4

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.

Keywords weighted logic programming, dynamic programming, program transformation, parsing algorithms

4.1 Introduction

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

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. Finally, our formalization of these transformations has been simplified by our use of "gap

FG-2006.
Edited by Paola Monachesi, Gerald Penn,
Giorgio Satta and Shuly Wintner.

passing" ideas from categorial grammar and non-ground terms from logic programming.

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 NLP researchers regularly propose new such algorithms. Dynamic programming is used to parse many different grammar formalisms. It is also used in stack decoding, grammar induction, finite-state methods, and syntax-based approaches to machine translation and language modeling.

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; in other cases, it will slow it down. The actual effect can of course be determined empirically by running the transformed program (or in some cases, predicted more quickly by profiling the *untransformed* program as it runs on typical inputs). Thus, at least in principle, one could apply automatic local search methods.

4.2 Our Formalism

4.2.1 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 grammar, starting at nonterminal X, can generate the input substring from position I to position K.

```
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,"Dumbo").
rewrite(np,"flies").
rewrite(vp,"flies").

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

We say that this logic program is a **dynamic program** because it satisfies a simple restriction: all **variables** (capitalized) in a rule's left-hand side (rule **head**) also appear on its right-hand side (rule **body**). Logic programs restricted in this way correspond to the "grammatical deduction systems" discussed by Shieber et al. (1995). They can be evaluated by a simple agenda-based, bottom-up dynamic programming algorithm.[1]

This paper, however, deals with general logic programs without this restriction. For example, one may wish to assert the availability of an "epsilon" word at *every* position K in the sentence: word(epsilon,K,K). We emphasize this because it is convenient for some of our transformations to introduce new non-dynamic rules. One can often eliminate non-dynamic 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.

4.2.2 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).

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,"Dumbo") = 0.6.  % p(np → "Dumbo" | np)
rewrite(np,"flies") = 0.4.      % 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 4.2.1, except that now the body of each rule is an arbitrary *expression*, and the :- operator is replaced by an "aggregation operator" such as += or max=. Since line 2 can be instantiated for example as constit(s,0,2) += rewrite(s,np,vp) * constit(np,0,1) * constit(vp,1,2), the value of rewrite(s,np,vp) * constit(np,0,1) * constit(vp,1,2) (if any) is used as a summand (i.e., an operand of +=) in the value of constit(s,0,2).

[1] This is superior to a Prolog-style backtracking algorithm. It runs in polynomial time, rather than wasting exponential time re-deriving the same constituents in different contexts, or failing to terminate if the grammar is left-recursive.

We will formalize this in section 4.2.3 below.

The result—for this program—is that the computed value of constit(s,0,2) will be the inside probability $\beta_s(0, 2)$ for a particular input sentence and grammar.[2] In practice one might wait until runtime to provide the description of the sentence (the rules for word and length) and perhaps even of the grammar (the rewrite axioms). In this case our transformations would typically be used only on the part of the program specified at compile time. But for simplicity, we suppose in this paper that the whole program is specified at compile time.

If the left-hand sides of two rules unify, then the rules must use the same aggregation operator, to guarantee that each item is aggregated in a consistent way. Each constit(...) item above is aggregated with +=.

4.2.3 Semantics of Weighted Logic Programs

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

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

For each provable ground item r, let $\mathcal{P}(r)$ be the non-empty multiset of all ground expressions E on provable items such that $r \oplus_r= E$ instantiates 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

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

(putting $[\![r]\!] = \infty$ 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 an error otherwise.

In the terminology of the logic programming community, this definition is equivalent to saying that the valuation function $[\![\cdot]\!]$ is a fixed point of the monotone consequence operator.[3]

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

[3] Such a fixed point need not be unique, and there is a rich line of research into attempting

Example. In the example of section 4.2.2, this means that for any particular X, I, K for which constit(X,I,K) is provable, $[\![\text{constit}(X, I, K)]\!]$ equals

$$\begin{aligned} & \textstyle\sum_{J,Y,Z} [\![\text{rewrite}(X,Y,Z)]\!] * [\![\text{constit}(Y,I,J)]\!] * [\![\text{constit}(Z,J,K)]\!] \\ + \quad & \textstyle\sum_{W} [\![\text{rewrite}(X,W)]\!] * [\![\text{word}(W,I,J)]\!] \end{aligned}$$

where, for example, the first 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 constit(X,I,J), which is being defined.

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 variables that appear only in the rule's head; these indicate that the rule is not a dynamic programming rule.

Discussion. 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 4.2.1 by writing expressions over boolean values:[4]

