

Transformation Schemes for Optimization of Parsing Algorithms

Jochen Setz

Seminar “Formal Grammars” WS 2006/2007
Programming Systems Lab
University of Saarbruecken

Abstract. Parsing Grammars can be done by deducting called “Parsing as Deduction”. The parser can be formalised by inference systems or logic programs. Efficient parsing algorithms like Early or CKY use some tricks on the inference systems to parse the grammars more efficient than a naive parsing algorithms.

Some of this tricks can be transferred to Transformation Schemes which possibly can optimise arbitrary inference systems. Three of these Transformation Schemes will be described here an their asymptotic time will be analysed.

1 Introduction

Parsing complex grammars like context-free grammars more powerful ones can easily become a very expensive thing, in the sense of running-time, because naive parsing algorithms tends to derive a lot of things, which will never play a role for the goal, by deriving without any strategy. Such parsers are mostly complete unusable, even for small problems. For example parsing very small English sentences could take a very long time with naive parsers.

To make this problems manageable, the natural language processing community developed several algorithms, which uses some tricks to reduce the running-time. For example the Early-Algorithm (Nederhof and Satta 2004) runs in time $O(N^2 \cdot n^3)$ where N is the size of the grammar, and n the size of the input sentence.

In general, paring algorithms are dynamic programming algorithms which can be formulated by inference systems. Here, we will describe, how some of these tricks can be abstracted to every parsing algorithm which can be described by a inference system. They will even work for parsing of Semiring-Weighted Grammars. Theses tricks are called *transformation schemes*. It will be shown, how these *transformation schemes* can reduce the asymptotic running-time, or at the least reduce the running-time by a constant factor. Of course these transformations will be completely formalised.

In the following, the Cocke-Younger-Kasami Algorithm (CYK) (Shieber et al. 1995) will be used as an example, how existing fast-running algorithms still use such transformations.

With his help, the *folding* transformation will be explained, a powerful transformation that for example allows a CKY-like bottom-up parser to have an asymptotic complexity that is independent of the length of the production rules.

While the usual CKY-Algorithm reaches this aim by assuming the grammar to be in Chomsky-Normal Form, the same optimisation can be obtained at the level of the inference system using the *Folding* transformation.

After that, we will go on with the *Unfolding* transformation, the inverse of the *Folding* transformation.

The third and last transformation will be *Speculation*, which is a quite new thing which is used to handle special cases like unary rule cycles, and is used for newer parsing algorithms for example to parse Splited Bilexicalized Grammars (Eisner and Satta 1999). Although *Speculation* is not really relevant for the older famous algorithms, the CKY-Algorithm will again serve as introduction example. Additionally we will have a view on bilexical-parsing, where *Speculation* fulfils his whole possibilities.

In the last section, the semantics of these transformations will be analysed.

This paper contains nothing new, it is completely based on “Program Transformations for Optimisation of Parsing Algorithms and Other Weighted Logic Programs” by Jason Eisner and John Blatz 2007.

1.1 Inference Systems - A very short introduction

Inference Systems consists of rules of the form:

conclusion :- ARG1, ARG2, ARG3, side-constraints.

Such a rule state a derivation step. It means, we can derive the conclusion, when we can derive all the argument, and all the side-constraints are fulfilled.

An Inference System describes a set of items, which can be derived. For example an item `constit(X,I,K)` means, an X can be derived from position I to K. A typical side-constraint would be a production rule $X \rightarrow YZ$ which will be notated in the following as `rewrite(X,Y,Z)`.

Where the rule above shows the boolean case, where the arguments and side-constraints are connected by a \wedge , which means for every argument/side-constraint it must be checked whether it is true. All the items which can be derived by that rule in a derivation step, are connected by a \vee .

The Inference Systems can be extended to Weighted Semiring Inference Systems. A semiring contains a set \mathcal{N} of possible values and two operators \oplus and \otimes , where \otimes distributes over \oplus .

