6

Program transformations for optimization of parsing algorithms and other weighted logic programs

JASON EISNER AND JOHN BLATZ

Abstract

Dynamic programming algorithms in statistical natural language processing can be easily described as weighted logic programs. We give a notation and semantics for such programs. We then describe several source-to-source transformations that affect a program's efficiency, primarily by rearranging computations for better reuse or by changing the search strategy. We present practical examples of using these transformations, mainly to optimize context-free parsing algorithms, and we formalize them for use with new weighted logic programs.

Specifically, we define *weighted* versions of the folding and unfolding transformations, whose unweighted versions are used in the logic programming and deductive database communities. We then present a novel transformation called speculation—a powerful generalization of folding that is motivated by gap-passing in categorial grammar. Finally, we give a simpler and more powerful formulation of the magic templates transformation.[1]

Keywords WEIGHTED LOGIC PROGRAMMING, DYNAMIC PROGRAMMING, PROGRAM TRANSFORMATION, PARSING ALGORITHMS

6.1 Introduction

In this paper, we show how some algorithmic efficiency tricks used in the natural language processing (NLP) community, particularly for parsing, can be regarded as specific instances of transformations on weighted logic programs.

[1] This material is based upon work supported by the National Science Foundation under Grants No. 0313193 and 0347822 to the first author.

Proceedings of FG-2006.
Editor: Shuly Wintner.

We define weighted logic programs and sketch the general form of the transformations, enabling their application to new programs in NLP and other domains. Several of the transformations (folding, unfolding, magic templates) have been known in the logic programming community, but are generalized here to our weighted framework and applied to NLP algorithms. We also present a powerful generalization of folding—speculation—which appears new and is able to derive some important parsing algorithms.

We also formalize these transformations in a way that we find more intuitive than conventional presentations. Influenced by the mechanisms of categorial grammar, we introduce "slashed" terms whose values may be regarded as functions. These slashed terms greatly simplify our constructions. In general, our work can be connected to the well-established literature on grammar transformation.

The framework that we use for specifying the weighted logic programs is roughly based on that of Dyna (Eisner et al., 2005), an implemented system that can compile such specifications into efficient C++. Some of the programs could also be handled by PRISM (Zhou and Sato, 2003), an implemented probabilistic Prolog.

It is especially useful to have general optimization techniques for dynamic programming algorithms (a special case in our framework), because computational linguists regularly propose new such algorithms. Dynamic programming is used to parse many different grammar formalisms, and in syntax-based approaches to machine translation and language modeling. It is also used in finite-state methods, stack decoding, and grammar induction.

One might select program transformations either manually or automatically. Our goal here is simply to illustrate the search space of semantically equivalent programs. We do not address the practical question of searching this space—that is, the question of where and when to apply the transformations. For some programs and their typical inputs, a transformation will speed a program up (at least on some machines); in other cases, it will slow it down. The actual effect can of course be determined empirically by running the transformed program on typical inputs (or, in some cases, can be reasonably well predicted from runtime profiles of the *untransformed* program). Thus, one could in principle use automated methods, such as stochastic local search, to search for sets of transformations that provide good practical speedups.

6.2 Weighted Logic Programming

Before moving to the actual transformations, we will take several pages to describe our proposed formalism of weighted logic programming.

6.2.1 Logical Specification of Dynamic Programs

We will use context-free parsing as a simple running example. Recall that one can write a logic program for CKY recognition (Younger, 1967) as follows, where constit(X,I,K) is provable iff the context-free grammar (CFG), starting at nonterminal X, can generate the input substring from position I to position K. The capitalized symbols are **variables**.

```
constit(X,I,K) :- rewrite(X,W), word(W,I,K).
constit(X,I,K) :- rewrite(X,Y,Z), constit(Y,I,J), constit(Z,J,K).
goal :- constit(s,0,N), length(N).

rewrite(s,np,vp).      % tiny grammar
rewrite(np,det,n).
rewrite(np,"Dumbo").
rewrite(np,"flies").
rewrite(vp,"flies").

word("Dumbo",0,1). % tiny input sentence
word("flies",1,2).
length(2).
```

For example, the second line permits us to prove the proposition constit(X,I,K) once we can prove that there exist constituents constit(Y,I,J) and constit(Z,J,K)—which are adjacent[2]—as well as a context-free grammar rule rewrite(X,Y,Z) (i.e, X → Y Z) to combine them. This deduction is permitted for *any* specific values of X,Y,Z (presumably nonterminals of the grammar) and I,J,K (presumably positions in the sentence).

We suppose in this paper that the whole program above is specified at compile time. In practice, one might instead wait until runtime to provide the description of the sentence (the word and length facts) and perhaps even of the grammar (the rewrite facts). In this case our transformations could be used only on the part of the program specified at compile time.[3]

The basic objects of the program are **terms**, defined in the usual way (as in Prolog). Following parsing terminology, we refer to some terms as **items**; these are terms that the program might prove in the course of its execution, such as constit(np,0,1) but not np (which appears only as a *sub-term* of items) nor constit(foo(bar),baz).[4] Each line in the program is an **inference rule** (or

[2] By convention, we regard positions as falling *between* input words, so that the substring from I to J is immediately adjacent to the substring from J to K.

[3] This is generally safe provided that the runtime rules may not define in the head, nor evaluate in the body, any term that unifies with the head of a compile-time rule. It is common to assume further that all the runtime rules are facts, known collectively as the **database**.

[4] It is of course impossible to determine precisely which terms the program *will* prove without running it. It is merely helpful to refer to terms as items when we are discussing their provability or, in the case of weighted logic programs, their value. ("Item" does have a more formal meaning

clause).

Each of the inference rules in the above example is **range-restricted**. In the jargon of logic programming, this means that all **variables** (capitalized) in the rule's left-hand side (rule **head**) also appear in its right-hand side (rule **body**). A rule with an empty body is called a **fact**. If all rules are range-restricted, then all provable terms are **ground terms**, i.e., terms such as constit(s,0,2) that do not contain any variables.

Logic programs restricted in this way correspond to the "grammatical deduction systems" discussed by Shieber et al. (1995). Sikkel (1997) gives many parsing algorithms in this form. More generally, programs that consist entirely of range-restricted rules correspond to conventional dynamic programming algorithms, and we may refer to them informally as **dynamic programs**.

Dynamic programs can be evaluated by various techniques. The specific technique chosen is not of concern to this paper except in section 6.6. However, for most NLP algorithms, it is common to use a bottom-up or **forward chaining** strategy such as the one given by Shieber et al., which iteratively proves all transitive consequences of the facts in the program. In the example above, forward chaining starts with the word, rewrite, and length facts and derives successively wider constit items, eventually deriving goal iff the input sentence is grammatical. This corresponds to chart parsing, with the role of the chart being played by a data structure that remembers which items have been proved so far.[5]

This paper deals with general logic programs, not just dynamic programs. For example, one may wish to state once and for all that an "epsilon" word is available at *every* position K in the sentence: word(epsilon,K,K). We allow this because it will be convenient for most of our transformations to introduce new non-range-restricted rules, which can *derive* non-ground items such as word(epsilon,K,K). The above execution strategies continue to apply, but the presence of non-ground items means that they must now use unification matching to find previously derived terms of a given form. For example, if the non-ground item word(epsilon,K,K) has been derived, representing an infinite collection of ground terms, then if the program looks up the set of terms in the chart matching word(W,2,K), it should find (at least) word(epsilon,2,2).

in a practical setting—where, for efficiency, the user or the compiler declares an item datatype that is guaranteed to be able to represent at least all provable terms, though not necessarily all terms. We then use "item" to refer to terms that can be represented by this explicit datatype.)

[5] An alternative strategy is Prolog's top-down **backward-chaining** strategy, which starts by trying to prove goal and tries to prove other items as subgoals. However, this strategy will waste exponential time by re-deriving the same constituents in different contexts, or will fail to terminate if the grammar is left-recursive. It may be rescued by memoization, also known as "tabling," which re-introduces a chart (Sagonas et al., 1994).

One can often eliminate non-range-restricted rules (in particular, the ones we introduce) to obtain a semantically equivalent dynamic program, but we do not here explore transformations for doing so systematically.

6.2.2 Weighted Logic Programs

We now define our notion of *weighted* logic programs, of which the most useful in NLP are the semiring-weighted dynamic programs discussed by Goodman (1999) and Eisner et al. (2005). See the latter paper for a discussion of relevant work on deductive databases with aggregation (e.g., Fitting, 2002, Van Gelder, 1992, Ross and Sagiv, 1992).

In a weighted logic program, each provable item has a *value*. Our running example is the inside algorithm for context-free parsing:

```
constit(X,I,K) += rewrite(X,W) * word(W,I,K).
constit(X,I,K) += rewrite(X,Y,Z) * constit(Y,I,J) * constit(Z,J,K).
goal += constit(s,0,N) * length(N).

rewrite(s,np,vp) = 1.                % p(s → np vp | s)
rewrite(np,det,n) = 0.5.             % p(np → det n | np)
rewrite(np,”Dumbo”) = 0.4.           % p(np → ”Dumbo” | np)
rewrite(np,”flies”) = 0.1.           % p(vp → ”flies” | vp)
rewrite(vp,”flies”) = 1.             % p(vp → ”flies” | vp)

word(”Dumbo”,0,1) = 1.               % 1 for all words in the sentence
word(”flies”,1,2) = 1.
length(2) = 1.
```

This looks just like the unweighted logic program in section 6.2.1, except that now the body of each inference rule is an arbitrary *expression*, and the :- operator is replaced by an **aggregation operator** such as += or max=. One might call these rules "Horn equations," by analogy with the (definite) Horn clauses of the unweighted case. A **fact** is now a rule whose body is a constant or an expression on constants.

To understand the meaning of the above program, consider for example the item constit(s,0,2). The old version of line 2 allowed one to *prove* constit(s,0,2) if rewrite(s,Y,Z), constit(Y,0,J), and constit(Z,J,2) were all true for at least one triple Y,Z,J. The new version of line 2 instead *defines the value* of constit(s,0,2)—or more precisely, as

$$\sum_{\mathrm{Y,Z,J}} \mathrm{rewrite(s,Y,Z)} * \mathrm{constit(Y,0,J)} * \mathrm{constit(Z,J,2)}$$

The aggregation operator += requires a sum over all ways of grounding the variables that appear only in the rule body, namely Y, Z, and J. The rest of the value of constit(s,0,2) is added in by line 1. We will formalize all of this in section 6.2.3 below.

To put this another way, one way of grounding line 2 (i.e., one way of substituting a ground term for each of its variables) is constit(s,0,2) += rewrite(s,np,vp) * constit(np,0,1) * constit(vp,1,2). Therefore, one operand to += in defining the value of constit(s,0,2) will be the value (if defined) of rewrite(s,np,vp) * constit(np,0,1) * constit(vp,1,2).

The result—for this program—is that the computed value of constit(X,I,J) will be the traditional inside probability $\beta_X(I, J)$ for a particular input sentence and grammar.[6]

If the heads of two rules unify, then the rules must use the same aggregation operator, to guarantee that each provable term's value is aggregated in a consistent way. Each constit(...) term above is aggregated with +=.

Substituting max= for += throughout the program would find Viterbi probabilities (best derivation) rather than inside probabilities (sum over derivations). Similarly, we can obtain the unweighted recognizer of section 6.2.1 by writing expressions over boolean values:[7]

constit(X,I,K) |= rewrite(X,$\underline{\text{Y}}$,$\underline{\text{Z}}$) & constit($\underline{\text{Y}}$,I,$\underline{\text{J}}$) & constit($\underline{\text{Z}}$,$\underline{\text{J}}$,K).

Of course, these programs are all essentially recognizers rather than parsers. They only compute a boolean or real value for goal. To recover actual parse trees, one can extract the proof trees of goal. To make the parse trees accessible to the program itself, one can define a separate item parse(constit(X,I,K)) whose value is a tree.[8] We do not give the details here to avoid introducing new notation and issues that are orthogonal to the scope of this paper.

The above examples, like most of our examples, can be handled by the framework of Goodman (1999). However, we allow a wider class of rules. Goodman allows only range-restricted rules (cf. our section 6.2.1), and he requires all values to fall in a single semiring and all rules to use only the semiring operations. The latter requirements—in particular the distributivity property of the semiring—imply that an item's value can be found by separately computing values for all of its complete proof trees and then aggregating them at the end. That is not the case for neural networks, game trees,

[6] However, unlike probabilistic programming languages (Zhou and Sato, 2003), we do not enforce that values be reals in [0, 1] or have probabilistic interpretations.

[7] Using | for "or" and & for "and." The aggregation operators |= and &= can be regarded as implementing existential and universal quantification.

