

Chapter 1

ON TWO CLASSES OF FEATURE PATHS IN LARGE-SCALE UNIFICATION GRAMMARS

Liviu Ciortuz*

Computer Science Department

University of York, UK

ciortuz@cs.york.ac.uk

Abstract We investigate two related techniques, Quick Check and Generalised Reduction, that contribute significantly to speeding up parsing with large-scale typed-unification grammars. These techniques exploit the properties of two particular classes of feature paths, which are obtained empirically by parsing a training corpus. Quick check is concerned with paths that most often lead to unification failure. Generalised reduction identifies a larger class, made of paths which contribute to (rule selection through) unification failure, and let us take advantage of those which do not (or seldom) do it. We experiment with the two techniques, using a compilation-based parsing system on a large-scale grammar of English. The combined improvement in parsing speed we obtained is 56%.

1. Introduction

Interest in unification grammars in Natural Language Processing can be traced back to the seminal work on the PATR-II system (Shieber et al., 1983). Since then, different types of unification-based grammar formalisms have been developed by the Computational Linguistics community: Lexical Functional Grammars (LFG) (Kaplan and Bresnan, 1983), Head-driven Phrase Structure Grammars (HPSG) (Pollard and Sag, 1994), and Categorical Unification Grammars (Uszkoreit, 1986). In

*This paper was published in the volume “New Developments in Parsing Technologies”, Hurry Bunt, Giorgio Satta, John Carroll (eds.), Kluwer Academic Publishers, 2004.

the past few years, a number of wide-coverage unification grammars have been implemented: the HPSG for English developed at Stanford called LinGO (Flickinger et al., 2000), two HPSGs for Japanese developed at Tokyo University (Mitsuishi et al., 1998), and at DFKI–Saarbrücken, Germany (Siegel, 2000), and an HPSG for German, developed also at DFKI (Müller, 1999). Large-scale LFG grammars have been developed by Xerox Corporation, but they are not publicly available.

Feature structure (FS) unification is by far the most time-consuming task during parsing with large-scale typed-unification grammars. Therefore making it more efficient is of great interest. Several solutions for speeding-up FS unification have been proposed until now: FS sharing (Pereira, 1985), (Tomabechi, 1992), FS unfilling (Gerdemann, 1995), FS restriction (Kiefer et al., 1999) and the quick check pre-unification filter (Malouf et al., 2000). Compilation is another approach to speed up FS unification (Aït-Kaci and Di Cosmo, 1993), (Wintner and Francez, 1999), (Ciortuz, 2001a).

The *quick check* (QC) pre-unification filter, one of the most effective speed-up techniques for FS unification, is very simple in itself. It considers the set of feature paths most probably leading to unification failure, and then compares the corresponding values of these paths inside two FSs to be unified. If such a pair of values is — eventually if their root sorts are — incompatible, then it must be the case that the two FSs do not unify. The quick check filter reduces impressively the FS unification time on all large-scale grammars on which it has been tested (Callmeier, 2000), (Oepen and Callmeier, 2000). On the LinGO grammar, the wide-coverage HPSG for English, the QC filter speeds up parsing by around 63% for the PET system (Callmeier, 2000), and 42% for the LIGHT compiler (Ciortuz, 2002b).¹

The effectiveness of the QC pre-unification technique resides in the fact that the proportion of failure paths inside rule FSs is relatively small. Using the LIGHT system, we identified 148 failure paths on the CSLI test-suite (Oepen and Carroll, 2000), out of the total of 494 paths inside rule FSs for LinGO. Among these paths, only a small number is responsible for most of the unification failures: for the LinGO grammar, in the LIGHT system, we used maximum 43 QC-paths. The compilation of quick check is not only recommended but also compulsory in a setup where the rule argument FS creation is avoided through a Specialised Rule Compilation scheme like the one developed in the LIGHT system (Ciortuz, 2001a). Recently, a version of quick check on dynamically determined QC-paths was proposed in (Ninomyia et al., 2002), to be used in the area of information retrieval and database query answering.