Often, the rules are notated as:

$$\frac{ARG1 \quad ARG2 \quad ARG3}{CONCLUSION} \textit{side - constraints.}$$

In a Weighted Logic Program, each provable item has a value out of \mathcal{N} which can be notated with an '='- Sign at the item. Often, Weighted Semiring Inference Systems are used to compute the probabilities, that items are derived. A rule in looks as follows:

`conclusion += ARG1 * ARG2 * ARG3 * side-constraints`

Where the probability to derive the conclusion would be the product of the arguments and the side-constraints. The probability of all conclusions which can be derived by that rule in one step would be the sum of all these conclusions.

Parsing as deduction works by applying inference rules on the set of already derived items, as long as an goal is reached, which yields a successful parse, or there is no rule which can be applied, which means, the input cannot be parsed and is therefore not in the grammar.

Applying a rule means, to iterate over the free variables, variables which don't occur in the conclusion, and prove if there is an item in the set of already derived items with this values.

The running-time of an inference rule depends on the number of free variables, in fact, the running-time is the sum of all possible combinations of the free variables. The running-time of an inference system depends on the running-time of the rules, asymptotically on the most expensive rule, and how often this rule could be applied.

1.2 The Cocke-Younger-Kasami (CYK) Algorithm

Let us have a quick look at the "traditional" CYK-Algorithm. Figure 1 shows the complete algorithm, together with a simple grammar which also contains the lexicon, and a input sentence.

Line one can derive unary rules with lexical entries on the left side, where line two derives the other, binary rules. An item like `constit(X,I,K)` means, we can derive a nonterminal `X` from position `I` to `K`.

Line three describes the whole parse-tree, we will obtain when we can derive the start symbol `s` from position `0` to `n`. The next four lines, denote a little example grammar, which can generate simple English sentences. The last lines represent the input sentence, or the sentence we want to derive.

Most of the following examples will base on this example.

1.3 Run-time of the CKY-Algorithm

The asymptotic run-time of our CKY-Algorithm is affected by the second rule:

```
constit(X,I,K) :- rewrite(X,Y,Z), constit(Y,I,J), constit(Z,J,K).
```

which contains the most variables, over whom, the parser have to iterate.

This rule contains 3 Variables out of the grammars alphabet `N` (Terminals and Non-Terminals): `X,Y,Z` and 3 Variables which indicate positions `n`: `I, K, J`. With this, it follows that the the CKY-Algorithm is in $O(N^3 \cdot n^3)$.

```

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).
rewrite(np,det,n).
rewrite(np, "Dumbo").
rewrite(vp,"flies").

word("Dumbo",0,1).
word("flies",1,2).
length(2).

```

Fig. 1. The CKY-parser, a grammar and an input sentence.

2 Folding

Now let's start with the first transformation which is called *Folding*. It is a very simple, but also very effective transformation, which may help to reduce the asymptotic running-time.

The basic idea is, to avoid unnecessary applying of rules and iterating over free variables. To sketch the system remember how the evaluation of a inference system works. We have to iterate over all values of all free variables for each rule, and for every instance of a rule. Now consider a rule with three arguments. Maybe there is a variable, which occurs to be free in the right side of the rule, and this variable doesn't occur in all arguments. Then it's possible to take just the arguments which use this variable to calculate the possible values. With that values the other variables can be calculated, without iterating over the whole domain of our previous calculated variable.

2.1 Example 1(CKY)

Let us consider the CKY-Algorithm, more precisely the second rule, which is the most expensive rule:

```
constit(X,I,K) :- rewrite(X,Y,Z), constit(Y,I,J), constit(Z,J,K).
```

All free occurrences of variables are underlined. We see, Y is free in this rule, and Y just occurs in the first two items of the rules body.

To get a feeling how *Folding* works, we start with a first *folding* step, which is obviously correct, but with no effect for the run-time.