[8] Another option is to say that the value of constit(X,I,K) is not just a number but a (number,tree) pair, and to define max= over such pairs Goodman (1999). This resembles the use of semantic attachments to build output in programming language parsers. However, it requires building a tree (indeed, many trees, of which the best is kept) for each constit, including constituents that do not appear in the final parse. Our preferred scheme is to hold the best tree in a separate parse(constit(X,I,K)) item. Then we can choose to use backward chaining, or the magic templates transformation of section 6.6, to limit our computation of parse trees to those that are needed to assemble the final tree, parse(goal).

practical NLP systems that mix summation and maximization, or other useful systems of equations that can be handled in our more general framework.

6.2.3 Semantics of Weighted Logic Programs

We now formalize the semantics of a weighted logic program, and define what it means for a program transformation to preserve the semantics. Readers who are interested in the actual transformations may skip this section, except for the brief definitions of the special aggregation operator = and of side conditions.

In an unweighted logic program, the semantics is the set of provable ground terms.[9] For a *weighted* logic program, the semantics is a partial function, the **valuation function**, that maps each provable ground term r to a value $[\![r]\!]$. All items in our example above take values in $\mathbb{R}$. However, one could use values of any type or of multiple types.

The domain of the valuation function $[\![\cdot]\!]$ is the set of ground terms for which there exist finite proofs under the *unweighted* version of the program. We extend $[\![\cdot]\!]$ in the obvious way to expressions on provable ground terms: for example, $[\![x * y]\!] \overset{\text{def}}{=} [\![x]\!] * [\![y]\!]$ provided that $[\![x]\!]$ and $[\![y]\!]$ are defined.

For each ground term r that is provable in program $\mathcal{P}$, let $\mathcal{P}(r)$ be the non-empty multiset of all expressions E, over provable ground terms, such that $r \oplus_r\!= E$ grounds some rule of $\mathcal{P}$. Here $\oplus_r\!=$ denotes the single aggregation operator shared by all those rules.

We now interpret the weighted rules as a set of simultaneous equations that constrain the $[\![\cdot]\!]$ function. If $\oplus_r\!=$ is +=, then we require that

$$[\![r]\!] = \sum_{E \in \mathcal{P}(r)} [\![E]\!]$$

(perhaps permitting $[\![r]\!] = \infty$ if the sum diverges). More generally, we require that

$$[\![r]\!] = [\![E_1]\!] \oplus_r [\![E_2]\!] \oplus_r \ldots$$

where $\mathcal{P}(r) = \{E_1, E_2, \ldots\}$. For this to be well-defined, $\oplus_r$ must be associative and commutative. If $\oplus_r =$ is the special operator =, as in the final rules of our example, then we set $[\![r]\!] = [\![E_1]\!]$ if $\mathcal{P}(r)$ is a singleton set $\{E_1\}$, and generate a runtime error otherwise.

Example. In the example of section 6.2.2, lines 1–2, this means that for any particular X, I, K for which $\mathsf{constit}(X,I,K)$ is a provable item, $[\![\mathsf{constit}(X, I, K)]\!]$ equals

[9] Note that if a non-ground term can be proved under the program, so can any one of the infinitely many ground terms that instantiates (specializes) that non-ground term. Our formal semantics are described in terms of these ground terms only.

$$\begin{aligned} & \textstyle\sum_{W} \llbracket \mathsf{rewrite}(X,W) \rrbracket * \llbracket \mathsf{word}(W,I,J) \rrbracket \\ + \quad & \textstyle\sum_{J,Y,Z} \llbracket \mathsf{rewrite}(X,Y,Z) \rrbracket * \llbracket \mathsf{constit}(Y,I,J) \rrbracket * \llbracket \mathsf{constit}(Z,J,K) \rrbracket \end{aligned}$$

where, for example, the second summation ranges over term triples J, Y, Z such that the summand has a value. We sum over J, Y, Z because they do not appear in the rule's head $\mathsf{constit}(X, I, K)$, which is being defined.

Remark. Our constraints on the valuation function $\llbracket\cdot\rrbracket$ are equivalent to saying that it is a fixed point of an "equational update" operator $\mathbf{T}_{\mathcal{P}}$,[10] which acts on valuation functions I and is analogous to the "monotone consequence" operator for unweighted logic programs. Such a fixed point need not be unique.[11] Operationally, one may seek a single fixpoint by initializing $I = \{\}$, repeatedly updating I to $\mathbf{T}_P(I)$, and hoping for convergence. That is the basic idea of the forward-chaining algorithm in section 6.2.4 below.

Side conditions. A mechanism for handling "side conditions" (e.g., Goodman, 1999) is to use rules like[12]

```
a += b * c whenever ?d.
```

We define $\llbracket \texttt{b * c whenever ?d} \rrbracket \overset{\text{def}}{=} \llbracket \texttt{b * c} \rrbracket$, independent of the value of d. But by our earlier definitions, it will appear in $\mathcal{P}(\mathsf{a})$ and be added into $\llbracket \mathsf{a} \rrbracket$ only if the side condition d, along with b and c, is provable.

Definition. Roughly speaking, a program transformation $\mathcal{P} \rightarrow \mathcal{P}'$ is said to be **semantics-preserving** iff it preserves the partial function $\llbracket\cdot\rrbracket$. In other words, exactly the same ground terms must be provable under both programs, and they must have the same values.

We make two adjustments to this rough definition. First, for generality, we must handle the case where $\mathcal{P}$ and $\mathcal{P}'$ do not both have uniquely determined semantics. In general, we say that the transformation is semantics-preserving iff it preserves the *set* of valuation functions.

Second, we would like a program transformation to be able to introduce new provable items for its own use. Therefore, we only require that it preserve

[10] That is, $\llbracket\cdot\rrbracket = \mathbf{T}_{\mathcal{P}}(\llbracket\cdot\rrbracket)$. Given a valuation function I, $\mathbf{T}_{\mathcal{P}}(I)$ is defined as follows: for ordinary ground terms r, put

$$(\mathbf{T}_P(I))(r) = \bigoplus_{\substack{E \text{ such that } E \text{ is a ground expression where} \\ I(E) \text{ is defined and } r \oplus_r\!= E \text{ grounds some rule of } \mathcal{P}}}\!\!{}_r \quad I(E)$$

if this sum is non-empty, and leave it undefined otherwise. Then extend $\mathbf{T}_{\mathcal{P}}(I)$ over expressions as usual.

[11] There is a rich line of research that attempts to more precisely characterize which fixed point gives the "intuitive" semantics of a logic program with negation or aggregation (see e.g. Fitting, 2002, Van Gelder, 1992, Ross and Sagiv, 1992).

[12] $\mathsf{whenever\ ?d}$ is defined to mean "whenever d is provable," whereas $\mathsf{whenever\ d}$ would mean "whenever d's value is true." The latter construction is also useful, but not needed in this paper.

the *restriction* of $[\![\cdot]\!]$ to the Herbrand base of $\mathcal{P}$ (more precisely, to the Herbrand base of all expressible constants and the functors in $\mathcal{P}$). Thus, a transformed version of the inside algorithm would be allowed to prove additional temp(...) items, but not additional constit(...) items. The user may therefore safely interrogate the transformed program to find out whether constit(np,0,5) is provable and if so, what its value is.

Notice that a two-step transformation $\mathcal{P} \to \mathcal{P}'' \to \mathcal{P}'$ might introduce new temp(...) items in the first step and eliminate them in the second. This composite transformation may still be semantics preserving even though its second step $\mathcal{P}'' \to \mathcal{P}'$ is not.

All of the transformations developed in this paper are intended to be semantics-preserving (except for rule elimination and magic templates, which preserve the semantics of only a subset of the ground terms). To prove this formally, one would show that every fixed point of $T_{\mathcal{P}'}$ is also a fixed point of $T_{\mathcal{P}}$, when restricted to the Herbrand base of $\mathcal{P}$, and conversely, that every fixed point of $\mathcal{P}$ can be extended to a fixed point of $T_{\mathcal{P}'}$.

6.2.4 Computing Semantics by Forward Chaining

A basic strategy for computing a semantic interpretation is "forward chaining." The idea is to maintain current values for all proved items, and to propagate updates to these values, from the body of a rule to its head, until all the equations are satisfied. This may be done in any order, or in parallel (as for the equational update operator of section 6.2.3). Note that in the presence of cycles such as x += 0.9 * x, the process can still converge *numerically* in finite time (to finite values or to ∞, representing divergence). Indeed, the forward chaining strategy terminates in practice for many programs of practical interest.[13]

As already noted in section 6.2.1, Shieber et al. (1995) gave a forward chaining algorithm (elsewhere called "semi-naive bottom-up evaluation") for unweighted *dynamic* programs. Eisner et al. (2005) extended this to handle arbitrary semiring-weighted dynamic programs. Goodman (1999) gave a mixed algorithm.

Dealing with our full class of weighted logic programs—not just semiring-weighted dynamic programs—is a substantial generalization. Once we allow inference rules that are not range-restricted, the algorithm must derive non-ground items and store them and their values in the chart, and obtain the value of foo(3,3), if not explicitly derived, by "backing off" to the derived value of non-ground items such as foo(X,X) or foo(X,3), which are preferred in turn to

[13] Of course, no strategy can possibly terminate on all programs, because the language (even when limited to unweighted range-restricted rules) is powerful enough to construct arbitrary Turing machines. We remark that forward chaining may fail to terminate either because of oscillation or because infinitely many items are derived (e.g., s(N) = N).

the less specific foo(X,Y). Once we drop the restriction to semirings, the algorithm must propagate arbitrary updates (notice that it is not trivial to update the result of max= if one of its operands decreases). Certain aggregation operators also allow important optimizations thanks to their special properties such as distributivity and idempotence. Finally, we may wish to allow rules such as reciprocal(X) = 1/X that cannot be handled at all by forward chaining. We defer all these algorithmic details to a separate paper, focusing instead on the denotational semantics.

6.3 Folding: Factoring Out Subexpressions

Weighted logic programs are schemata that define possibly infinite systems of simultaneous equations. Finite systems of equations can often be rearranged without affecting their solutions (e.g., Gaussian elimination). In the same way, weighted logic programs can be transformed to obtain new programs with better runtime.

Notation. We will henceforth adopt a convention of underlining any variables that appear only in a rule's body, to more clearly indicate the range of the summation. We will also underline any variables that appear only in the rule's head; these indicate that the rule is not range-restricted.

Example. Consider first our previous rule from section 6.2.2,

constit(X,I,K) += rewrite(X,$\underline{Y}$,$\underline{Z}$) * constit($\underline{Y}$,I,$\underline{J}$) * constit($\underline{Z}$,$\underline{J}$,K).

If the grammar has N nonterminals, and the input is an n-word sentence or an n-state lattice, then the above rule can be grounded in only $O(N^3 \cdot n^3)$ different ways. For this—and the other parsing programs we consider here—it turns out that the runtime of forward chaining can be kept down to $O(1)$ time per grounding.[14] Thus the runtime is $O(N^3 \cdot n^3)$.

However, the following pair of rules is equivalent:

temp(X,Y,Z,I,J) = rewrite(X,Y,Z) * constit(Y,I,J).

constit(X,I,K) += temp(X,$\underline{Y}$,$\underline{Z}$,I,$\underline{J}$) * constit($\underline{Z}$,$\underline{J}$,K).

We have just performed a weighted version of the classical **folding** transformation for logic programs (Tamaki and Sato, 1984). The original body expression would be explicitly parenthesized as (rewrite(X,Y,Z) * constit(Y,I,J)) * constit(Z,J,K); we have simply introduced a "temporary item" (like a temporary variable in a traditional language) to hold the result of the parenthesized subexpression, then "folded" that temporary item into the computation

[14] Assuming that the grammar is acyclic (in that it has no unary rule cycles) and so is the input lattice. Even without such assumptions, a meta-theorem of McAllester (1999) allows one to derive asymptotic run-times of appropriately-indexed forward chaining from the number of instantiations. However, that meta-theorem applies only to unweighted dynamic. Similar results in the weighted case require acyclicity. Then one can use the two-phase method of Goodman (1999), which begins by running forward chaining on an unweighted version of the program.

of constit. The temporary item mentions all the capitalized variables in the expression.

Distributivity. A more important use appears when we combine folding with the distributive law. In the example above, the second rule's body sums over the (underlined) free variables, J, Y, and Z. However, Y appears only in the temp item. We could therefore have summed over values of Y *before* multiplying by constit(Z,J,K), obtaining the following transformed program instead:

```
temp2(X,Z,I,J) += rewrite(X,Y,Z)  * constit(Y,I,J).
constit(X,I,K)   += temp2(X,Z,I,J) * constit(Z,J,K).
```

This version of the transformation is permitted only because + distributes over *.[15] By "forgetting" Y as soon as possible, we have reduced the runtime of CKY from $O(N^3 \cdot n^3)$ to $O(N^3 \cdot n^2 + N^2 \cdot n^3)$.