```
constit(X,I,K) |= rewrite(X,Y,Z) & constit(Y,I,J) & constit(Z,J,K).
```

In general, this framework subsumes the practically useful case of Goodman (1999), which requires all values to fall in a single semiring and all rules to use only the semiring operations.[5]

Definition. A program transformation $T : \mathcal{P} \rightarrow \mathcal{P}'$ is defined to be **semantics-preserving** if for every item r which is provable by $\mathcal{P}$, r is also provable by $\mathcal{P}'$ and

$$[\![r]\!]_{\mathcal{P}} = [\![r]\!]_{\mathcal{P}'}$$

4.2.4 Computing Semantics by Forward-Chaining

A basic strategy for computing the semantics is "forward chaining." The idea is to maintain current values for all proved items, and to propagate updates to these values, from the right-hand side of a rule to its left-hand side, until all the equations are satisfied. (This might not halt: even an unweighted dynamic program can encode an arbitrary Turing machine.)

---

to more precisely characterize the intuitive semantics of logic programs with negation or aggregation. The interested reader should refer to Fitting (2002), or to, for example, Van Gelder (1992) or Ross and Sagiv (1992) for a discussion of the semantics of aggregate logic programs. In practice, one may obtain some single fixpoint by running the forward-chaining algorithm of the section 4.2.4 below and hoping that it converges.

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

[5] Dropping these requirements allows our framework to handle neural networks, game trees, and other interesting systems of equations. Note that Goodman's "side conditions" can be easily handled in our framework (see Eisner et al., 2005).

As already noted in section 4.2.1, 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 the semiring-weighted case. 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. The algorithm must propagate arbitrary updates, derive values for non-ground items, and obtain the value of foo(3,3), if not explicitly derived, from (e.g.) the derived value of foo(X,X) or foo(X,3) in preference to the less specific foo(X,Y). Furthermore, certain aggregation operators, but not all, permit optimizations that are important for efficiency. We defer these algorithmic details to a separate paper.

4.3 Folding

Weighted dynamic programs are schemata that define systems of simultaneous equations. Such systems can often be rearranged without affecting their solutions. In the same way, weighted dynamic programs can be transformed to obtain new programs with better runtime.

For a first example, consider our previous rule from section 4.2.2,

```
constit(X,I,K) += rewrite(X,Y,Z) * constit(Y,I,J) * constit(Z,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 instantiated in only $O(N^3 \cdot n^3)$ different ways. For this—and the other parsing programs we consider here—it turns out the runtime of forward chaining can be kept down to $O(1)$ time per instantiation.[6] 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,Y,Z,I,J) * constit(Z,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

[6] 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 runtimes of appropriately-indexed forward chaining from the number of instantiations. However, that meta-theorem applies only to unweighted dynamic programs. Similar results in the weighted case require acyclicity. Then one can use the two-phase method of Goodman (1999), which begins with a run of McAllester's method 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 *.[7] 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 an algorithm inspired by the junction-tree algorithm for probabilistic inference in graphical models which they call the "generalized distributive law," which is equivalent to repeated application of the folding transformation, and which they demonstrate 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* in this program are non-ground terms such as constit(s,0,K)/constit(n,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 equality of the two K arguments (by internal unification) indicates that the missing Z is always at the *right* of the X, while their freeness means

[7] Since all semirings enforce a similar distributive property, the trick can be applied equally well to Viterbi parsing and unweighted recognition (section 4.2.3).

that the 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—the source of folding's efficiency. Our earlier program with temp2 could have been obtained by a further automatic transformation that replaced all constit(X,I,K)/constit(Z,J,K) having free K with the more compactly stored temp2(X,Z,I,J).

