I propose a technique that can be considered the off-line variant of an indexing technique described in Gerdemann (1991).[3] The indexing technique is illustrated on the basis of the running example: Rule 14 in figure 1 is coupled to the modified version of the original `s` rule that instigated it, i.e., rule 2. Both rules receive an index:

```
s(P0,P,VForm,SSem):-
    magic_s(P0,P,VForm,SSem),
    vp(P1,P,VForm,[CSem],SSem),
    np(P0,P1,CSem,index_1).

magic_np(CSem,index_1):-
   magic_s(P0,P,VForm,SSem),
   vp(P1,P,VForm,[CSem],SSem).
```

The modified versions of the rules defining `np`s are adapted such that they percolate up the index of the guarding magic fact that licensed its application. This is illustrated on the basis of the adapted version of rule 14:

```
np(P0,P,NPSem,INDEX):-
   magic_np(NPSem,INDEX),
   pn(P0,P,NPSem).
```

As is illustrated in section 3.3 this allows the avoidance of spurious ambiguities in the absence of subsumption check in case of the example grammar.

3.2 Redundant Filtering Steps

Unfolding can also be used to collapse filtering steps. As becomes apparent upon closer investigation of the abstract unfolding tree in figure 4 the magic predicates `magic_sentence`, `magic_s` and `magic_vp` provide virtually identical variable bindings to guard bottom-up application of the modified versions of the original grammar rules. Unfolding can be used to reduce the number of magic facts that are produced during processing. E.g., in figure 2 the `magic_s` rule:

```
magic_s(finite,SSem):-
    magic_sentence(decl(SSem)).
```

can be eliminated by unfolding the `magic_s` literal in the modified `s` rule:

```
s(P0,P,VFORM,SSem):-
    magic_s(VFORM,SSem),
    vp(P1,P,VFORM,,[CSem],SSem),
    np(P0,P1,CSem).
```

This results in the following new rule which uses the seed for filtering directly without the need for an intermediate filtering step:

```
s(P0,P,finite,SSem):-
    magic_sentence(decl(SSem)),
    vp(P1,P,finite,[CSem],SSem),
    np(P0,P1,CSem).
```

Note that the unfolding of the `magic_s` literal leads to the instantiation of the argument `VFORM` to `finite`. As a result of the fact that there are no other `magic_s` literals in the remainder of the magic-compiled grammar the `magic_s` rule can be discarded.

This filter optimization is reminiscent of computing the deterministic closure over the magic part of a compiled grammar (Dörre, 1993) at compile time. Performing this optimization throughout the magic part of the grammar in figure 2 not only leads to a more succinct grammar, but brings about a different processing behavior. Generation with the resulting grammar can be compared best with head corner generation (Shieber et al., 1990) (see next section).

3.3 Example

After cycle removal, incorporating relevant indexing and the collapsing of redundant magic predicates the magic-compiled grammar from figure 2 looks as displayed in figure 5. Figure 6 shows the chart resulting from generation of the sentence "John buys Mary a book".[4] The seed is identical to the one used for the example in the previous section. The *facts* in the chart resulted from *not-so-naive* bottom-up evaluation: semi-naive evaluation without subsumption checking (Ramakrishnan et al., 1992). The resulting processing behavior is similar to the behavior that would result from head corner generation except that the different filtering steps are performed in a bottom-up fashion. The head corner approach jumps top-down from pivot to pivot in order to satisfy its assumptions concerning the flow of semantic information, i.e., semantic chaining, and subsequently generates starting from the semantic head in a bottom-up fashion. In the example, the seed is used without any delay to apply the base case of the `vp`-procedure, thereby jumping over all intermediate chain and non-chain rules. In this respect the initial reordering of rule 2 which led to rule 2 in the final grammar in figure 5 is crucial (see section 4).

4 Dependency Constraint on Grammar

To which extent it is useful to collapse magic predicates using unfolding depends on whether the grammar has been optimized through reordering the

[3]This technique resembles an extension of Magic called *Counting* (Beeri and Ramakrishnan, 1991). However, Counting is more refined as it allows to distinguish between different levels of recursion and serves entirely different purposes.