Using the distributive law to improve runtime is a well-known technique. Aji and McEliece (2000) present what they call a "generalized distributive law," which is equivalent to repeated application of the folding transformation. While their inspiration was the junction-tree algorithm for probabilistic inference in graphical models (discussed below), they demonstrate the approach to be useful on a broad class of weighted logic programs.

A categorial grammar view of folding. From a parsing viewpoint, notice that the item temp2(X,Z,I,J) can be regarded as a categorial grammar constituent: an incomplete X missing a subconstituent Z at its right (i.e., an X/Z) that spans the substring from I to J. This leads us to an interesting and apparently novel way to write the transformed program:

```
constit(X,I,K)/constit(Z,J,K) += rewrite(X,Y,Z) * constit(Y,I,J).
constit(X,I,K) += constit(X,I,K)/constit(Z,J,K)  * constit(Z,J,K).
```

Here A/B is syntactic sugar for slash(A,B). That is, / is used as an infix functor and does not denote division, However, it is meant to *suggest* division: as the second rule shows, A/B is an item which, if multiplied by B, yields a summand of A. In effect, the first rule above is derived from the original rule at the start of this section by dividing both sides by constit(Z,J,K). The second rule multiplies the missing factor constit(Z,J,K) back in, now that the first rule has summed over Y.

Notice that K appears free (and hence underlined) in the head of the first rule. The only slashed items that are actually *provable* by forward chaining are non-ground terms such as constit(s,0,K)/constit(vp,1,K). That is, they have the form constit(X,I,K)/constit(Z,J,K) where X,I,J are ground variables but K remains free. The way that K appears twice in the slashed item (i.e., internal

[15] All semiring-weighted programs enforce a similar distributive property. In particular, the trick can be applied equally well to common cases discussed in section 6.2.2: Viterbi parsing (max distributes over either * or +) and unweighted recognition (| distributes over &.

unification) indicates that the missing Z is always at the *right* of the X, while the fact that K remains a variable means that the shared right edge of the full X and missing Z are still unknown (and will remain unknown until the second rule fills in a particular Z). Thus, the first rule performs a computation once for *all* possible K—always the source of folding's efficiency.

Our earlier program with temp2 could now be obtained by a further automatic transformation that replaces all constit(X,I,K)/constit(Z,J,K) having free K with the more compactly stored temp2(X,Z,I,J). The resulting rules are all range-restricted.

We emphasize that although our slashed items are inspired by categorial grammars, they can be used to describe folding in *any* weighted logic program. Section 6.5 will further exploit the analogy to obtain a novel "speculation" transformation.

Further applications. The folding transformation unifies various ideas that have been disparate in the natural language processing literature. Eisner and Satta (1999) speed up parsing with bilexical context-free grammars from $O(n^5)$ to $O(n^4)$, using precisely the above trick (see section 6.4 below). Huang et al. (2005) employ the same "hook trick" to improve the complexity of syntax-based MT with an n-gram language model.

Another parsing application is the common "dotted rule" trick (Earley, 1970). If one's CFG contains ternary rules X → Y1 Y2 Y3, the naive CKY-like algorithm takes $O(N^4 \cdot n^4)$ time:

```
constit(X,I,L) += ((rewrite(X,Y1,Y2,Y3) * constit(Y1,I,J))
                         * constit(Y2,J,K)) * constit(Y3,K,L).
```

Fortunately, folding allows one to sum first over Y1 before summing separately over Y2 and J, and then over Y3 and K:

```
temp3(X,Y2,Y3,I,J) += rewrite(X,Y1,Y2,Y3) * constit(Y1,I,J).
temp4(X,Y3,I,K)    += temp3(X,Y2,Y3,I,J)  * constit(Y2,J,K).
constit(X,I,L)     += temp4(X,Y3,I,K)     * constit(Y3,K,L).
```

This restores $O(n^3)$ runtime (more precisely, $O(N^4 \cdot n^2 + N^3 \cdot n^3 + N^2 \cdot n^3)$)[16] by reducing the number of nested loops. Even if we had declined to sum over Y1 and Y2 in the first two rules, then the summation over J would already have obtained $O(n^3)$ runtime, in effect by binarizing the ternary rule. For example, temp4(X,Y1,Y2,Y3,I,K) would have corresponded to a partial constituent matching the *dotted* rule X → Y1 Y2 . Y3. The additional summations over Y1 and Y2 result in a more efficient dotted rule that "forgets" the names of the nonterminals matched so far, X → ? ? . Y3. This takes further advantage of distributivity by aggregating dotted-rule items (with +=) that will behave the same in subsequent computation.

[16] For a dense grammar, which may have up to N^4 ternary rules. Tighter bounds on grammar size would yield tighter bounds on runtime.

The variable elimination algorithm for graphical models can be viewed as repeated folding. An undirected graphical model expresses a joint probability distribution over P,Q by marginalizing (summing) over a product of clique potentials. In our notation,

```
marginal(P,Q) += p1(...) * p2(...) * ··· * pn(...).
```

where a function such as p5(Q,X,Y) represents a clique potential over graph nodes corresponding to the random variables Q,X,Y. Assume without loss of generality that variable X appears as an argument only to p_{k+1}, p_{k+2}, . . . , p_n. We may *eliminate* variable X by transforming to

```
temp5(...)      += p_{k+1}(..., X, ...) * ··· * p_n(..., X, ...).
marginal(P,Q)  += p_1(...)               * ··· * p_k(...)          * temp5(...).
```

Line 2 no longer mentions X because line 1 has summed over it. To eliminate the remaining variables one at a time, the variable elimination algorithm applies this procedure repeatedly to the last line.[17]

Common subexpression elimination. Folding can also be used multiple times to eliminate common subexpressions. Consider the following code, which is part of an inside algorithm for *bilexical* CKY parsing:[18]

```
constit(X:H,I,K) += rewrite(X:H,Y:H,Z:H2)
                    * constit(Y:H,I,J) * constit(Z:H2,J,K).
constit(X:H,I,K) += rewrite(X:H,Y:H2,Z:H)
                    * constit(Y:H2,I,J) * constit(Z:H,J,K).
```

Here X:H is syntactic sugar for ntlex(X,H), meaning a nonterminal X whose head word is the lexical item H. The grammar uses two types of lexicalized binary productions (defined by rewrite facts not shown here), which pass the head word to the left or right child, respectively.

We could fold together the last two factors of the first rule to obtain

```
temp6(Y:H,Z:H2,I,K) += constit(Y:H,I,J) * constit(Z:H2,J,K).
constit(X:H,I,K)      += rewrite(X:H,Y:H,Z:H2) * temp6(Y:H,Z:H2,I,K).
constit(X:H,I,K)      += rewrite(X:H,Y:H2,Z:H)
                         * constit(Y:H2,I,J) * constit(Z:H,J,K).
```

We can *reuse* this definition of the temp rule to fold together the last two factors of line 3—which is the same subexpression, modulo variable renaming.

[17] Determining the optimal elimination order is NP-complete. However, there are many heuristics in the literature (such as min-width) that could be used if automatic optimization of long rules is needed.

[18] This algorithm is obviously an extension of the ordinary inside algorithm in section 6.2.2. The other rules are

```
constit(X:H,I,K) += rewrite(X,H) * word(H,I,K).
goal += constit(s:H,0,N) * length(N).
```

Given a new rule R in the form $r \oplus= F[s]$ (which will be used to replace a group of rules $R_1, \ldots, R_n$ in $\mathcal{P}$). Let $S_1, \ldots, S_n$ be the complete list of rules in $\mathcal{P}$ whose heads unify with s. Suppose that all rules in this list use $\odot$ as their aggregation operator.
Now for each i, when s is unified with the head of S_i, the tuple (r, F, s, S_i)[19] takes the form $(r_i, F_i, s_i, s_i \odot= E_i)$. Suppose that for each i, there is a distinct rule R_i in the program that is equal to $r_i \oplus= F_i[E_i]$, modulo renaming of its variables.
Then the folding transformation deletes the n rules $R_1, \ldots, R_n$, and replaces them with the new rule R, provided that

- Any variable that occurs in any of the E_i which also occurs in either F_i or r_i must also occur in s_i.[20]
- Either $\odot=$ is simply =,[21] or else the distributive property $[\![F[\mu] \oplus F[\nu]]\!] = [\![F[\mu \odot \nu]]\!]$ holds for all assignments of terms to variables and all valuation functions $[\![\cdot]\!]$.[22]

[19] Before forming this 4-tuple, rename the variables in S_i so that they do not conflict with those in r, F, s. Perform the desired unification within the 4-tuple by unifying it with the fixed term (R,F,S,S $\odot$= E), which contains two copies of S.

[20] This ensures that computing s_i by rule S_i does not sum over this variable, which would break the covariation of E_i with F or r as required by the original rule R_i.

[21] For instance, in the very first example of section 6.3, the temp item was defined using = and therefore performed no aggregation (see section 6.2.3). No distributivity was needed.

[22] That is, all valuation functions over the space of ground terms, including dummy terms μ and ν, when extended over expressions in the usual way.

FIGURE 1 The weighted folding transformation.

(Below, for clarity, we explicitly and harmlessly swap the names of H2 and H within the temp rule.)

```
temp7(Y:H2,Z:H,I,K) += constit(Y:H2,I,J) * constit(Z:H,J,K).
constit(X:H,I,K)       += rewrite(X:H,Y:H,Z:H2) * temp7(Y:H,Z:H2,I,K).
constit(X:H,I,K)       += rewrite(X:H,Y:H2,Z:H) * temp7(Y:H2,Z:H,I,K).
```

Using the same temp7 rule (modulo variable renaming) in both folding transformations, rather than introducing a new temporary item for each fold, gives us a constant-factor improvement in time and space.

Formal definition of folding. Our definition, shown in Figure 1, may seem surprisingly complicated. Its most common use is to replace a single rule $r \oplus= F[E]$ with $r \oplus= F[s]$ in the presence of a rule $s \odot= E$. However, we have given a more general form that is best understood as precisely reversing the weighted unfolding transformation to be discussed in the next section (Figure 2). In unfolding, it is often useful for s to be defined by a group of rules whose heads unify with s (i.e., they may be more general or specific patterns than s). We define folding to allow the same flexibility.

In particular, this definition of folding allows an additional use of distributivity. Both the original item r and the temp item s may aggregate values not just within a single rule (summing over free variables in the body), but also across n rules. In ordinary mathematical notation, we are performing a generalized version of the following substitution:

$$\begin{array}{lcl} \textbf{Before} & & \textbf{After} \\ r = \sum_{i=1}^{n} (f * E_i) & \Rightarrow & r = f * s \\ s = \sum_{i=1}^{n} E_i & \Rightarrow & s = \sum_{i=1}^{n} E_i \end{array}$$

given the distributive property $\sum_i (f * E_i) = f * \sum_i E_i$. The common context in the original rules is the function "multiply by expression f," so the temp item s plays the role of r/f.

Figure 1 also generalizes beyond "multiply by f." It allows an arbitrary common context F—a sort of function. In Figure 1 and throughout the paper, we use the notation $F[E]$ to denote the *literal* substitution of expression E for all instances of μ in an expression F over items, even if X contains variables that appear in E or elsewhere in the rule containing $F[E]$. We assume that μ is a distinguished symbol that does not appear elsewhere.

Generalized distributivity. Figure 1 states the required distributive property as generally as possible in terms of F. An interesting example is ⟦log(p) + log(q)⟧ = ⟦log(p * q)⟧, which says that log distributes over * and changes it to +. This means that the definition s(J) *= e(J,K) may be used to replace r += b(I,J) * log(e(J,K)) with r += b(I,J) * log(s(J)). Here $n = 1$, $F =$ b(I,J) * log(μ), $E_1 =$ e(J,K), and $s =$ s(J).

By contrast, the definition s += e(J) may *not* be used to replace r += e(J)*e(J) with r += s*s, which would incorrectly replace a sum of squares with a square of sums. If we take F to be e(J)*μ or μ*e(J), it is blocked by the first requirement in Figure 1 (variable occurrence). If we take F to be μ*μ, it is blocked by the second requirement (distributivity).

Introducing slashed definitions for folding. Notice that Figure 1 requires the rules defining the temp item s to be in the program *already* before folding occurs. If necessary, their presence may be arranged by a trivial **definition introduction** transformation that adds slash(r,F) $\odot$= E_i for each i, where slash is a new functor not used elsewhere in $\mathcal{P}$, and $\odot$ is chosen to ensure the required distributive property. We then take s to be slash(r,F) (or if one wants to use syntactic sugar, r/F).

Note that then s_i will be slash(r_i,F_i), which automatically satisfies the requirement in Figure 1 that certain variables that occur in F_i or r_i must also occur in s_i. This technique of introducing slashed items will reappear in section 6.5, where it forms a fundamental part of our speculation transformation.