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 4.5 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 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 4.4 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:

```
temp(X,Y2,Y3,I,J) += rewrite(X,Y1,Y2,Y3) * constit(Y1,I,J).
temp2(X,Y3,I,K)   += temp(X,Y2,Y3,I,J)   * constit(Y2,J,K).
constit(X,I,L)    += temp2(X,Y3,I,K)     * constit(Y3,K,L).
```

This restores $O(n^3)$ runtime (more precisely, $O(N^4 \cdot n^2 + N^3 \cdot n^3 + N^2 \cdot n^3)$)[8] 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, temp2(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.

The variable elimination algorithm for undirected graphical models can be viewed as repeated folding. An undirected graphical model expresses a joint

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

probability distribution over P,Q by marginalizing (summing) over a product of clique potentials:

```
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 $p_{k+1}, p_{k+2}, \ldots, p_n$. We may *eliminate* variable X by transforming to

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

The first line no longer mentions X because the second line sums over it. The variable elimination algorithm applies this procedure repeatedly to the last line to eliminate the remaining variables.[9]

Common subexpression elimination. Folding can also be used multiple times to eliminate common subexpressions. Consider the following code for *bilexical* CKY parsing:

```
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 lexicalized at head word H. The program effectively has two types of rewrite rule, 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

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

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

Using the same temp 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.

[9] Determining the optimal elimination order is NP-complete.

Definition of folding. Our definition allows an additional use of the distributive law. The original program may define the value of item r by aggregating values not only over free variables in the body of one rule, but also across n rules. Thus, when defining the temp item s, we also allow it to aggregate across n rules. In ordinary mathematical notation, we are performing a generalized version of the following substitution:

Before		**After**
$r = \sum_i (E_i * F)$	$\Rightarrow$	$r = s * F$
$s = \sum_i E_i$	$\Rightarrow$	$s = \sum_i E_i$

given the distributive property $\sum_i (E_i * F) = (\sum_i E_i) * F$. 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. We will generalize by allowing this common context to be an arbitrary function F.

We require that the rules defining the temp item, $s = \sum_i E_i$, be in the program *already* before folding occurs. If necessary, their presence may be arranged by a trivial **definition introduction** transformation that adds $r/F = \sum_i E_i$. (Explicitly using the slashed item r/F for s will ensure that the variable occurrence requirement below is met.) We claim without proof that all transformations in this paper are semantics-preserving in the sense of section 4.2.3.

Below and throughout the paper, we use the notation $F[X]$ to denote the *literal* substitution of expression X for all instances of μ in an expression F over items, even if X contains variables that appear in F or elsewhere in the rule containing $F[X]$. We assume that μ is a distinguished item name, of the same value type as X, and does not appear elsewhere.

Algorithm 4.3.1 (Folding transformation)

Given n distinct rules $R_1, \ldots, R_n$ in $\mathcal{P}$, where each R_i has the form $r \oplus\!= F[E_i]$. Given also a term s that unifies with the heads of exactly n rules in the program, all of which are distinct from the R_i, and which respectively take the form $s \odot\!= E_i$ after this unification.
Then the folding transformation deletes the rules $R_1, \ldots, R_n$, replacing them with a new rule $r \oplus\!= F[s]$, provided that

- *Any variable that occurs in any of the E_i which also occurs in either F or r must also occur in s.*[10]
- *Either $\oplus\!=$ or $\odot\!=$ is simply $=$,*[11] *or else the distributive property $[\![F[x \odot y]]\!] = [\![F[x]]\!] \oplus [\![F[y]]\!]$ holds for all assignments of terms to variables and all valuation functions $[\![\cdot]\!]$.*[12]

As a tricky example, one can replace r += p(I,J) * log(q(J,K)) with r += p(I,J) * log(s(J)) in the presence of s(J) *= q(J,K). Here E_1 is q(J,K), and F[x] is p(I,J) * log(x).