[4]In addition to the conventions already described regarding figure 3, indices are abbreviated.

```
(1)  sentence(P0,P,decl(SSem)):-
        magic_sentence(decl(SSem)),
        s(P0,P,finite,SSem).
(2)  s(P0,P,finite,SSem):-
        magic_sentence(decl(SSem)),
        vp(P1,P,finite,[CSem],SSem),
        np(P0,P1,CSem,index_1).
(3)  vp(P0,P,finite,Args,SSem):-
        magic_sentence(decl(SSem)),
        vp(P0,P1,finite,[CSem|Args],SSem),
        np(P1,P,CSem,index_2).
(4)  vp(P0,P,finite,Args,SSem):-
        magic_sentence(decl(SSem)),
        v(P0,P,finite,Args,SSem).
(5)  np(P0,P,NPSem,INDEX):-
        magic_np(NPSem,INDEX),
        pn(P0,P,NPSem).
(6)  np(P0,P,NPSem,INDEX):-
        magic_np(NPSem,INDEX),
        det(P0,P1,NSem,NPSem),
        n(P1,P,NSem).
(7)  det([a|P],P,NSem,a(NSem)).
(8)  v([buys|P],P,finite,[I,D,S],buys(S,D,I)).
(9)  pn([mary|P],P,mary)
(10) n([book|P],P,book).
(14) magic_np(CSem,index_1):-
        magic_sentence(decl(SSem)),
        vp(P1,P,finite,[CSem],SSem).
(15) magic_np(CSem,index_2):-
        magic_sentence(decl(SSem)),
        vp(P0,P1,finite,[CSem|Args],SSem).
```

Figure 5: *Magic compiled version 2 of the grammar in figure 1.*

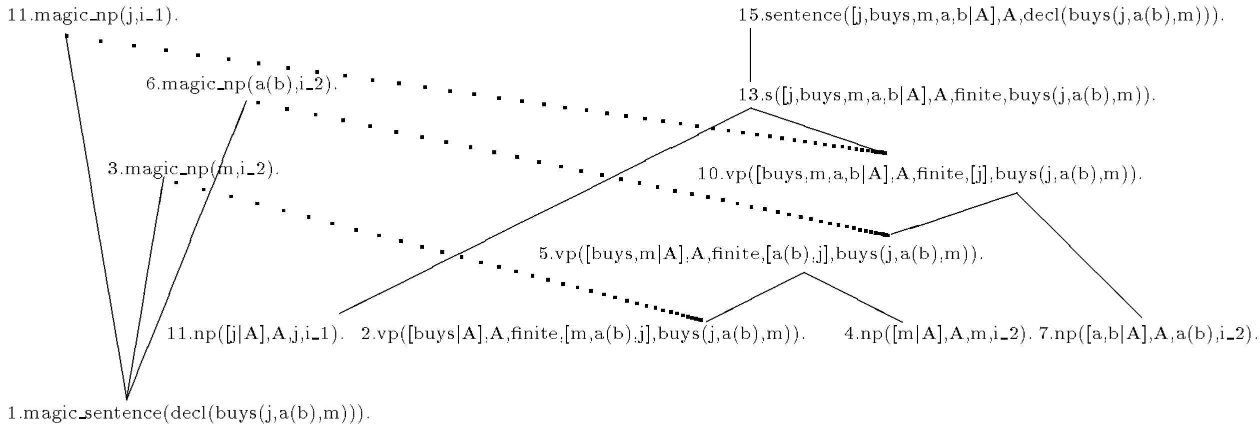

Figure 6: *'Connecting up' facts resulting from not-so-naive generation of the sentence "John buys Mary a book" with the magic-compiled grammar from figure 5.*

right-hand sides of the rules in the grammar as discussed in section 3.3. If the `s` rule in the running example is not optimized, the resulting processing behavior would not have fallen out so nicely: In this case it leads either to an intermediate filtering step for the non-chaining `sentence` rule or to the addition of the literal corresponding to the subject `np` to all chain and non-chain rules along the path to the semantic head.

Even when cycles are removed from the magic part of a compiled grammar and indexing is used to avoid spurious ambiguities as discussed in the previous section, subsumption checking can not always be eliminated. The grammar must be finitely ambiguous, i.e., fulfill the *off-line parsability constraint* (Shieber, 1989). Furthermore, the grammar is required to obey what I refer to as the *dependency constraint*: When a particular right-hand side literal can not be evaluated deterministically, the results of its evaluation must uniquely determine the remainder of the right-hand side of the rule in which it appears. Figure 7 gives a schematic example of a grammar that does not obey the dependency constraint. Given a derived fact or seed `magic_cat_1(property_1)` bottom-up evaluation of the abstract grammar in figure 7 leads to spurious ambiguity. There are two possible solutions for `cat_2` as a result of the fact that the filtering resulting from the magic literal in rule 1 is too unspecific. This is not problematic as long as this nondeterminism will eventually disappear, e.g., by combining these solutions with the solutions to `cat_3`. The problem arises as a result of the fact that these solutions lead to identical filters for the evaluation of the `cat_3` literal, i.e., the solutions to `cat_2` do not uniquely determine `cat_3`.