Let R be a rule in $\mathcal{P}$, given in the form $r \oplus= F[s]$. Let $S_1, \ldots, S_n$ be the complete list of rules in $\mathcal{P}$ whose heads unify with s. Suppose that all rules in this list use $\odot$ as their aggregation operator.
Now for each i, when s is unified with the head of S_i, the tuple (r, F, s, S_i)[23] takes the form $(r_i, F_i, s_i, s_i \odot= E_i)$.
Then the unfolding transformation deletes the rule R, replacing it with the new rules $r_i \oplus= F_i[E_i]$ for $1 \leq i \leq n$. The transformation is allowed under the same two conditions as for the weighted folding transformation:

- Any variable that occurs in any of the E_i which also occurs in either F_i or r_i must also occur in s_i.
- Either $\odot =$ is simply $=$, or else we have the distributive property $\llbracket F[\mu] \oplus F[\nu] \rrbracket = \llbracket F[\mu \odot \nu] \rrbracket$.

[23] Before forming this tuple, rename the variables in S_i so that they do not conflict with those in r, F, s.

FIGURE 2 The weighted unfolding transformation.

If no operator $\odot$ can be found such that the distributive property will hold, and $n = 1$, then one can still use folding without the distributive property (as in the example that opened this section). In this case, introduce a rule temp(E_1) $=$ E_1, and take s to be temp(E_1), which "memoizes" the value of expression E_1. Again, this satisfies the requirements of Figure 1.

6.4 Unfolding and Rule Elimination: Inlining Subroutines

Unfolding. The inverse of the folding transformation, called **unfolding** (Figure 2), replaces s with its definition inside the rule body $r \oplus= F[s]$. This definition may comprise several rules whose heads unify with s. If s is regarded as a subroutine call, then unfolding is tantamount to inlining that call.

Recall that a *folding* transformation leaves the asymptotic runtime alone, or may improve it when combined with the distributive law. Hence *unfolding* makes the asymptotic runtime the same or worse. However, it may help the *practical* runtime by reducing overhead. (This is exactly the usual argument for inlining subroutine calls.)

An obvious example is program specialization. Consider the inside algorithm in section 6.2.2. If we take the second line,

```
constit(X,I,K) += rewrite(X,Y,Z) * constit(Y,I,J) * constit(Z,J,K).
```

and unfold rewrite(X,Y,Z) inside it, then we replace the above rule with a *set* of rules, one for each binary production of the grammar (i.e., each rule whose head unifies with rewrite(X,Y,Z)):

```
constit(s,I,K) += 1 * constit(np,I,J) * constit(vp,J,K).
constit(np,I,K) += 0.5 * constit(det,I,J) * constit(n,J,K).
```

The resulting parser is specialized to the grammar, perhaps providing a constant-time speedup. It avoids having to look up the value of rewrite(s,np,vp) or rewrite(np,det,n) since these are now "hard-coded" into specialized inference rules. A compiled version can also "hard-code" pattern matching routines against specialized patterns such as constit(np,I,J); such pattern matches are used during forward or backward chaining to determine which rules to invoke.

Note that recursive calls can also be unfolded. For example, constit is recursively defined in terms of itself. If we unfold the constit(np,I,J) inside the first of the new rules above, we get

```
constit(s,I,K) += 1 * 0.5 * constit(det,I,L) * constit(n,L,J) * constit(vp,J,K).
constit(s,I,K) += 1 * 0.4 * word("Dumbo",I,J) * constit(vp,J,K).
constit(s,I,K) += 1 * 0.1 * word("flies",I,J) * constit(vp,J,K).
```

Unfolding is often a precursor to other transformations. For example, the pattern constit(vp,I,J) above can now be transformed to constit_vp(I,J) for more efficient storage. Furthermore, constant subexpressions like 1 * 0.5 can now be replaced in the source code by their values—a transformation that is known, not coincidentally, as constant folding. We will see another useful example of this unfold-refold pattern below, and yet another when we derive the (Eisner and Satta, 1999) algorithm in section 6.5.1.

Rule elimination. A practically useful transformation that is closely related to unfolding is what we call **rule elimination** (Figure 3). Rather than fully expanding one call to subroutine s, it removes one of the defining clauses of s and requires *all* of its callers to do the work formerly done by that clause.

This may change or eliminate the definition of s, so the transformation is not semantics-preserving. The advantage of changing the semantics is that if some s items become no longer provable, then it is no longer necessary to store them in the chart.[26] Thus, *rule elimination saves space*. It also shares the advantages of unfolding—it can specialize a program, move unification to compile-time, eliminate intermediate steps, and serve as a precursor to other

[26] A similar space savings—while preserving semantics—could be arranged simply by electing not to memoize these items, so that they are computed on demand rather than stored. Indeed, if we extend our formalism so that a program can specify what to memoize, then it is not hard to combine folding and unfolding to define a transformation that acts just like rule elimination (in that the callers are specialized) and yet preserves semantics. The basic idea is to fold together all of the *other* clauses that define s, then unfold all calls to s (which accomplishes the specialization), and finally declare that s (which is no longer called) should not be memoized (which accomplishes the space savings). However, we suppress the details as beyond the scope of this paper. Our main interest in rule elimination in this paper is to eliminate rules for temp items, whose semantics were introduced by a previous transformation and need not be preserved.

Let S be a rule of $\mathcal{P}$ to eliminate, with head s. Let $R_1, R_2, \ldots R_n$ be a complete list of rules in $\mathcal{P}$ whose bodies may depend on s.[24] Suppose that each R_i can be expressed in the form $r_i \oplus_i\!= F_i[s_i]$, where s_i is a term that unifies with s and F_i is an expression that is independent of s.[25]

For each i, when s_i is unified with the head of S, the tuple (r_i, F_i, s_i, S) takes the form $(r'_i, F'_i, s'_i, s'_i \odot\!= E'_i)$. Then the rule elimination transform removes rule S from $\mathcal{P}$ and, for each i, adds the new rule $r'_i \oplus\!= F'_i[E'_i]$ (while also retaining R_i). The transformation is allowed under the same two conditions as for weighted folding and unfolding:

- Any variable that occurs in any of the E'_i which also occurs in either F'_i or r'_i must also occur in s'_i.
- Either $\odot =$ is simply $=$, or else we have the distributive property $[\![F[\mu] \oplus F[\nu]]\!] = [\![F[\mu \odot \nu]]\!]$.

Warning: This transformation alters the semantics of ground terms that unify with s.

FIGURE 3 The weighted rule elimination transformation.

transformations.

To see the difference between rule elimination and unfolding, let us start with the same example as before, and selectively eliminate just the single binary production rewrite(np,det,n) = 0.5. In contrast to unfolding, this no longer replaces the original general rule

```
constit(X,I,K) += rewrite(X,Y,Z) * constit(Y,I,J) * constit(Z,J,K).
```

with a slew of specialized rules. Rather, it *keeps* the general rule but adds a specialization

```
constit(np,I,K) += 0.5 * constit(det,I,J) * constit(n,J,K).
```

while deleting rewrite(np,det,n) = 0.5 so that it does not *also* feed into the general rule.

A recursive example of rule elimination. An interesting recursive example is shown below. The original program is in the first column. Eliminating

[24] That is, the bodies of all other rules in $\mathcal{P}$ must be independent of s. The notion of independence relies on the semantics of expressions, not on the particular program $\mathcal{P}$. An expression E is said to be **independent** of a term s if for any two valuation functions on ground terms that differ only in the values assigned to groundings of s, the extensions of these valuation functions over expressions assign the same values to all groundings of E.

[25] For example, suppose s is s(X,X). Then the rule r(X) += s(X,Y) * t(Y) should be expressed as r(X) += (μ * t(Y))[s(X,Y)], while r(X) min= s(X,Y) * s(X,Y) should be expressed as r(X) += (μ * μ)[s(X,Y)] and r min= 3 can be expressed as r min= 3[s(X,X)]. However, r(X) += s(X,Y) * s(Y,Z) cannot be expressed in the required form at all. We regard μ as a ground term in considering whether F_i is independent of s.

its second or third rule gives the program in the second or third column, respectively. Each of these changes damages the semantics of s, as warned, but preserves the value of r.[27]

s += 1.		s += 1.

s += 0.5*s.	s += 0.5*s.
	s += 0.5*1.
		s += 0.5*0.5*s.

r += s.	r += s.
	r += 1.
		r += s.
		r += 0.5*s.

$[\![\mathsf{s}]\!] = 2, [\![\mathsf{r}]\!] = 2$	$[\![\mathsf{s}]\!] = 1, [\![\mathsf{r}]\!] = 2$	$[\![\mathsf{s}]\!] = \frac{4}{3}, [\![\mathsf{r}]\!] = 2$

Unfolding or rule elimination followed by folding.[28] Recall the bilexical CKY parser given near the end of section 6.3. The first rule originally shown there has runtime $O(N^3 \cdot n^5)$, since there are N possibilities for each of X,Y,Z and n possibilities for each of I,J,K,H,H2. Suppose that instead of that slow rule, the original programmer had written the following folded version:

```
temp8(X:H,Z:H2,I,J) += rewrite(X:H,Y:H,Z:H2) * constit(Y:H,I,J).
constit(X:H,I,K)       += temp8(X:H,Z:H2,I,J)    * constit(Z:H2,J,K).
```

This partial program has asymptotic runtime $O\left(N^3 \cdot n^4 + N^2 \cdot n^5\right)$. and needs $O\left(N^2 \cdot n^4\right)$ space to store the items (rule heads) it derives.

By either unfolding the call to temp8 or eliminating the temp8 rule, we recover the first rule of the original program:

```
constit(X:H,I,K) += rewrite(X:H,Y:H,Z:H2)
                          * constit(Y:H,I,J) * constit(Z:H2,J,K).
```

This worsens the time complexity to $O\left(N^3 \cdot n^5\right)$. The payoff is that now we can refold this rule differently—either as follows (Eisner and Satta, 1999),

```
temp9(X:H,Y:H,J,K) += rewrite(X:H,Y:H,Z:H2) * constit(Z:H2,J,K).
constit(X:H,I,K)       += temp9(X:H,Y:H,J,K)    * constit(Y:H,I,J).
```

or alternatively as already shown in section 6.3 (whose temp item had the additional advantage of being reusable). Either way, the asymptotic time complexity is now $O\left(N^3 \cdot n^4 + N^2 \cdot n^4\right)$—better than the original programmer's version.

How about the asymptotic space complexity? If the first step used rule elimination rather than unfolding, then it actually eliminated storage for the temp8 items, reducing the storage requirements from $O\left(N^2 \cdot n^4\right)$ to $O\left(N \cdot n^3\right)$.

[27] Since r was defined to equal the (original) value of s, it provides a way to recover the original semantics of s. Compare the similar construction sketched in footnote 26.

[28] Rule elimination can also be used *after* another transformation, such as speculation, to clean away unnecessary temp items. See footnote 42.

Regardless, the refolding step increased the space complexity back to the original programmer's $O\left(N^2 \cdot n^4\right)$.

6.5 Speculation: Factoring Out Chains of Computation

In the most important contribution of this paper, we now generalize folding to handle unbounded sequences of rules, including cycles. This **speculation** transformation, which is novel as far as we know, is reminiscent of gap-passing in categorial grammar. It has many uses; we limit ourselves to two real-world examples.

6.5.1 Examples of the Speculation Transformation

Unary rule closure. Unary rule closure is a standard optimization on context-free grammars, including probabilistic ones (Stolcke, 1995). We derive it here as an instance of speculation. Suppose we begin with a version of the inside algorithm that allows unary nonterminal rules as well as the usual binary ones:

```
constit(X,I,K) += rewrite(X,W)   * word(W,I,K).
constit(X,I,K) += rewrite(X,Y)   * constit(Y,I,K).
constit(X,I,K) += rewrite(X,Y,Z) * constit(Y,I,J) * constit(Z,J,K).
```

Suppose that the grammar includes a unary rule cycle. For example, suppose that rewrite(np,s) and rewrite(s,np) are both provable. Then the values of constit(np,I,K) and constit(s,I,K) "feed into each other." Under forward chaining, updating either one will cause the other to be updated; this process repeats until numerical convergence.[29]

This computation is somewhat time-consuming—yet it is essentially the same for every constit(np,I,K) we may start with, regardless of the particular span I–K or the particular input sentence. We would prefer to do the computation only once, "offline."

A difficulty is that the computation does incorporate the particular real value of the constit(np,I,K) that we start with. However, if we simply ignore this factor during our "offline" computation, we can multiply it in later when we get an actual constit(np,I,K). That is, we compute *speculatively* before the particular constit(np,I,K) and its value actually arrive.