4.4 Unfolding

In general, a folding transformation leaves the asymptotic runtime alone, or may improve it when combined with the distributive law.[13] Thus, the inverse of the folding transformation, called **unfolding**, makes the asymptotic time complexity the same or worse. However, unfolding may be advantageous as a precursor to some other transformation that improves runtime. It also saves space. Sometimes we can improve both time and space complexity by unfolding and then transforming the program further.

For example, recall the bilexical CKY parser given near the end of section 4.3. 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:

```
temp3(X:H,Z:H2,I,J) += rewrite(X:H,Y:H,Z:H2) * constit(Y:H,I,J).
constit(X:H,I,K)    += temp3(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 unfolding the temp3 item—that is, substituting its definition in place each time it is used, which uses unification and relies on distributivity—and then trimming away its now-unneeded definition, 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 $O\left(N^3 \cdot n^5\right)$, but by eliminating storage of the temp items, it improves the space complexity to $O\left(N \cdot n^3\right)$. The payoff is that now we can refold this rule differently—either as in section 4.3, or alternatively as follows (Eisner and Satta, 1999, which misses the chance to eliminate common subexpressions):

[10] This ensures that s does not sum over any variables that must remain visible in the revised r rule.

[11] For instance, in the very first example of section 4.3, the temp item was defined using = and therefore performed no summation. No distributivity was needed.

[12] That is, all valuation functions over the space of items, including dummy items x and y, when extended over expressions in the usual way.

[13] It may either help or hurt the *actual* runtime, and it certainly increases the space needed to store items' values.

```
temp4(X:H,Y:H,J,K) += rewrite(X:H,Y:H,Z:H2) * constit(Z:H2,J,K).
constit(X:H,I,K)   += temp4(X:H,Y:H,J,K)    * constit(Y:H,I,J).
```

Either way, the time complexity is now $O\left(N^3 \cdot n^4 + N^2 \cdot n^4\right)$—better than the original programmer's version—while the space complexity has increased only back to the original programmer's $O\left(N^2 \cdot n^4\right)$.

Unfolding resembles inlining of a subroutine call. Section 4.5 will show how it can thus be used for program specialization—improving efficiency by a constant factor and also enabling further transformations that improve asymptotic efficiency.

4.5 Speculation

We now generalize folding to handle recursive rules. 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 examples.

Split head-automaton grammars. We consider a restricted kind of bilexical CFG in which a head word combines with all of its right children before any of its left children (Eisner and Satta, 1999). The "inside algorithm" below[14] builds up rconstit items by starting with a word and successively adding 0 or more child constituents to the right, then builds up constit items by adding 0 or more child constituents to the left of this.

```
rconstit(X:H,I,K) += 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).
```

This algorithm has runtime $O(N^3 \cdot n^5)$ (dominated by line 4). We now exploit the conditional independence of left children from right children. Instead of building up a constit from a particular, existing rconstit (line 3) and then adding left children (line 4), we transform the program so it builds up the constit item *speculatively*, waiting until the end to fill in each of the various rconstit items that could have spawned it. Replace lines 3–4 with