We introduce a new rule, and adjust the old rule as follows:

```
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).
```

Because of the associativity of the *and* operation, it is no problem, to compute the two first items first. We hold the result of that calculation with help of the new introduced “temp”-item. After that, it is folded into the computation of `constit`.

Distributivity Because \vee distributes over \wedge we can modify our previous *Folding*. In our second rule, `Y` appears only in the `temp` item. Nevertheless the rule sums over `J`, `Y` and `Z`.

Because of the distributivity of \vee and \wedge its possible to sum just over `Y` before multiplying by `constit(Z,J,K)`. With this we receive a better *Folding* transformation:

```
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).
```

Here we can see the whole idea of folding. Because of the distributivity we can compute `Y` first, and then we can forget it as soon as possible, and don't have to iterate unnecessarily.

2.2 Run-time Analysis of the first Example

Let us analyse the second folding. The first rule iterates over the 3 Variables `X`, `Y`, `Z` out of the grammars alphabet `N`, and over two positions `N`: `I` and `J` which leads to a run-time of $O(N^3 \cdot n^2)$ of the first rule.

The second rule only iterates over `X` and `Z` out of `N` and over the three positions `I`, `J` and `K`, which yields an run-time of $O(N^2 \cdot n^3)$.

So the whole overall run-time is $O(N^3 \cdot n^2 + N^2 \cdot n^3)$, compared to $O(N^3 \cdot n^3)$ from our original rule we have seen in Chapter 1.3. Our next example will demonstrate a better result of the folding.

2.3 Example 2 (CKY without CNF)

In this example we will see how the CKY-Algorithm uses folding to reduce the running time.

Consider that CKY requires the rewrite-rules to be in Chomsky-Normal Form (CNF). We will see that this restriction is highly related to folding. For this, let's have a look on this rule:

```
constit(X,I,L) :- ((rewrite(X,Y1,Y2,Y3), constit (Y1,I,J)),
                  constit (Y2,J,K)),
                  constit(Y3,K,L)
```

This rule can derive projections with three items on the right side. Note that the brackets works like the “dot” in dotted-rules.