In the transformed code below, temp(X,X0) represents the inside probability of building up a constit(X,I0,K0) from a constit(X0,I0,K0) by a sequence of 0 or more unary rules. In other words, it is the total probability of all (possibly empty) unary-rewrite chains X $\rightarrow^*$ X0. While line 2 of the transformed

[29] If we gave the Viterbi algorithm instead, with max= in place of +=, then convergence would occur in finite time (at least for a PCFG, where all rewrite items have values in [0, 1]). The same algorithm applies.

program still computes these items by numerical iteration, it only needs to compute them once for each X,X0, since they are now independent of the particular span I0–K0 covered by these two constituents.[30]

```
temp(X0,X0) += 1.
temp(X,X0) += rewrite(X,Y) * temp(Y,X0).
other(constit(X,I,K)) += rewrite(X,W)   * word(W,I,K).
other(constit(X,I,K)) += rewrite(X,Y,Z) * constit(Y,I,J) * constit(Z,J,K).
constit(X,I,K) += temp(X,X0)*other(constit(X0,I,K)).
```

The temp(s,np) item sums the probabilities of the infinitely many unary-rewrite chains s $\rightarrow^*$ np, which build np up into s using only line 2 of the original program. Now, to get values like constit(s,4,6) for a particular input sentence, we can simply sum finite products like temp(s,np) * other(constit(np,4,6)), where other(constit(np,4,6)) sums up ways of building an constit(np,4,6) *other than* by line 2 of the original program.[31]

The semantics of this program, which can derive non-ground terms. fully defined by section 6.2.3. We omit further discussion of how it executes directly under forward chaining (section 6.2.4). However, note that the program could be transformed into a more conventional dynamic program by applying rule elimination to the first rule (the only one that is not range-restricted).[32]

For efficiency, our formal transformation adds a “filter clause” to each of the temp rules:

```
temp(X0,X0) += 1 needed_only_if constit(X0,I0,K0).
temp(X,X0)   += rewrite(X,Y)*temp(Y,X0) needed_only_if constit(X0,I0,K0).
```

The exact meaning of this clause will be discussed in section 6.5.2. It permits laziness, so that we compute portions of the unary rule closure only when they will be needed. Its effect is that for each nonterminal X0, the temp items

[30] We remark that the first n steps of this iterative computation could be moved to compile time, by eliminating line 1 as discussed below, specializing line 2 to the grammar by unfolding rewrite(X,Y), and then computing the series sums by alternately unfolding the temp items and performing constant folding to consolidate each temp item’s summands.

[31] For example, it includes derivations where the np is built from a determiner and a noun, or directly from a word, but not where it is built directly from some s or another np. Excluding the last option prevents double-counting of derivations.

[32] Here is the result, which alters the semantics of the slashed temp item to ignore derivations of length 0:

```
temp(X,X0) += rewrite(X,Y) * temp(Y,X0).
temp(X,X0) += rewrite(X,X0) * 1.
other(constit(X,I,K)) += rewrite(X,W)   * word(W,I,K).
other(constit(X,I,K)) += rewrite(X,Y,Z) * constit(Y,I,J) * constit(Z,J,K).
constit(X,I,K) += temp(X,X0)*other(constit(X0,I,K)).
constit(X0,I,K) += 1*other(constit(X0,I,K)).
```

are proved or updated only once some constit(X0,I0,K0) constituent has been built.[33] At that time, all the temp(X,X0) values for this X0 will be computed once and for all, since there is now something for them to combine with. Usefully, these values will then remain static while the grammar does, even if the sentence changes.

Adopting the categorial view we introduced in section 6.3, we can regard temp(s,np) as merely an abbreviation for the *non-ground* slashed item constit(s,I0,K0)/constit(np,I0,K0): the cost of building up a constit(s,I0,K0) if we already had a constit(np,I0,K0). This cost is independent of I0 and K0, which is why we only need to compute a single item to hold it, albeit one that contains variables.

As we will see, the slashed notation is not merely expository. Our formal speculation transformation will actually produce a program with slashed terms, essentially as follows:

```
constit(X0,I0,K0)/constit(X0,I0,K0) += 1.
constit(X,I,K)/constit(X0,I0,K0)    += rewrite(X,Y)
                                        * constit(Y,I,K)/constit(X0,I0,K0).
other(constit(X,I,K))               += rewrite(X,W) * word(W,I,K).
other(constit(X,I,K))               += rewrite(X,Y,Z)
                                        * constit(Y,I,J) * constit(Z,J,K).
constit(X,I,K)                      += (constit(X,I,K)/constit(X0,I0,K0))
                                        * other(constit(X0,I0,K0)).
```

A variable-free example. To understand better how the slash and other mechanisms work, consider this artificial variable-free program, illustrated by the hypergraph in Figure 4:

```
a += b * c.
b += r.
c += f * c.
c += d * e * x.
c += g.
x += . . .
```

The values of a and c depend on x. We elect to create speculative versions of the first, third, and fourth rules. The resulting program is drawn in Figure 5. It includes rules to compute slashed versions of a, c and x itself that are "missing an x":

```
a/x += b * c/x.
c/x += f * c/x.
c/x += d * e * x/x.
x/x += 1.
```

[33] In this example, the filter clause on the second rule is redundant. Runtime analysis or static analysis could determine that it has no actual filtering effect, allowing us to drop it.

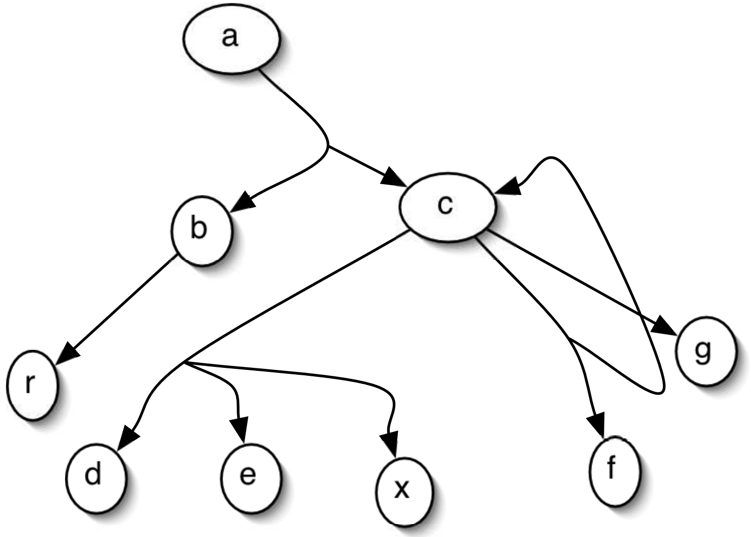

FIGURE 4 A simple variable-free program before applying the speculation transformation.

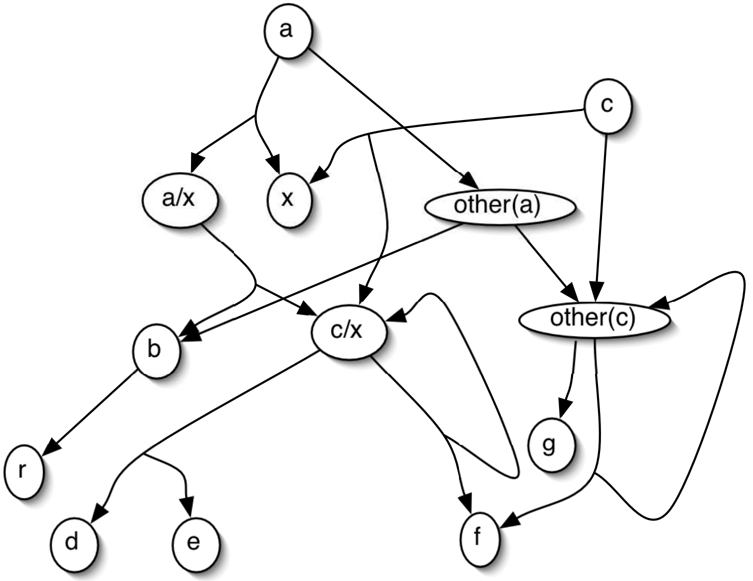

FIGURE 5 The program of Figure 4 after applying the speculation transformation. The x/x rule and various other(...) rules have been eliminated for simplicity.

It also reconstitutes full-fledged versions of a, c, and x. Each is defined by a sum that is split into two cases: summands that were built from an x using a sequence of ≥ 0 of the selected rules, and "other" summands that were not. (Notice that the first rule is *not* a += a/x * x; this is because x might in general include derivations that are built from another x (though not in this example), and this would lead to double-counting. By using a += a/x * other(x), we split each derivation of a uniquely into a maximal sequence of selected rules, applied to a minimal instance of x.)

```
a += a/x * other(x).
c += c/x * other(x).
x += x/x * other(x).
a += other(a).
c += other(c).
```

Finally, the program must define the "other" summands:

```
other(a) += b * other(c).
other(c) += f * other(c).
other(c) += g.
other(x) += . . .
```

In Figure 5, this program has been further simplified by eliminating the rules for x/x and other(x).

Split bilexical grammars. For our next example, consider a "split" bilexical CFG, in which a head word must combine with all of its right children before any of its left children. The naive algorithm for bilexical context-free parsing is $O(n^5)$. In the split case, we will show how to derive the $O(n^4)$ and $O(n^3)$ algorithms of Eisner and Satta (1999).

The "inside algorithm" below[34] builds up rconstit items by successively adding 0 or more child constituents to the right of a word, then builds up constit items by adding 0 or more child constituents to the left of this rconstit. As before, X:H represents a nonterminal X whose head word is H.

```
rconstit(X:H,I,K) += rewrite(X,H) * word(H,I,K).  % 0 right children so far
rconstit(X:H,I,K) += rewrite(X:H,Y:H,Z:H2)        % add right child
                     * rconstit(Y:H,I,J) * constit(Z:H2,J,K).
constit(X:H,I,K)  += rconstit(X:H,I,K).           % 0 left children so far
constit(X:H,I,K)  += rewrite(X:H,Y:H2,Z:H)        % add left child
                     * constit(Y:H2,I,J) * constit(Z:H,J,K).
goal += constit(s:H,0,N) * length(N).
```

[34] We deal here with context-free grammars, rather than the head-automaton grammars of Eisner and Satta. In particular, our complete constituents carry nonterminal categories and not just head words. Note that the algorithm is correct only for a "split" grammar (formally, one that does not contain two rules of the form rewrite(X:H1,Y:H2,Z:H1) and rewrite(V:H1,X:H1,W:H3)), since otherwise the grammar would license trees that cannot be constructed by the algorithm.

This obvious algorithm has runtime $O(N^3 \cdot n^5)$ (dominated by line 4). We now speed it up by exploiting the conditional independence of left children from right children. To build up a constit whose head word starts at I, we will no longer start with a *particular*, existing rconstit from I to K (line 3) and then add left children (line 4). Rather, we transform the program so that it abstracts away the choice of starting rconstit. It can then build up the constit item *speculatively*, adding left children without having committed to any particular rconstit. As this work is independent of the rconstit, the items derived during it do not have to specify any value for K. Thus, work is shared across all values of K, improving the asymptotic complexity. Only after finishing this speculative computation does the program fill in each of the various rconstit items that could have been chosen at the start. To accomplish this transformation, replace lines 3–4 with

```
lconstit(X0:H0,X0,J0,J0) += 1.
lconstit(X:H0,X0,I,J0)     += rewrite(X:H0,Y:H2,Z:H)
                                 * constit(Y:H2,I,J) * lconstit(Z:H0,X0,J,J0).
constit(X:H0,I,K0)          += lconstit(X:H0,X0,I,J0) * rconstit(X0:H0,J0,K0).
```

The new temp item lconstit(X:H0,X0,I,J0) represents the *left* half of a constituent, stretching from I to J0. We can regard it again in categorial terms: as the last line suggests, it is just a more compact notation for a constit missing its rconstit right half from J0 to some K0. This can be written more perspicuously as constit(X:H0,I,K0)/rconstit(X0:H0,J0,K0), where K0 is always a free variable, so that lconstit need not specify any particular value for K0.

The first lconstit rule introduces an empty left half. This is extended with its left children by recursing through the second lconstit rule, allowing X and I to diverge from X0 and J0 respectively. Finally, the last rule fills in the missing right half rconstit.