```
lconstit(X0:H0,X0,J0,J0) += 1   needed_only_if  rconstit(X0:H0,J0,K0).
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).
constit(X:H0,I,K0)       += lconstit(X:H0,X0,I,J0) * rconstit(X0:H0,J0,K0).
```

[14] For simplicity, this code ignores the cost of starting, "flipping," or stopping in different nonterminal states.

The new temp item lconstit(X:H0,X0,I,J0) represents the *left* half of a constituent. We can regard it in the categorial terms of section 4.3: as the last line illustrates, it is just a more compact notation for a constit missing its rconstit right half—namely 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, equivalent to constit(X0:H0,J0,K0)/rconstit(X0:H0,J0,K0). 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 finally fills in the missing right half rconstit.

The special filter clauses needed_only_if rconstit(X0:H0,J0,K0) are added solely for efficiency. They say that it is not necessary to build "useless" left halves purely speculatively, but only when there is some right half for them to combine with. Their semantics are sketched below.

In this case, the filter clause on the second rule manages to ensure that in any lconstit(X:H, X0,I,J0) that we need to build, J0 will be the start position of the head word H. (Such a constraint is already true for rconstits; an empty lconstit inherits it from the rconstit filter, and passes it along to successively wider lconstit.) Since the temp item records only this redundant position and not K (the right boundary of the unknown rconstit), runtime falls from $O\left(n^5\right)$ to $O\left(n^4\right)$.

As a bonus, we can now obtain the $O\left(n^3\right)$ algorithm of Eisner and Satta (1999). Simply unfold the instances of constit in the rconstit and temp rules (i.e., replacing them with lconstit * rconstit per our new definition). Then refold those rules differently.[15]

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 4.2.3), they relax the restrictions on the $[\![\cdot]\!]$ function, allowing more possible semantics (all of which, however, 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 an instantiated rule $r \oplus_r= E$ if it has a filter clause needed_only_if C such that no consistent instantiation of C has been proved. (The "consistent" instantiations are those where variables of C that are shared with r or E are instantiated accordingly. Other variables, such as K0 in the example above, may have any instantiation.)

How does this help operationally, in the forward chaining algorithm?

[15] In each case, use a rewrite to combine a rconstit with an lconstit to its right (first folding the rewrite with whichever one does not contribute the head word).

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 placed on the forward-chaining agenda. The update must still be carried out if the filter clause is proved later.[16]

In the example above, forward chaining on the first lconstit rule produces an "zero-width" lconstit(X0:H0,X0,J0,J0) in which all variables are free.[17] This lconstit can be used anywhere; in particular, it can combine with any rconstit, so the filter clause says it is needed as soon as *any* rconstit has been proved. The real filtering power comes when the second rule tries to build further from the zero-width lconstit using the second rule. Then X0, H0, and J0 indirectly become bound to values in the rewrite and constit items of that rule (because of the internal unification in the zero-width lconstit(X0:H0,X0,J0,J0)). Thus, the filter clause is now better instantiated, e.g., needed_only_if rconstit(vp:"flies",1,K0). Only if such an rconstit has been derived (for some K0) are we required to consider updating the clause head, e.g., lconstit(s:"flies",vp,0,1).

Unary rule closure. Before formalizing speculation, we informally show another instructive application: precomputing unary rule closure in a CFG. We start with a version of the inside algorithm that allows nonterminal unary rules:

program fragment $\mathcal{P}_0$*:*

```
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 rules include, among others,

program fragment $\mathcal{P}_0$*: (continued)*

```
rewrite(np1,np3)  = 0.1.
rewrite(np3,np2)  = 0.2.
rewrite(np2,np3)  = 0.3.
rewrite(np3,det,n)= 0.4. . . .
```

We can *unfold* the grammar into the program to get rules such as

program fragment $\mathcal{P}_1$*:*

```
constit(np1,I,K) += 0.1 * constit(np3, I, K).
constit(np3,I,K) += 0.2 * constit(np2, I, K).
constit(np2,I,K) += 0.3 * constit(np3, I, K).
```

[16] Sometimes a filter is true, causing the update, but later becomes false. For instance, rconstit(vp:"flies",1,K0) may no longer be provable after sentence-specific word(. . .) axioms are retracted. Because the update is now optional, the algorithm is not required to retract the update (at least not on that basis), although it is free to do so in order to reclaim memory. This optionality is useful in some of our examples below: entries will be filled into the unary-rule-closure and left-corner tables only as needed, but need not be retracted after each sentence and then rederived.