With the folding transformation we can transfer the above rule like this:

```

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 folding is nothing else than a transfer of a projection with three items on the right side to CNF, but with good advantages in the running-time, as we will see in the run-time analysis.

2.4 Run-time Analysis of the second Example

The original rule in this example runs in $O(N^4 \cdot n^4)$ cause it iterates over four Grammar Variables N ($X, Y1, Y2, Y3$) and four positions n (I, J, K, L).

The folded version runs quite faster. Rule one iterates over $X, Y2, Y3, Y1$ out of N and over two positions I, J which means this rule runs in $O(N^4 \cdot n^2)$. The second rule iterates over tree N 's ($X, Y2, Y3$) and tree positions n (I, J, K) which means the asymptotic time is in $O(N^3 \cdot n^3)$. The last rule just iterates over two grammar variables N ($X, Y3$) and over three positions n (I, K, L) which leads to $O(N^2 \cdot n^3)$.

The all together asymptotic run-time is $O(N^4 \cdot n^2 + N^3 \cdot n^3 + N^2 \cdot n^3)$ which is really faster than $O(N^4 \cdot n^4)$ regarding to n .

2.5 Formalisation of Folding

The first example demonstrated the use and the idea of folding. The second example showed, that the original CKY-Algorithm ever used a special case of folding.

Now we have to take a look at the formal abstraction of folding to every inference system including semiring-weighted inference systems.

Figure 2 shows the formal definition of folding. Note that $F[E]$ denote the *literal* substitution where items are the literals, and F and E are expressions over items. $F[E]$ substitutes E for all instances of μ in F , where μ is a distinguished symbol that does not appear elsewhere.

The new introduced rule R corresponds to our **temp**-rules. The iteration over i searches for all rules, where the new rule R can be folded in. A little thing we haven't seen in the examples yet, is, variable renaming. Suppose we have two rules with the same items, where just the variables are different. Then both of this rules can use the same new **temp**-rule to fold out these items.

The first bullet provided by the replacement, means, a variable which occurs in our items we folded out in the original rule and also occurs in the body or head of the **temp**-rule must also occur in the instance of the **temp**-rule we filled in the modification of the original rule. This prevents a wrong iteration over these variable.

The second bullet describes, that the distributivity still holds for all variables and all valuations, which means no restriction happens here.

Given a new rule R in the form $r \oplus = F[s]$ (which will be used to replace a group of rules R_1, \dots, R_n in \mathcal{P}). Let S_1, \dots, 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) ¹ 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, \dots, 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 ².
- Either $\odot =$ is simply $=$, or else the distributive property $\llbracket F[\mu] \oplus F[\nu] \rrbracket = \llbracket F[\mu \odot \nu] \rrbracket$ holds for all assignments of terms to variables and all valuation functions $\llbracket \cdot \rrbracket$ ³.

Fig. 2. The weighted folding transformation taken from Eisner and Blatz 2007

3 Unfolding

The next transformation we want to have a look on is called *Unfolding*. As the name indicates, unfolding is the inverse transformation of *Folding*, we have seen in the section before.

Here, the basic idea is to replace a rule by a set of new rules. Every rule is more specific than the original one. To Unfold one of the premises of a rule, we add a new rule for every rule, whose conclusion unifies with this premise to the inference system. Such a new rule contains all other premises of the original rule, but the variables are replaced by the variables, which comes from the unified rule.

Again, the distributivity of the operations (\vee, \wedge) makes this transformation possible. Whereas we factored out in the *Folding* transformation, we expand in the *Unfolding* transformation.

With *Unfolding*, new, more specific parsers emerge, which can improve the running-time by a constant factor, which can be very useful for specific grammars.

Another very important field of application is, to restore folded grammars. Maybe you get a grammar, where someone already did some folding applications, and you think you could do a better folding. Here, unfolding helps, to improve the asymptotic behaviour, because a good folding can improve the asymptotic running-time.

¹ 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

² This ensures that computing s_i by rule S_i does not sum over this variable, which would break the co-variation of E_i with F or r as required by the original rule R_i .

³ 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.

3.1 Example 3 (CKY)

Let us again look at the CKY-Algorithm. More precisely this three rules:

```
constit(X,I,K) :- rewrite(X,Y,Z), constit(Y,I,J), constit(Z,J,K).
                rewrite(s,np,vp).
                rewrite(np,det,n).
```

Here, we want to unfold the `rewrite(X,Y,Z)` premise. The following rules are those, whose conclusion unifies with this premise. After the transformation, we get this inference system:

```
constit(s,I,K) :- constit(np,I,J), constit(vp,J,K).
constit(np,I,K) :- constit(det,I,J), constit(n,J,K).
                rewrite(s,np,vp).
                rewrite(np,det,n).
```

The first rule is replaced by two, more specific rules. Please note, in this example, you could throw away the last two rules, because there is no way in the inference system, to reach them. But in general, there could be another rule, which still contains a `rewrite(X,Y,Z)` premise, and so this rules have to stay in these system.

3.2 Run-time Analysis of the third example

As there are still 3 Variables out of the Grammar: `s`, `np` and `vp` in the first rule, respectively `np`, `det` and `n` in the second one, and 3 indices in both rules: `I`, `J` and `K` the asymptotic run-time is still $O(N^3 \cdot n^3)$.

But nevertheless this transformation can be very useful in practise. It avoids to try every `rewrite`-rule and therefore reduces the running-time by a constant factor.

3.3 Formalisation of Unfolding

Now we want to abstract the previous example to every inference system including semiring-weighted inference systems.

Figure 3 shows the formal definition of folding. The rule R from the definition correspond to the first rule of the third example, before the transformation. The `rewrite` item is contained in the s , and S_1, \dots, S_n corresponds to the two `rewrite` rules of the example. A new rule emerges by unifying s with the conclusion of S_i , which must be done for every $1 \leq i \leq n$. After that, the rule R can be replaced by the rules $r_i \oplus = F_i[E_i]$, where E_i denotes the unification of s with the conclusion of S_i .

The two bullets provided by the replacement mean exactly the same as described earlier (Section 2.5). The first one prevents a wrong iteration, and the second one assures the distributivity.

⁴ Before forming this tuple, rename the variables in S_i so that they do not conflict with those in r, F, s .

Let R be a rule in \mathcal{P} , given in the form $r \oplus = F[s]$. Let S_1, \dots, 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 $(r, F, S, S_i)^4$ 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 $\llbracket F[\mu] \oplus F[\nu] \rrbracket = \llbracket F[\mu \odot \nu] \rrbracket$.

Fig. 3. The weighted unfolding transformation taken from Eisner and Blatz 2007

4 Speculation

The last transformation we will discuss here, is called *Speculation*, which main application area is to handle cycles in weighted semiring inference systems. Obviously, parsing a cycle by deduction will never terminate. In practise, parsing a cycle in weighted semiring inference systems will abort, for example, if the value becomes stable enough. That means, it aborts, when the difference between the previous value, and the value computed by the next turn, added to the previous one is smaller than a predefined ϵ . This computation is somewhat time-consuming (Eisner and Blatz 2007)

Note that a circle is on top of an $\mathbf{constit}(X, I, K)$, that means, if we have derived an X we have to compute this cycle. This cycle doesn't depend on I and K , and will therefore always have the same value. So we can prevent computing this circle again and again by doing it just once, and multiply the precalculated value if necessary.

That is the idea of *Speculation*. It speculates this value, by assuming this X is already derived.

In the boolean case, *Speculation* is less interesting, cause there is no need to compute cycles in more than one turn, as its value will always be 1 if it can be derived. But assume we have unary rules, with which it is possible to build extremely long chains. So that computing such a chain will be very time-consuming. Then, with *Speculation* it is again possible, to do this expensive computing just once.

4.1 Example 4 (CKY)

Let's start with a first example, again the CKY-Algorithm. This time, we will not restrict this algorithm to be in Chomsky Normal Form, and allow unary rules of the form $\mathbf{rewrite}(X, Y)$, which leads to these rules:

```

    constit(X,I,K) :- rewrite(X,W),word(W,J,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).

```

Depending on the grammar, such long chains can occur, and this parser would derive them as often, as they occur with the second rule. Note that the `rewrite(X,Y)` in that rule, doesn't depend of the I and K of the following `constit` item. (Illustrated in Figure 5)

After the *Speculation* transformation, the rules look as follows:

```

temp(X0,X0).
temp(X,X0) :- rewrite(X,Y), temp(Y,X0).
other(constit(X,I,K)) :- rewrite(X,W), word(W,J,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 two `temp` items derive the chain. The first rule is the termination condition, which just says, that it's always possible to derive an `X0` when `X0` is already derived. The second rule derives the rest of the chain, starting at `X0`.

The last rule calls the computation of this chain, by pretending, the `X0` is already derived.

The rules three and four are nearly the same as in our previous examples, excepting the 'other' in the conclusion, which is necessary to prevent computing the chain, although it is already computed. Without this 'other' the chain would be computed all over again because of the recursive rules.

Figure 4 shows the weighted semiring version of that algorithm, which works completely analogous to the boolean case, excepts that here cycles must be computed, which really occurs in real applications, whereas the boolean case is improbable to.

```

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

```

Fig. 4. Weighted Semiring Version of Example 4

4.2 Example 5 (Split Bilexical CFG)

A very important area of application of *Speculation* is parsing *Split Bilexical CFGs*, in which a head word must combine with all of its right children before any of its left children (Eisner and Satta 1999). Here, the left children are completely independent of its right children. So with *Speculation* it is possible to speculate the left children and therefore compute them just once.