Again, our speculation transformation will actually produce the slashed notation as its output. Specifically, it will replace lines 3–4 of the original untransformed program with the following.[35]

```
rconstit(X0:H0,J0,K0)/rconstit(X0:H0,J0,K0) += 1
                              needed_only_if  rconstit(X0:H0,J0,K0).

constit(X:H,J,K)/rconstit(X0:H0,J0,K0)
                              += rconstit(X:H,J,K)/rconstit(X0:H0,J0,K0)
                              needed_only_if  rconstit(X0:H0,J0,K0).
```

[35] In fact our transformation in Figure 6 will produce something a bit more complicated. The version shown has been simplified by using rule elimination (section 6.4) to trim away all other(...) items, which do not play a significant role in this example. That is because the only slashed items are constit/rconstit, and there is no other way to build a constit except from an rconstit.

```
constit(X:H,I,K)/rconstit(X0:H0,J0,K0)
                         += rewrite(X:H,Y:H2,Z:H) * constit(Y:H2,I,J)
                            * constit(Z:H,J,K)/rconstit(X0:H0,J0,K0)
                         needed_only_if  rconstit(X0:H0,J0,K0).
constit(X:H,I,K)         += constit(X:H,I,K)/rconstit(X0:H0,J0,K0)
                            * rconstit(X0:H0,J0,K0).
```

The first line introduces a slashed item. The next two lines are the result of slashing rconstit(X0:H0,J0,K0) out of the original lines 3–4; note that X0, H0, J0, and K0 appeared nowhere in the original program. The final line reconstitutes the constit item defined by the original program, so that the transformed program preserves the original program's semantics.

By inspecting this program, one can see that the only provable items of the form constit(X:H,I,K)/rconstit(X0:H0,J0,K0) actually have H=H0, K=K0, and K0 a free variable.[36] These conditions are true for the slashed item that is introduced in the first line, and they are then preserved by every rule that derives a new slashed item. This is why in our earlier presentation of this code, we were able to abbreviate such a slashed item by lconstit(X:H0,X0,I,J0), which uses only 5 variables rather than 8. Discovering such abbreviations by static analysis is itself a transformation that we do not investigate in this paper.

Filter clauses can improve asymptotic runtime. The special filter clause needed_only_if rconstit(X0:H0,J0,K0) is added solely for efficiency, as always. It says that it is not necessary to build a left half that might be useless (i.e., purely speculatively), but only when there is at least one right half for it to combine with.

In this example, the filter clause is subtly responsible for avoiding an extra factor of V in the runtime, where $V \gg n$ is the size of the vocabulary. For simplicity, let us return to the unslashed notation:

```
lconstit(X:H0,X0,I,J0) += rewrite(X:H0,Y:H2,Z:H)
                              * constit(Y:H2,I,J) * lconstit(Z:H0,X0,J,J0).
                          needed_only_if  rconstit(X0:H0,J0,K0).
```

The intent is to build only left halves lconstit(H:H0,X0,I,J0) whose head word H0 will actually be found starting at the right edge J0. However, without the filter, the above rule could be far more speculative, combining a finished left child such as constit(np:dumbo,4,5) with a rewrite rule such as rewrite(s:flies,np:dumbo,vp:flies) and the non-ground item lconstit(X0:H0,X0,J0,J0) (defined elsewhere with value 1) to obtain lconstit(s:flies,vp,4,5)—regardless of whether flies actually starts at position 5 or even appears in the input sentence at all! This would lead to a proliferation of

[36] By contrast, we already noted that X and I could diverge from X0 and J0 respectively, in this particular program.

$O(V)$ lconstit items with speculative head words such as flies that *might* start at position 5. The filter clause prevents this by "looking ahead" to see whether any items of the form rconstit(vp:flies,5,K0) have actually been proved.

As a result, the runtime is now $O\left(n^4\right)$ (as compared to $O\left(n^5\right)$ for the untransformed program).[37] This is so because the rule above may be grounded in $O\left(n^4\right)$ ways reflecting different bindings of I, J, J0, and word H2, where H2 may in practice be any of the words in the span I–J. Although the rule also mentions H0, the filter clause has ensured that H0's binding is completely determined by J0's.

As a bonus, we can now apply the unfold-refold pattern to obtain the $O\left(n^3\right)$ algorithm of Eisner and Satta (1999). Starting with our transformed program, unfold constit in the body of each rule where it appears,[38] giving

```
rconstit += rconstit * rewrite * (lconstit′ * rconstit′).
lconstit += (lconstit′ * rconstit′) * rewrite * lconstit.
```

where the ′ symbol marks the halves of the unfolded constit, and the three adjacent half-constituents are written in left-to-right order. Now re-parenthesize these rules as

```
rconstit += (rconstit * (rewrite * lconstit′)) * rconstit′.
lconstit += lconstit′ * ((rconstit′ * rewrite) * lconstit).
```

and fold out each parenthesized subexpression, using distributivity to sum over its free variables. The items introduced when folding the large subexpressions correspond, respectively, to Eisner and Satta's "right trapezoid" and "left trapezoid" items. The speedup arises because there are $O(n)$ fewer possible trapezoids than constits: a constit had to specify a head word that could be any of the words covered by the constit, but a trapezoid's head word is always the word at its left or right edge.

6.5.2 Semantics and Operation of Filter Clauses

Our approach to filtering is novel. Our needed_only_if clauses may be regarded as "relaxed" versions of side conditions (Goodman, 1999). In the denotational semantics (section 6.2.3), they relax the restrictions on the valuation function, allowing more possible valuations for the transformed program. (In the case of speculation, these valuations may disagree on the new slashed items, but all of them preserve the semantics of the original program.)

Specifically, when constructing $\mathcal{P}(r)$ to determine whether a ground item r is provable and what its value is, we may *optionally* omit the summand cor-

[37] We could also have achieved $O\left(n^4\right)$ simply by folding the original program as discussed in section 6.3. However, that would not have allowed the further reduction to $O\left(n^3\right)$ discussed below.

[38] Including if desired the goal rule, not discussed here. The old constit rule is then useless, except perhaps to the user, and may be trimmed away if desired by rule elimination.

responding to a grounded rule $r \oplus_r = E$ if this rule has an attached filter clause needed_only_if C such that no consistent grounding of C has been proved.[39]

How does this help operationally, in the forward chaining algorithm? When a rule triggers an update to a ground *or* non-ground item, but carries a (partly instantiated) filter clause that does not unify with any proved item, then the update has infinitely low priority and need not be propagated further by forward chaining. The update must still be carried out if the filter clause is proved later.

The optionality of the filter is crucial for two reasons. First, if a filter becomes false, the forward-chaining algorithm is not required to retract updates that were performed when the filter was true. In the examples of section 6.5.1 above or section 6.6.3 below, the filter clauses ensure that entries are filled into the unary-rule-closure and left-corner tables only as needed. Once this work has been done, however, these entries are allowed to persist even when they are no longer needed, e.g., once the facts describing the input sentence are retracted. This means that we can reuse them for a future sentence rather than re-deriving them every time they are needed.

Second, the forward-chaining algorithm is not required to bind variables in the rule when it checks for consistent groundings of the filter clause. Consider this rule from earlier:

```
constit(X:H,I,K)/rconstit(X0:H0,J0,K0)
      += rewrite(X:H,Y:H2,Z:H) * constit(Y:H2,I,J)
          * constit(Z:H,J,K)/rconstit(X0:H0,J0,K0)
      needed_only_if  rconstit(X0:H0,J0,K0).
```

Recall that the only constit/rconstit items that are actually derived are non-ground items in which K=K0 and is free, such as constit(s:flies,4,K0)/ rconstit(vp:flies,5,K0). Such a non-ground item actually represents an infinite collection of possible ground items that specialize it. The semantics of needed_only_if, which are defined over ground terms, say that we only need to derive a subset of this collection: rather than proving the non-ground item above, we are welcome to prove only the "needed" ground instantiations, with specific values of K0 such that rconstit(vp:flies,5,K0) has been proved. However, in general, this would prove $O(n)$ ground items rather than a single non-ground item. It would destroy the whole point of speculation, which is to achieve a speedup by leaving K0 free until specific rconstit items are multiplied back in at the end. Thus, the forward-chaining algorithm is better off exploiting the optionality of filtering and proving the non-ground version—thus

[39] The "consistent" groundings are those in which variables of C that are shared with r or E are instantiated accordingly. In the speculation transformation, all variables of C are in fact shared with r and E. If they were not, C could have many consistent groundings, but we would still aggregate only one copy of E, just as if the filter clause were absent, not one per copy per consistent grounding.

proving more than is strictly needed—as long as at least one of its groundings is needed (i.e., as long as some item that unifies with rconstit(vp:flies,4,K0) has already been proved).

For a simpler example, consider

rconstit($\underline{\mathsf{X0}}$:$\underline{\mathsf{H0}}$,$\underline{\mathsf{J0}}$,$\underline{\mathsf{K0}}$)/rconstit($\underline{\mathsf{X0}}$:$\underline{\mathsf{H0}}$,$\underline{\mathsf{J0}}$,$\underline{\mathsf{K0}}$) += 1
needed_only_if rconstit(X0:H0,J0,K0).

A reasonable implementation of forward chaining will prove the non-ground item rconstit(X0:H0,J0,K0)/rconstit(X0:H0,J0,K0)—just as if the filter clause were not present—provided that *some* grounding of rconstit(X0:H0,J0,K0) has been proved. It is not required to derive a separate grounding of the slashed item for *each* grounding of rconstit(X0:H0,J0,K0); this would also be correct but would be less efficient.

6.5.3 Formalizing Speculation for Semiring-Weighted Program Fragments

To formalize the speculation transformation, we begin with a useful common case that suffices to handle our previous examples. This definition (Figure 6) allows us to speculate over a set of rules that manipulate values in a semiring of weights W. Such rules must all use the same aggregation operator, which we call $\oplus$=, with identity element $\bar{0}$. Furthermore, each rule body must be a simple product of one or more items, using an associative binary operator $\otimes$ that distributes over $\oplus$ and has an identity element $\bar{1}$. This version of the transformation is only able to slash the final term in such a product; however, if $\otimes$ is commutative, then the terms in a product can be reordered to make any term the final term.

Our previous examples of speculation can be handled by taking the semiring $(\mathcal{W}, \oplus, \otimes, \bar{0}, \bar{1})$ to be $(\mathbb{R}, +, *, 0, 1)$. Moreover, any unweighted program can be handled by taking $(\mathcal{W}, \oplus, \otimes, \bar{0}, \bar{1})$ to be ({F, T}, |, &, F, T).

Intuitively, other(A) in Figure 6 accumulates ways of building A other than groundings of $F_{i_1} \otimes F_{i_2} \otimes \cdots \otimes F_{i_j} \otimes x$ for $j > 0$. Meanwhile, slash(A,x) accumulates ways of building A by grounding products of the form $F_{i_1} \otimes F_{i_2} \otimes \cdots \otimes F_{i_j}$ for $j \geq 0$. To "fill in the gap" and recover the final value of A (as is required to preserve semantics), we multiply slash(A,x) only by other(x) (rather than by x), in order to prevent double-counting (which is analogous to spurious ambiguity in a categorial grammar).

To apply our formal transformation in the unary-rule elimination example, take x=constit(X0,I0,K0). As required, X0,I0,K0 do not appear in the original program. Take R_1 to be the "unary" constit rule of the original program where t_1 is the last item in the body of R_i. Here $k = 0$.

To apply our formal specification in the artificial variable-free example of Figures 4–5, take x = x, R_1 = a += b * c, R_2 = c += f * c, and R_3 = c += (d * e) * x. Since, among the t_i, only t_3 unifies with x, we have $k = 2$.

Given a semiring $(\mathcal{W}, \oplus, \otimes, \bar{0}, \bar{1})$.

Given a term x to slash out, where any variables in x do not occur anywhere in the program $\mathcal{P}$. Given distinct rules $R_1, \ldots, R_n$ in $\mathcal{P}$ from which to simultaneously slash x out, where each R_i has the form $r_i \oplus\!= F_i \otimes t_i$ for some expression F_i (which may be $\bar{1}$) and some item t_i.

Let k be the index[40] such that $0 \leq k \leq n$ and

- For $i \leq k$, t_i does not unify with x.
- For $i > k$, t_i unifies with x; moreover, their unification equals t_i.[41]

Then the speculation transformation constructs the following new program. Recall that $\oplus_r$ denotes the aggregation operator for r (which may or may not be $\oplus$). Let slash, other, and matches_x be new functors that do not already appear in $\mathcal{P}$.

- $\mathsf{slash}(x,x) \oplus_x\!= \bar{1}$ needed_only_if x.
- $(\forall 1 \leq i \leq n)$ $\mathsf{slash}(r_i, x) \ \oplus\!= F_i \otimes \mathsf{slash}(t_i, x)$ needed_only_if x.
- $(\forall 1 \leq i \leq k)$ $\mathsf{other}(r_i) \ \oplus\!= F_i \otimes \mathsf{other}(t_i)$.
- ($\forall$ rules $p \oplus_p\!= E$ not among the R_i) $\mathsf{other}(p) \oplus_p\!= E$.[42]
- $\mathsf{matches\_x}(x)$ |= true.
- matches_x(A) |= false.
- A $\oplus_{\mathsf{A}}$= other(A) if not matches_x(A).[43]
- A $\oplus_{\mathsf{A}}$= $\mathsf{slash}(\mathsf{A},x) \otimes \mathsf{other}(x)$.