[17] As well as adding 1 to any other items that specialize and override this one.

```
constit(np3,I,K) += 0.4 * constit(det, I, J) * constit(n, J, K). . . .
```

This amounts to program specialization. If we have unfolded (at least) the unary rewrite rules into the program, we can now apply speculation to eliminate them "offline":

program fragment $\mathcal{P}_2$*:*

```
temp(X0,X0)     += 1   needed_only_if  constit(X0,I0,K0).
temp(np1,X0)    += 0.1 * temp(np3,X0).
temp(np3,X0)    += 0.2 * temp(np2,X0).
temp(np2,X0)    += 0.3 * temp(np3,X0).
constit(X,I0,K0) += temp(X,X0) * other(constit(X0,I0,K0)).
other(constit(np3,I,K)) += 0.4 * constit(det,I,J) * constit(n,J,K). . . .
```

For any nonterminals X and Y, our temporary item temp(X,X0) is just compact notation for constit(X,I0,K0)/constit(X0,I0,K0): the inside probability of deriving a constit(X,I0,K0) by a sequence of 0 or more unary rules from a constit(X0,I0,K0) that covers the same span I0–K0. In other words, it is the total probability of all (possibly empty) unary-rewrite chains $X \rightarrow^* X0$.

The final two rules recover unslashed constit items. other(constit(X,I,K)) is any constit(X0,I,K) whose derivation does *not* begin with a unary rule. The next-to-last rule builds this into constit(X,I,J) through a sequence of 0 or more unary rules.

Crucially, the temp(X,X0) items have values that are *independent of* I and K. So they need not be computed separately for every span in every sentence. For each nonterminal X0, all temp(X,X0) values will be computed once and for all (the very first time a constit(X0,I,K) constituent is built) by iterating the first three rules below to convergence. These values will then remain static while the grammar does, even if the sentence changes (see footnote 16).

Definition of speculation. In general, the value of a slashed item is a *function*, just like the semantics of a slashed constituent in categorial grammar. Also as in categorial grammar, gaps are introduced with the identity function, passed with function composition, and eliminated with function application. Fortunately, in commutative semiring-weighted programs like the ones above, all functions have the form "multiply by x" for some weight x. We can represent such a function simply as x, using semiring 1 for the identity function, semiring multiplication for both composition and application, and semiring addition for pointwise addition.

Algorithm 4.5.1 (Speculation transformation)

Let a be an item to slash out, where any variables in a do not occur elsewhere in $\mathcal{P}$. Let slash *and* other *be functors that do not already appear in $\mathcal{P}$. Let $R_1, \ldots, R_n$ be distinct rules in $\mathcal{P}$, where each R_i is $r_i \oplus_i= F_i[t_i]$, and*
- *For $i \leq k$, t_i does not unify with a.*
- *For $i > k$, t_i unifies with a; more strongly, it matches a non-empty subset of the ground terms that a does.*[18]
- *Certain conditions on distributivity (satisfied by semiring programs).*

Then the speculation transformation constructs the following new program, in which the values of slash *items are functions, $\oplus_i$ is extended to sum functions pointwise, $\circ$ denotes function composition, and $F[x]$ denotes function application.*
- *slash(a,a)* $\oplus_{a=}$ *($\lambda x.\ x$) needed_only_if a.*
- *($\forall 1 \leq i \leq n$) slash(r_i, a)* $\oplus_i=$ *$F_i \circ$ slash(t_i, a) needed_only_if a.*
- *($\forall 1 \leq i \leq k$) other(r_i)* $\oplus_i=$ *F_i[other(t_i)].*
- *($\forall$ rules p* $\oplus=$ *q not among the R_i) other(p)* $\oplus=$ *q.*[19]
- *X* $\oplus_X=$ *other(X) unless X is an instance of a.*
- *X* $\oplus_X=$ *(slash(X,a))[other(a)].*[20]

Intuitively, other(X) accumulates ways of building X other than instantiations of $F_{i_1}[F_{i_2}[\cdots F_{i_j}[a]]]$ for $j > 0$. slash(X,a) aggregates all instantiations of the function $\lambda x.F_{i_1}[F_{i_2}[\cdots F_{i_j}[x]]]$ for $j \geq 0$. This pointwise sum of functions is only applied to other(...) items, to prevent double-counting (analogous to spurious ambiguity in a categorial grammar).