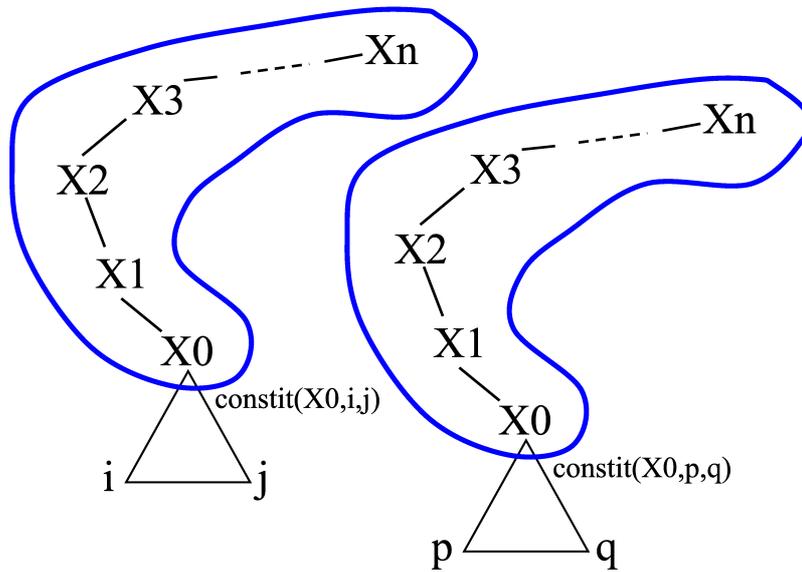


Fig. 5. Principle of speculation - Boolean Case

The naive algorithm for bilexical context-free parsing is $O(n^5)$ (Eisner and Blatz 2007). But it can be break down over $O(n^4)$ to $O(n^3)$ (Eisner and Satta 1999). The techniques used by Eisner and Satta to break down the running-time can be abstracted which yields the *Speculation*-Transformation (Eisner and Blatz 2007).

This shows, that *Speculation* can break down the asymptotic enormously.

4.3 Filter Clauses

Another trick, which can improve the asymptotic running-time are *Filter Clauses* which prevents the parser to speculate things, which aren't required. This looks as follows:

```
temp(X0,X0) :- needed_only_if constit(X0,I0,K0).
temp(X,X0) :- rewrite(X,Y), temp(Y,X0) needed_only_if constit(X0,I0,K0).
```

This just means, that before computing a cycle, it will be checked, if there is at least one X_0 derived so far.

4.4 Run-time of Speculation

As mentioned in Chapter 4.2 *Speculation* can improve the asymptotic behaviour. Suppose, that in the worst-case, a cycle for on the top of a $\text{constit}(X,I,J)$ must

be computed n^2 times (all possible values for the indices I and J), whereas with *speculation* this computation will be done just for one time.

Suppose computing this cycle takes n^x , the whole time of the algorithm without *Speculation* would be $O(n^x \cdot n^2) = O(n^{x+2})$. With *Speculation* it is in $O(n^x \cdot 1) = O(n^x)$.

4.5 Formalisation of Speculation

Now we have to formalise the *Speculation*. Figure 6 shows the definition. This definitions works on semirings of weights W . All the rules must use the same aggregation operator $\oplus=$, with identity element $\bar{0}$, and each rule's body must be a product of items, using \otimes as associative binary operator that distributes over \oplus .

In Example 4 we have \vee as the aggregation operator with identity **false**, and \wedge as the \otimes operator with identity element **true**.

The **slash** item in our example would be the **temp** items. **slash**(r_i, x) means, deriving products we want to speculate. Where x is an item which is slashed out of that rule. In our example **constit**($X0, I, K$), which doesn't unify with the part of the rule, we want to speculate, and must be more general with the other part t . The definition claims, that the t must be the last item of the rule. To avoid this restriction, \otimes have to be commutative. The **slash** item can only multiplied to an **other** item, to prevent multiple computing of the speculated part, as we discussed in Example 4. Intuitively, **other**(r_i) accumulates ways of building r_i other than just grounding r_i .