[40] If necessary, the program can be pre-processed so that such an index exists. Any rule can be split into three more specialized rules: an $i \leq k$ rule, an $i > k$ rule, and a rule not among the R_i. Some of these rules may require boolean side conditions to restrict their applicability.

[41] That is, t_i is "more specific" than x: it matches a non-empty subset of the ground terms that x does.

[42] It is often worth following the speculation transformation with a rule elimination transformation (section 6.4) that removes some of these other items. In particular, if p does not unify with x or any of the t_i, then the only rule that uses $\mathsf{other}(p)$ is A $\oplus_{\mathsf{A}}$ = other(A). In this case, eliminating the old rule $\mathsf{other}(p) \oplus_p\!= E$ simply restores the original rule $p \oplus_p\!= E$.

[43] Note that A ranges over all ground terms. (Except those that unify with x, which are covered by the next rule.) The aggregation into a particular ground term A must be handled using the appropriate aggregation operator for that ground term, here denoted $\oplus_{\mathsf{A}}$=. ($\oplus_x$= was similarly used above.) In the example programs, this awkward notation was avoided by splitting this rule into several rules, which handle various *disjoint* classes of items A that have different aggregation operators.

FIGURE 6 The semiring-weighted speculation transformation.

To apply our formal transformation in the split bilexical grammar example, take x=rconstit(X0:H0,J0,K0), the R_i to be the two rules defining constit, each t_i to be the last item in the body of R_i, and $k = 1$.

Folding as a special case of speculation. As was mentioned earlier, the folding transformation is a special case of the speculation transformation, in which application is restricted to rules with a single ground term at their head, and the item to be slashed out must appear literally in each affected rule. For ease of presentation, however, the formulations above are not quite parallel. In folding, we adopt the convention that a common function F is being "slashed out" of a set of rules, and the different items to which that function applies are aggregated first into a new intermediate item. In speculation, we take the opposite view, where there is a common item to be slashed out present as the argument to different functions, so that the functions get aggregated into a new lambda term. We chose the former presentation for folding to avoid the needless complication of using the lambda calculus, but we needed the flexibility of the latter for a fully general version of speculation. In the case of ordinary semiring-weighted programs , this distinction is trivial; when slashing out item a from a rule like f += a * b, we can equally easily say that we are slashing out the function "multiply by a" from its argument b or that we are slashing out the item a from inside the function "multiply by b". However, in general, being able to leave behind functions allows us to construct intermediate terms which don't carry a numerical value; for example, we could choose to slash out the a item from a rule like f += log(a) and propagate just the function slash(f, a) += λx log(x).

6.5.4 Formalizing Speculation for Arbitrary Weighted Logic Programs

The speculation transformation becomes much more complicated when it is not restricted to semiring-weighted programs. In general, the value of a slashed item is a *function*, just like the semantics of a slashed constituent in categorial grammar. Functions are aggregated pointwise: that is, we define $(\lambda z.\ f(z)) \oplus (\lambda z.\ g(z)) = \lambda z.\ (f(z) \oplus g(z))$.

As in categorial grammar semantics, gaps are introduced with the identity function, passed with function composition, and eliminated with function application.

In the commutative semiring-weighted programs discussed above, all functions had the form "multiply by w" for some weight w. We were able to avoid the lambda-calculus there by representing such a function simply as w, and by using $\bar{1}$ for the identity function, semiring multiplication $\otimes$ for both composition and application, and semiring addition $\oplus$ for pointwise addition.

We defer the details of the formal transformation to a later paper. It is significantly more complicated than Figure 6 because we can no longer rely on the mathematical properties of the semiring. As in folding and unfolding (Figures 1–2), we must demand a kind of distributive-law property to ensure that the semantics will be preserved (recall the log example from section 6.3). This property is harder to express for speculation, which is like folding through unbounded sequences of rules, including cycles.

Consider the semiring-weighted program in Figures 4–5. The original program only used the item x early in the computation, multiplying it by d * e.

The transformed program had to reconstitute a from a/x and x (and c from c/x and x). This meant multiplying x in later, only after the original d * e had passed through several levels of * and += in the rule a += b * c and the cyclic rule c += f * c.

In general, we want to be sure that delaying the introduction of x until after several intermediate functions and aggregations does not change the value of the result. Hence, a version of the distributive property must be enforced at *each* intermediate rule affecting the slashed items.

Furthermore, if the slashed-out item x contains variables, then introducing it will aggregate over those variables. For example, the rule a(B,C) += (a(B,C)/x(B0,C0,D0))(x(B0,C0,D0)) not only applies a function to an argument, but also aggregates over B0, C0, and D0. In the original version of the program, these aggregations might have been performed with various operators. When a(B,C) is reconstituted in the transformed version, we must ensure that the *same* sequence of aggregations is observed. In order to do this correctly, it is necessary to keep track of the association between the variables being aggregated in the original program and the variables in the slashed item, so that we can ensure that the same aggregations are performed.

6.6 Magic Templates: Filtering Useless Items

The bottom-up "forward-chaining" execution strategy of section 6.2.4 will (if it converges) compute values for all provable items. Typically, however, the user is primarily interested in certain items (often just goal) and perhaps the other items involved in deriving them.

In parsing, for example, the user may not care about building all legal constituents—only those that participate in a full parse. Similarly, in a program produced by speculation, the user does not care about building all possible slashed items r/x—only those that can ultimately combine with some actual x to reconstitute an item r of the original program.

Can we prevent forward chaining from building at least some of the "useless" items? In the speculation transformation (section 6.5 and Figure 6), we accomplished this by filtering our derivations with needed_only_if clauses.

We now give a transformation that explains and generalizes this strategy. It prevents the generation of some "useless" items by automatically adding needed_only_if filter clauses to an existing program. A version of this **magic templates** transformation was originally presented in a well-known paper by Ramakrishnan (1991), generalizing an earlier transformation called magic sets.

6.6.1 An Overview of the Transformation

Since this transformation makes some terms unprovable, it cannot be semantics-preserving. We will say that a ground term a is **charmed** if we *should* preserve its semantics.[44] In other words, the semantic valuation functions of the transformed program and the original program will agree on at least the charmed terms. The program will determine at runtime which terms are charmed: a ground term *a* is considered charmed iff the term magic(*a*) is provable (inevitably with value true).

The user should explicitly charm the ground terms of interest to him or her by writing rules such as

```
magic(goal)              |= true.
magic(rewrite(X,Y,Z)) |= true.
magic(rewrite(X,W))   |= true.
magic(word(W,I,J))    |= true.
magic(length(N)       |= true.
```

The transformation will then add rules that charm additional terms by proving additional magic(...) items (known as magic templates). Informally, a term needs to be charmed if it might contribute to the value of another charmed term. A formal description appears in Figure 7.

Finally, filter clauses are added to say that among the ground terms provable under the original program, only the charmed ones actually need to be proved. This means in practice (see section 6.5.2) that forward chaining will only prove an item if at least one grounding of that item is charmed.

The filter clauses in the speculation transformation were effectively introduced by explicitly charming all non-slashed items, running the magic templates transformation, and simplifying the result.

6.6.2 Examples of Magic Templates

Deriving Earley's algorithm. What happens if we apply this transformation to the CKY algorithm of section 6.2.1, after explicitly charming the items shown above?

Remarkably, as previously noticed by Minnen (1996), the transformed program acts just like Earley's (1970) algorithm. We can derive a weighted version of Earley's algorithm by beginning with a weighted version of CKY (the inside algorithm of section 6.2.2).[45] The transformation adds filter clauses to the constit rules, saying that the rule's head is needed only if charmed:

```
constit(X,I,K) += rewrite(X,W) * word(W,I,K)
                  needed_only_if magic(constit(X,I,K)).
constit(X,I,K) += rewrite(X,Y,Z) * constit(Y,I,J) * constit(Z,J,K)
                  needed_only_if magic(constit(X,I,K)).
```

[44] This terminology does not appear in previous literature on magic templates.

[45] At least, Earley's algorithm restricted to grammars in Chomsky Normal Form, since those are the only grammars that CKY handles before we transform it. The full Earley's algorithm in roughly our notation can be found in (Eisner et al., 2005); it allows arbitrary CFG rules to be expressed using lists, as in rewrite(np,["the",adj,n]). Section 6.3 already sketched how to handle ternary rules efficiently.

Based on the structure of the constit rules, the transformation also adds the following magic rules to the ones provided earlier by the user. Recall that ?rewrite(X,Y,Z) is considered to be true iff rewrite(X,Y,Z) is provable (footnote 12).

magic(constit(s,0,N)) |= magic(goal).
magic(constit(Y,I,$\underline{\text{J}}$)) |= ?rewrite($\underline{\text{X}}$,Y,$\underline{\text{Z}}$) & magic(constit($\underline{\text{X}}$,I,$\underline{\text{K}}$)).
magic(constit(Z,J,K)) |= ?rewrite($\underline{\text{X}}$,$\underline{\text{Y}}$,Z) & ?constit($\underline{\text{Y}}$,$\underline{\text{I}}$,J)
& magic(constit($\underline{\text{X}}$,$\underline{\text{I}}$,K)).

What do these rules mean? The second magic rule above says to charm all the possible left children constit(Y,I,J) of a charmed constituent constit(X,I,K). The third magic rule says to charm all possible right children of a charmed constituent whose left child has already been proved.

By inspecting these rules, one can see inductively that they prove only magic templates of the form magic(constit(X,I,K)) where X,I are bound and K is free.[46]

The charmed constituents are exactly those constituents that are possible given the context to their left. As in Earley's algorithm, these are the only ones that we try to build. Where Earley's algorithm would predict a vp constituent starting at position 5, we charm all potential constituents of that form by proving the non-ground item magic(constit(vp,5,K)).

Just as Earley's predictions occur top-down, the magic rules reverse the proof order of the original rule—they charm items in the body of the original rule once the head is charmed. The magic rules also work left to right within an original rule, so we only need to charm constit(vp,5,K) once we have proved a receptive context such as rewrite(s,np,vp) * constit(np,4,5). This context is analogous to having the dotted rule s → np . vp in column 5 of Earley's parse table.

In effect, the transformed program uses forward chaining to simulate the backward-chaining proof order of a strategy like that of pure Prolog.[47] The magic templates correspond to query subgoals that would appear during backward chaining. The filter clauses prevent the program from proving items that

[46] Note that it would not be appropriate to replace . . . & ?rewrite(X,Y,Z) with . . . needed_only_if ?rewrite(X,Y,Z), since that would make this condition optional, allowing the compiler to relax it and therefore charm more terms than intended. Concretely, in this example, forward chaining with our usual efficient treatment of needed_only_if (section 6.5.2) would prove overly general magic templates of the form magic(constit(X,I,K)) where not only K but also K was free.

[47] However, pure Prolog's backtracking is deterministic, whereas forward chaining is free to propagate updates in any order. It is more precise to say that the transformed program simulates a *breadth-first* or *parallel* version of Prolog which, when it has several ways to match a query subgoal, pursues them along parallel threads (whose actual operation on a serial computer may be interleaved in any way). Furthermore, since forward chaining uses a chart to avoid duplicate work, the transformed version acts like a version of Prolog with *tabling* (see footnote 5).

do not yet match any of these subgoals.

Shieber et al. (1995), specifying CKY and Earley's algorithm, remark that "proofs of soundness and completeness [for the Earley's case] are somewhat more complex ... and are directly related to the corresponding proofs for Earley's original algorithm." In our perspective, the correctness of Earley's emerges directly from the correctness of CKY and the correctness of the magic templates transformation (i.e., the fact that it preserves the semantics of charmed terms).

On-demand computation of reachable states in a finite-state machine. Another application is "on-the-fly" intersection of weighted finite-state automata, which recalls the left-to-right nature of Earley's algorithm.[48]

In automaton intersection, an arc $Q_1 \xrightarrow{X} R_1$ (in some automaton M_1) may be paired with a similarly labeled arc $Q_2 \xrightarrow{X} R_2$ (in some automaton M_2, perhaps equal to M_1). This yields an arc in the intersected machine $M_1 \cap M_2$ whose weight is the product of the original arcs' weights:

```
arc(Q1:Q2,R1:R2,X) = arc(Q1,R1,X) * arc(Q2,R2,X).
```

However, including this rule in a forward-chained program will pairs all compatible arcs in all known machines (including arcs in the new machine $M_1 \cap M_2$, leading to infinite regress). A magic templates transformation can restrict this to arcs that actually need to be derived in the service of some larger goal—just as if the definition above were backward-chained.

Consider for example the following useful program (which uses Prolog's notation for bracketed lists):

```
sum(Q,[ ]) += final(Q).      % weight of stopping at final state Q
sum(Q,[X | Xs]) += arc(Q,R,X) * sum(R,Xs).
```

Now the value of sum(q,["a","b","c"]) is the total weight of all paths from state q that accept the string abc.