To apply this formal transformation in the unary-rule elimination example, take a=constit(X0,I0,K0), and the R_i to be the "unary" constit rules, where each t_i is the last item in the body of R_i. Here $k = 0$. The resulting slashed items have the form slash(constit(X,I,K), constit(X0,I0,K0)), but the rules would only derive instances where I=I0 and K=K0. All such rules are filtered by needed_only_if constit(X0,I0,K0).[21]

To apply the transformation in the split head-automaton example, take a=constit(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 $k = 1$.[22]

[18] By adding side conditions, any rule can be split into an $i \leq k$ rule, an $i > k$ rule, and a rule not among the R_i.

[19] Typically, many of the other(...) items can be unfolded and then their defining rules removed. This is why few or none remained in the informal examples above.

[20] In the final two rules, X ranges over the entire universe of terms. Recall that $\oplus_X$ is the aggregation operator for X. One could construct separate rules for items aggregated with different operators.

[21] In this example, the efficiency filters are redundant on rules after the first. Runtime analysis or (perhaps) static analysis would show that they have no actual filtering effect, allowing us to eliminate them.

[22] In this program, *all* constits are built from rconstits using R_1 and R_2, so other(constit(...)) has no derivations. Concretely, the single rule that the transformation generates to define

4.6 Converting bottom-up to top-down

4.6.1 Magic Templates

Finally, we give an important transformation that explains and generalizes the way that speculation introduced needed_only_if filters.

The bottom-up "forward-chaining" execution strategy mentioned in section 4.2.4 will compute the values for all provable items. Many of these items may, however, be irrelevant in the sense that they do not contribute directly or indirectly to the value of goal. (In parsing, they are legal constituents that do not lead to a complete parse.) We can avoid generation of these irrelevant items by employing the magic templates transformation (Ramakrishnan, 1991), which prevents an item from being built unless it will help lead to a "desired" item.

We need the value of a theorem foo if it occurs in in the body of a rule where (1) we need the value of the rule's head and (2) we have already derived the items preceding foo in the rule's body.[23] For example, in the CKY parsing rule

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

we need constit(Y,I,J) (for a particular Y,I,J) if we need constit(X,I,K) (for some X,K) and we already know that rewrite(X,Y,Z) is provable (for some Z), which we denote ?rewrite(X,Y,Z).[24] Hence

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

For example, the above rule may derive magic(constit(vp,1,J)) as true. That means it is worthwhile to look for vp objects *starting* at position 1. The *ending* position J is unspecified—a free variable. Ramakrishnan (1991)'s original presentation drops such superfluous variables to obtain a dynamic programming version:

```
magic_constit(Y,I)) |= magic(constit(X,I,K)) & ?rewrite(X,Y,Z).
```

Ramakrishnan's move is not necessary for the present section, but it improves efficiency, and will simplify section 4.6.2.

Here are all the magic rules for CKY parsing (section 4.2.2):

```
magic_goal          |= true.
magic_constit(s,0)  |= magic_goal.
```

---

other(constit(...)) depends on other(constit(...)) itself, so it can never be derived from the axioms (and may be trimmed away as useless).

[23] This left-to-right order within a rule is traditional, but any order would do.

[24] It would not be appropriate to write needed_only_if rewrite(X,Y,Z). The rewrite item is part of the definition of whether the magic item should be true or false—not simply a condition on whether a more lenient definition of magic (which would serve as a less effective filter) is worth deriving. As a concrete consequence, using needed_only_if rewrite(X,Y,Z) below would derive a magic item in which Y remained a free variable, a lenient definition that would license the main program to derive many useless constit items.

```
magic_constit(Y,I)    |= magic_constit(X,I,K)) & ?rewrite(X,Y,Z).
magic_constit(Z,J)    |= magic_constit(X,I)
                          & ?rewrite(X,Y,Z) & ?constit(Y,I,J).
```

Then, we modify the rules of the original program, adding magic_foo as a filter on the derivation of foo:

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

This transformed program uses forward chaining to simulate backward chaining (though perhaps a breadth-first version of backward chaining). Since we ultimately want the value of goal (or derivations of goal), we set magic_goal=true. That causes us to derive magic_constit facts at the start of the sentence, which license the building of actual constit items with values, which let us derive magic_constit facts later in the sentence, and so on. Remarkably, as previously noticed by Minnen (1996), the operation of this transformed program is the same as Earley's algorithm (Earley, 1970): constituents are predicted top-down, and built bottom-up only if they have a "customer" to the immediate left.