According to our Example, the R_1 from the definition would be the rule which contains the unary production.

5 Semantics

Before we have to look at the specific transformations, we have to define the phrase 'semantics preserving'. This means, two inference systems are *semantics preserving* if they can derive the same items, and none of the inference systems can derive more than the other.

Now we can claim, that all our transformations are *semantics preserving*. Except for the new introduced **temp** items, which obviously cannot be derived by the original inference system, because they don't occur there.

Nevertheless they can still be said to be *semantics preserving* because in such two-step transformations $\mathcal{P} \rightarrow \mathcal{P}'' \rightarrow \mathcal{P}'$ introduce new items in the first step $\mathcal{P} \rightarrow \mathcal{P}''$, and eliminate them in the second $\mathcal{P}'' \rightarrow \mathcal{P}'$.

⁵ If necessary, the program can be preprocessed so that such an index exists. Any rule can be split into three more specialised rules: an $i \leq k$ rule, an $i \geq k$ rule, and a rule not among the R_i . Some of these rules may require boolean side conditions to restrict their applicability.

⁶ That is, t_i is "more specific" than x : it matches a non-empty subset of the ground terms that x does.

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, \dots, 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 index⁵ such that $0 \leq k \leq n$ and

- For $i \leq k$, t_i does not unify with x .
- For $i \geq k$, t_i unifies with x ; moreover, their unification equals t_i ⁶.

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} .

- **slash**(x, x) $\oplus_x = \bar{1}$ **needed_only_if** x .
- $(\forall 1 \leq i \leq n)$ **slash**(r_i, x) $\oplus = F_i \otimes$ **slash**(t_i, x) **needed_only_if** x .
- $(\forall 1 \leq i \leq k)$ **other**(r_i) $\oplus = F_i \otimes$ **other**(t_i).
- $(\forall$ rules $p \oplus_p = E$ not among the R_i) **other**(p) $\oplus_p = E$.
- **matches_x**(x) \models **true**.
- **matches_x**(A) \models **false**.
- $A \oplus_A =$ **other**(A) **if not** **matches_x**(A).
- $A \oplus_A =$ **slash**(A, x) \otimes **other**(x).

Fig. 6. The semiring-weighted speculation transformation taken from Eisner and Blatz 2007

This composite transformation is *semantics preserving* although the step $P'' \rightarrow P'$ is not.

6 Conclusions

We have seen tree techniques called transformations which helps to optimise inference systems:

- Folding
- Unfolding
- Speculation

While *Folding* and *Speculation* can improve the asymptotic behaviour, *Unfolding* cannot directly improve the asymptotic behaviour, but can reduce the running-time by a constant factor, and can help to prepare a good folding and thus can also be used to improve the asymptotic behaviour indirectly.

Folding follows from tricks, famous algorithms used. For example the CKY-Algorithm or the Early-Algorithm which make context-free parsing manageable. Now we have a corresponding abstraction to every inference system, to optimise it.

Speculation is something new, which allows to handle unbounded sequences of rules, included cycles. It is a really important transformation for Parsers of new grammars like Bilexicalized grammars.(Eisner and Blatz 2007)

References

- Eisner, J., Blatz, J.: Program Transformations for Optimisation of Parsing Algorithms and Other Weighted Logic Programs. In Proceedings of FG 2006: The 11th Conference on Formal Grammar, pp. 45-85. CSLI Publications.
- Nederhof, M-N. and Satta, G.: Introduction to Parsing Algorithms for NLP, Lecture Notes, ESSLI 2004.
- Shieber, S. M., Schabes, Y., Pereira, F. C. N.: Principles and Implementation of Deductive Parsing, *Journal of Logic Programming* 24:1+2, 3-36, 1995.
- Eisner, Jason and Giorgio Satta (1999). Efficient parsing for bilexical context-free grammars and head automaton grammars. Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics, pages 457-464, College Park, Maryland, June.