We might like to find (for example) sum(q1:q2,["a","b","c"]), constructing just the necessary paths from state q1:q2 in the intersection of q1's automaton with q2's automaton. To enable such queries, apply magic templates transformation to the sum rules and the arc intersection rule, charming nothing in advance. We can then set magic(sum(q1:q2, ["a","b","c"])) to true at runtime. This results in charm spreading forward from q1:q2 along paths in the intersected machine, and spreading "top-down" from each arc along this path to the arcs that must be intersected to produce it (and which may themselves be the result of intersections). This permits the weights of all relevant arcs to be computed "bottom-up" and multiplied together along the desired paths.

[48] Composition of finite-state transducers is similar.

6.6.3 Second-order magic

Earley's algorithm does top-down prediction quite aggressively, since it predicts all possible constituents that are consistent with the left context. Many of these predictions could be ruled out with one word of lookahead—an important technique when using Earley's algorithm in practice.[49] This is known as a "left-corner" filter: we should only bother to prove magic(constit(X,I,K)) if there is some chance of proving constit(X,I,K), in the sense that there is a word(W,I,J) that could serve as the first word ("left corner") in a phrase constit(X,I,K).

Remarkably, we can get this behavior automatically by applying the magic templates transformation a *second* time. We now require the magic items themselves to be charmed before we will derive them. This activation flows bottom-up in the parse tree: we first charm magic(constit(X,I,K)) where X can rewrite directly as W, then move up to nonterminals that can rewrite starting with X, and so on.

Thus, the original CKY algorithm proved constituents bottom-up; the transformed Earley's algorithm filtered these constituents by top-down predictions; and the doubly transformed algorithm will filter the predictions by bottom-up propagation of left corners.

Before illustrating this transformation, we point out some simplifications that are possible with second-order magic. This time we will use magic2 to indicate charmed items, to avoid conflict with the magic predicate that already appears in the input program. We also assume that the user is willing to explicitly charm everything but the magic terms—since any other terms that the user regards as uninteresting are presumably already being filtered by the last transformation. Suppose that the original program contained the rule a += b * c. The input program then also usually contains

```
magic(b) |= magic(a).
magic(c) |= ?b & magic(a).
```

However, either of these rules may be omitted if the user explicitly charmed its head during the first round of magic (i.e., by stating magic(b) |= true or magic(c) |= true). As we will see, omitting these rules when possible will reduce the work that second-order magic has to do.

If we apply a second round of magic literally, the above rules (when present) respectively yield the new rules

```
magic2(magic(a))  |= magic2(magic(b)).
```

and

```
magic2(b)         |= magic2(magic(c)).
magic2(magic(a))  |= ?b & magic2(magic(c)).
```

These rules propagate charm on the magic items, from magic(b) or magic(c)

[49] Earley (1970) himself described how to use *k* words of lookahead.

up to magic(a). However, it turns out that the second and often the third of these magic2 rules can be discarded, as they are redundant with more lenient rules that prove the same heads. The second is redundant because the user has already explicitly stated that magic2(b) |= true. The third is redundant *if* the first is present, since if the program has proved b then it must have previously proved magic(b) and before that magic2(magic(b)), so that the first rule (if present) would already have been able to prove magic2(magic(a)).

The input program also contains

```
a += b * c needed_only_if magic(a).
```

We want to prove magic(a) if it will be useful in such a clause, so the second round of magic will also generate

```
magic2(magic(a)) |= ?b & ?c & magic2(a).
```

This is the rule that initiates the desired bottom-up charming of magic items. It too can be simplified. We can drop the magic2(a) condition, since the user has already explicitly stated that magic2(a) |= true. We can drop the ?c condition if the third magic2 rule above is present, and drop the entire rule if the first magic2 rule above is present. (Thus, we will end up discarding the rule unless b was charmed by the user prior to the first round of magic—making it the appropriate "bottom" where bottom-up propagation begins.)

Applying second-order magic with these simplifications to our version of Earley's algorithm, we obtain the following natural rules for introducing and propagating left corners. Note that these affect only the constit terms. Intuitively, the other terms of the original program do not need magic2 templates to entitle them to acquire first-order charm, as they were explicitly charmed by the user prior to first-order magic.

```
magic2(magic(constit(X,I,K))) |= rewrite(X,W) & word(W,I,K).
magic2(magic(constit(X,I,K))) | = ?rewrite(X,Y,Z).
                                  & magic2(magic(constit(Y,I,J))).
```

The transformation then applies the left-corner filter to the magic templates defined by first-order magic:

```
magic(constit(s,0,N)) | = magic(goal)
                       needed_only_if magic2(magic(constit(s,0,N))).
magic(constit(Y,I,J))  | = ?rewrite(X,Y,Z)
                           & magic(constit(X,I,K))
                       needed_only_if magic2(magic(constit(Y,I,J))).
magic(constit(Z,J,K)) | = ?rewrite(X,Y,Z) & ?constit(Y,I,J)
                           & magic(constit(X,I,K)).
                       needed_only_if magic2(magic(constit(Z,J,K))).
```

Note that the magic2(constit(X,I,K)) items proved above are specific to the span I–K in the current sentence: they have X,I,K all bound. However, one could remove this dependence on I,K by using the speculation transformation (section 6.5). Then the first time a particular word W is observed via some fact

Given a unary predicate magic that may already appear in $\mathcal{P}$. We say that a term t is **already charmed**[50] if $\mathcal{P}$ contains a rule magic(s) |= true where s is at least as general as t.
For each rule R_i in $\mathcal{P}$, of the form $r_i \oplus_i = E_i$, given an ordering $e_{i1}, \ldots, e_{ik_i}$ of the items whose values are combined by E_i (including any filter clauses).[51]

```
foreach rule $R_i$
    unless $r_i$ is already charmed
        append "needed_only_if magic($r_i$)" to $R_i$
    for $j = 1, 2, \ldots k_i$
        unless $e_{ij}$ is already charmed
            add "magic($e_{ij}$) |= ?$e_{i1}$ & $\cdots$ & ?$e_{i(j-1)}$ & magic($r_i$)" to $\mathcal{P}$
            optionally relax this new rule by generalizing its head[52]
```

[50] This test is used only to simplify the output.

[51] In the examples in the text, this is taken to be the order of mention, which is a reasonable default.

[52] That is, replace the head with a more general pattern. For example, one may replace some variables or other sub-terms in the head with variables that do not appear in the rule body. See section 6.6.4 for discussion.

FIGURE 7 The magic templates transformation.

word(W,I,K), the program will derive magic2(constit(X,I0,K0))/word(W,I0,K0) for each nonterminal X of which W is a possible left corner. These left corner table entries leave I0,K0 free, so once they are computed, they can be combined not only with word(W,I,K) but also with later appearances of the same word, such as word(W,I2,K2).

6.6.4 Formalizing Magic Templates

Our version of magic templates is shown in Figure 7. Readers who are familiar with Ramakrishnan (1991) should note that our presentation focuses on the case that Ramakrishnan calls "full sips," where each term used in a rule's body constrains the bindings of variables in subsequent terms.

However, to allow other "sips" (sideways information-passing strategies), we can optionally rename variables in the heads of magic($\ldots$) |= $\cdots$ rules so that they become free. This results in proving fewer, more general magic($\ldots$) items.[53]

Ramakrishnan's construction instead attempts to *drop* these variables—as well as other variables that provably remain free. However, his construction

[53] This may even avoid an asymptotic slowdown. Why? It is possible to prove more magic templates than items in the original program, because magic(a) (proved top-down) may acquire bindings for variables that are still free in a (proved bottom-up). It is wise to drop such variables from magic(a) or leave them free.

is less flexible because it only drops variables that appear as direct arguments to the top-level predicate. It also leads to a proliferation of new and non-interacting predicates (such as magic_constit^{bbf}/2), which correspond to different binding patterns in the top-level predicate.

Dropping variables rather than freeing them does have the advantage that it makes terms smaller, perhaps resulting in constant-time reductions of speed and space. However, we opt to defer this kind of "abbreviation" of terms to an optional subsequent transformation—the same transformation (not given in this paper) that we would use to abbreviate the slash(...) items introduced by speculation in section 6.5.1. Pushing abbreviation into a separate transformation keeps our Figure 7 simple. The abbreviation transformation could also attempt more ambitious simplifications than Ramakrishnan (1991), who does not simplify away nested free variables, duplicated bound variables, or constants, nor even detect all free variables that are arguments to the top-level predicate.

6.7 Conclusions

This paper has introduced the formalism of weighted logic programming, a powerful declarative language for describing a wide range of useful algorithms.

We outlined several fundamental techniques for rearranging a weighted logic program to make it more efficient. Each of the techniques is connected to ideas in both logic programming and in parsing, and has multiple uses in natural language processing. We used them to recovered several known parsing optimizations, such as

- unary rule cycle elimination
- Earley's (1970) algorithm with an added left corner filter
- Eisner and Satta's (1999) $O(n^3)$ bilexical parsing
- on-the-fly intersection of weighted automata

as well as various other small rearrangements of algorithms, such as a slight improvement to lexicalized CKY parsing.

We showed how weighted logic programming can be made more expressive and its transformations simplified by allowing non-ground items to be derived, and we introduced a new kind of side condition that does not bind variables—the needed_only_if construction—to streamline the use of non-ground items.

Our specific techniques included weighted generalizations of folding and unfolding; the speculation transformation (an original generalization of folding); and an improved formulation of the magic templates transformation. This work does not exhaust the set of useful transformations. For example, Eisner et al. (2005) briefly discuss transformations that derive programs to compute gradients of, or bounds on, the values computed by the original dynamic program. We intend in the future to give formal treatments of these. We also plan to investigate other potentially useful transformations, in particular, transformations that exploit program invariants to perform tasks such as "abbreviating" complex items.

We hope that the paradigm presented here proves useful to those who wish to further study the problems to which weighted logic programming can be applied, as well as to those who wish to apply it to those problems themselves.

In the long run, we hope that by detailing a set of possible program transformation steps, we can work toward creating a system that would search automatically for practically effective transformations of a given weighted logic program, by incorporating observations about the program's structure as well as data collected from experimental runs. Such an implemented system could be of great practical value.

References

Aji, S. and R. McEliece. 2000. The generalized distributive law. *IEEE Transactions on Information Theory* 46(2):325–343.

Earley, J. 1970. An efficient context-free parsing algorithm. *Comm. ACM* 13(2):94–102.

Eisner, J., E. Goldlust, and N. A. Smith. 2005. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proc of HLT/EMNLP*.

Eisner, J. and G. Satta. 1999. Efficient parsing for bilexical context-free grammars and head-automaton grammars. In *Proc. of ACL*, pages 457–464.

Fitting, M. 2002. Fixpoint semantics for logic programming a survey. *TCS* 278(1-2):25–51.

Goodman, J. 1999. Semiring parsing. *Computational Linguistics* 25(4):573–605.

Huang, L., H. Zhang, and D. Gildea. 2005. Machine translation as lexicalized parsing with hooks. In *Proc. of IWPT*, pages 65–73.

McAllester, D. 1999. On the complexity analysis of static analyses. In *Proc of 6th Internat. Static Analysis Symposium*.

Minnen, G. 1996. Magic for filter optimization in dynamic bottom-up processing. In *Proc 34th ACL*, pages 247–254.

Ramakrishnan, R. 1991. Magic templates: a spellbinding approach to logic programs. *J. Log. Prog.* 11(3-4):189–216.

Ross, K. A. and Y. Sagiv. 1992. Monotonic aggregation in deductive databases. In *PODS '92*, pages 114–126.

Sagonas, Konstantinos, Terrance Swift, and David S. Warren. 1994. XSB as an efficient deductive database engine. *ACM SIGMOD Record* 23(2):442–453.

Shieber, S. M., Y. Schabes, and F. Pereira. 1995. Principles and implementation of deductive parsing. *J. Logic Prog*. 24(1–2):3–36.

Sikkel, Klaus. 1997. *Parsing Schemata: A Framework for Specification and Analysis of Parsing Algorithms*. Texts in Theoretical Computer Science. Springer-Verlag.

Stolcke, A. 1995. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational Linguistics* 21(2):165–201.

Tamaki, H. and T. Sato. 1984. Unfold/fold transformation of logic programs. In *Proc 2nd ICLP*, pages 127–138.

Van Gelder, A. 1992. The well-founded semantics of aggregation. In *PODS '92*, pages 127–138. New York, NY, USA: ACM Press. ISBN 0-89791-519-4.

Younger, D. H. 1967. Recognition and parsing of context-free languages in time n^3. *Info. and Control* 10(2):189–208.

Zhou, N.-F.. and T. Sato. 2003. Toward a high-performance system for symbolic and statistical modeling. In *Proc of IJCAI Workshop on Learning Stat. Models from Relational Data*.