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 semantics-preserving nature of the magic templates transformation.

Another application is "on-the-fly" intersection of weighted finite-state automata, which recalls the left-to-right nature of Earley's algorithm. Intersection of arcs $Q \xrightarrow{X} R$ in machines M_1 and M_2, bearing the same symbol X, is accomplished by multiplying their weights:

```
arc(M1:M2,Q1:Q2,R1:R2,X) += arc(M1,Q1,R1,W) * arc(M2,Q2,R2,X).
```

But this pairs all compatible arcs in all known machines (including the new machine M1:M2, leading to infinite regress). A magic templates transformation can restrict to arcs that actually need to be derived in the service of some larger goal (e.g., summing over selected paths from a specified paired start state Q1:Q2).

4.6.2 Second-order magic

Using magic templates to change to a top-down computation order will still allow some irrelevant items to be derived. Not all items we "need" to derive a value for goal, according to a top-down search from goal, will actually turn out

to be provable bottom-up. This may lead to too much top-down exploration: Earley's algorithm may predict many categories such as vp at position 1 (i.e., derive magic(constit(vp,1,J))) when there is not even a possible verb at position 1.

We can therefore apply the magic templates transformation a second time, to the rules that defined the first-order magic items. This yields second-order magic items of the form magic_magic_foo, meaning "we need to realize that we need to build foo":

```
magic_magic_goal          |= magic_magic_constit(s,0).
magic_magic_constit(X,I)  |= magic_magic_constit(Y,I) & ?rewrite(X,Y,Z).
magic_magic_constit(X,I)  |= magic_magic_constit(Z,J)
                               & ?rewrite(X,Y,Z) & ?constit(Y,I,J).
```

They can be added as needed_only_if filter clauses that limit Earley's "predict" rules (i.e., the rules that derive the first-order magic items). As before, K remains free. Consider in particular the second rule above, which says that if Earley's can wisely predict Y at position I, it can also wisely predict X and (by recursion) any other nonterminal of which Y is a left corner. (Using speculation to abstract away from the sentence position I, we could build up a left corner table offline.)

The base case of this left-corner computation comes from enchanting one of the rules that *uses* rather than *defines* a first-order magic item,[25]

```
constit(X,I,K) += rewrite(X,W) * word(W,I,K)
                  needed_only_if magic_constit(X,I).
```

to obtain

```
magic_magic_constit(X,I) |= ?rewrite(X,W) & ?word(W,I,K).
```

Thus, the second-order predicates will constrain top-down prediction at position I to predict only nonterminals that are left-corner compatible with the word W at I. In short, we have derived the left-corner filter on Earley's algorithm, by repeating the same transformation that derived Earley's algorithm in the first place!

4.7 Conclusions

We introduced a weighted logic programming formalism for describing a wide range of useful algorithms. After sketching its denotational and operational semantics, we outlined a number of fundamental techniques—program transformations—for rearranging a weighted logic program to make it more efficient.

[25] We do not show the enchantments of the other such rules, as they do not add any further power.

In addition to exploiting several known logic programming transformations, we described a weighted extension of folding and unfolding, and presented the speculation transformation, a substantial generalization of folding.

We showed that each technique was connected to ideas in both logic programming and in parsing, and had multiple uses in NLP algorithms. We recovered several known parsing optimizations by applying reusable transformations: for example, Earley's algorithm, the left-corner filter, parser specialization, offline unary rule cycle elimination, and the bilexical parsing techniques from (Eisner and Satta, 1999).

We noted throughout how program transformations could be simplified by allowing the resulting programs to derive non-ground items. One important tool was our proposed needed_only_if filter.

The paradigm and techniques presented here may be directly useful to algorithm designers as well as to those who are interested in formalisms for specifying and manipulating algorithms.

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.

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

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*.