Also with respect to the dependency constraint an

```
(1)  cat_1(...):-
          magic_cat_1(Filter),
          cat_2(Filter,Dependency,...),
          cat_3(Dependency).
(2)  magic_cat_3(Filter):-
          magic_cat_1(Filter),
          cat_2(Filter,Dependency,...).
     ...
(3)  cat_2(property_1,property_2,...).
(4)  cat_2(property_1,property_2,...).
```

Figure 7: *Abstract example grammar not obeying the dependency constraint.*

optimization of the rules in the grammar is important. Through reordering the right-hand sides of the rules in the grammar the amount of nondeterminism can be drastically reduced as shown in Minnen et al. (1996). This way of following the intended semantic dependencies the dependency constraint is satisfied automatically for a large class of grammars.

5 Concluding Remarks

Magic evaluation constitutes an interesting combination of the advantages of top-down and bottom-up evaluation. It allows bottom-up filtering that achieves a goal-directedness which corresponds to dynamic top-down evaluation with abstraction and subsumption checking. For a large class of grammars in effect identical operations can be performed off-line thereby allowing for more efficient processing. Furthermore, it enables a reduction of the number of edges that need to be stored through unfolding magic predicates.

6 Acknowledgments

The presented research was sponsored by Teilprojekt B4 "From Constraints to Rules: Efficient Compilation of HPSG Grammars" of the Sonderforschungsbereich 340 of the Deutsche Forschungsgemeinschaft. The author wishes to thank Dale Gerdemann, Mark Johnson, Thilo Götz and the anonymous reviewers for valuable comments and discussion. Of course, the author is responsible for all remaining errors.

References

[Bancilhon1985] Francois Bancilhon. 1985. Naive Evaluation of Recursively Defined Relations. In Brodie and Mylopoulos, editors, *On Knowledge Base Management Systems - Integrating Database and* AI *Systems*. Springer-Verlag.

[Beeri and Ramakrishnan1991] Catriel Beeri and Raghu Ramakrishnan. 1991. On the Power of Magic. *Journal of Logic Programming 10*.

[Dörre1993] Jochen Dörre. 1993. Generalizing Earley Deduction for Constraint-based Grammars. Dörre and Dorna, editors, *Computational Aspects of Constraint-Based Linguistic Description I*, DYANA-2, Deliverable R1.2.A.

[Gerdemann1991] Dale Gerdemann. 1991. *Parsing and Generation of Unification Grammars*. Ph.D. thesis, University of Illinois, USA.

[Johnsonforthcoming] Mark Johnson. forthcoming. *Constraint-based Natural Language Parsing*. Brown University, Richmond, USA. Draft of 6 August 1995.

[Minnen et al.1996] Guido Minnen, Dale Gerdemann, and Erhard Hinrichs. 1996. Direct Automated Inversion of Logic Grammars. *New Generation Computing 14*.

[Pereira and Shieber1987] Fernando Pereira and Stuart Shieber. 1987. *Prolog and Natural Language Analysis*. CSLI Lecture Notes, No. 10. Center for the Study of Language and Information, Chicago, USA.

[Pettorossi and Proietti1994] Alberto Pettorossi and Maurizio Proietti. 1994. Transformations of Logic Programs: Foundations and Techniques. *Journal of Logic Programming 19/20*.

[Ramakrishnan et al.1992] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. 1992. Efficient Bottom-up Evaluation of Logic Programs. In Vandewalle, editor, *The State of the Art in Computer Systems and Software Engineering*. Kluwer Academic Publishers.

[Sato and Tamaki1984] Taisuke Sato and Hisao Tamaki. 1984. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Sience 34*.

[Shieber et al.1990] Stuart Shieber, Gertjan van Noord, Robert Moore, and Fernando Pereira. 1990. Semantic Head-driven Generation. *Computational Linguistics 16*.

[Shieber1985] Stuart Shieber. 1985. Using Restriction to Extend Parsing Algorithms for Complex Feature-based Formalisms. In *Proceedings of the 23rd Annual Meeting Association for Computational Linguistics*, Chicago, USA.

[Shieber1988] Stuart Shieber. 1988. A Uniform Architecture for Parsing and Generation. In *Proceedings of the 12th Conference on Computational Linguistics*, Budapest, Hungary.

[Shieber1989] Stuart Shieber. 1989. Parsing and Type Inference for Natural and Computer Languages. Ph.D. thesis, Stanford University, USA.

[Tamaki and Sato1984] Hisao Tamaki and Taisuke Sato. 1984. Unfold/Fold Transformation of Logic Programs. In *Proceedings of the 2nd International Conference on Logic Programming*, Uppsala, Sweden.