In contrast to the quick check, the *generalised reduction* (GR) technique works on paths inside rule FSs, eliminating those that do not contribute (or only seldom contribute) to unification failure. We refer to the set of paths retained by the GR technique as *GR-paths*. We present an efficient algorithm for generalised reduction of rule FSs in large-scale typed-unification grammars. Measurements done on the LinGO grammar revealed that almost 60% of the feature constraints in the expanded rule FSs can be eliminated without affecting the parsing result on the CSLI test-suite. As a consequence, the LIGHT system registered a reduction of the unification time that sped parsing up to 22%. It turns out that most QC-paths are also GR-paths — the few exceptions are extensions of maximal GR-paths — so the quick check and the generalised reduction techniques are orthogonal: they can be applied at the same time leading to significant improvements in parsing speed.

The fact that — despite their declarative/logic un-differentiated status — feature constraints inside rule FSs can be preferentially treated in order to speed-up parsing, was the subject of some early work on unification-based grammars. In (Shieber, 1985) certain features were selected for enabling top-down (predictive) operations, in the context of bottom-up parsing. (Nagata, 1992) studied empirically the effect of interleaving parsing with two versions of FS unification: early unification and late unification. (Maxwell and Kaplan, 1993) reports experiments in which a small Lexical Functional Grammar was tuned by changing the status of certain constraints (from “functional” into “phrasal”) and compared the effect these changes had on parsing performance.

Generalised reduction performs feature constraint selection in an automatic way, based on a learning procedure using as parameter the parsing result. Unlike (Nagata, 1992) and (Maxwell and Kaplan, 1993), our strategy for feature constraint selection does not propose new rules. It simply proposes for each rule a new FS version to be used in early unification. The active bottom-up chart-based parsing strategy in LIGHT, when using (early unification on) the GR-restricted form of rules, is extended with a consistency-check phase, which is performed by either type-checking or late unification. In this way, a parsing algorithm which, using early unification, may slightly over-generate, may in the end perform better than one which does not differentiate between classes of feature constraints/paths. This fact, which is true for LinGO, is similar to the effect noted by (Maxwell and Kaplan, 1993) following experiments with a small LFG grammar: with that grammar, a parser which does not prune may behave better than one that prunes.

Although the quick check and generalised reduction were developed independently, they complement each other in the sense that both are

concerned with classes of feature paths that warrant specialised processing in large-scale typed-unification grammars. We first concentrate on the two techniques separately: Section 2 describes research into the compilation of the quick check, and Section 3 generalised reduction of rule feature structures. We use as our experimental setup the LIGHT compiler and the LinGO grammar. We bring the two strands of work together at the end of Section 3 and in the Conclusions.

2. Compiling the Quick Check Pre-unification Filter

This section describes how to obtain a compiled form of the QC filter, such that it can be elegantly and efficiently combined with compilers for unification-based grammars. In Section 2.1 we outline the problems we encountered when trying to accommodate the simple (interpreted-like) QC filter into the LIGHT compiler setup, in particular its co-existence with the other main optimisation technique we proposed — the specialised compiled form of rules (Ciortuz, 2001a). Section 2.2 describes the compiled (“pre-computed”) form of the QC-vectors, while Section 2.3 details the computation of their run-time, completed form. Section 2.4 reports measurements for running the LIGHT system with the LinGO grammar on the CSLI test-suite, both with and without QC filtering, and makes suggestions for further improvements.

Making it simple, the *main idea* behind the pre-unification QC test is the following: Having got the knowledge about the most probable failure paths $\pi_1, \pi_2, \dots, \pi_n$ in the application of parsing rules, before doing the unification of a certain feature structure ψ_1 (representing a phrase) with another feature structure ψ_2 (representing a syntactic rule argument), one can check whether for every path π_i , its values in ψ_1 and respectively ψ_2 are compatible, i.e., $root(\psi_1.\pi_i) \wedge root(\psi_2.\pi_i) \neq \perp$. (The function *root* designates the root sort of its argument FS, $\psi.\pi$ is the usual notation for the sub-structure identified inside ψ by the feature path π , and \perp is the bottom/inconsistent sort in the grammar’s sort hierarchy.) The sort hierarchy is assumed an inferior semi-lattice, with $s \wedge t$ designating the unique greatest lower bound (glb) of the sorts s and t . If the sort compatibility test is not passed for a path π , then it follows immediately that the feature structures ψ_1 and ψ_2 do not unify. This simple technique eliminates much of the actually unnecessary work performed during unification in case of failure.

Now, surprisingly enough, the introduction of the QC technique in the *compilation approach* for the HPSG-like type unification grammars as tried out for the LIGHT system was not immediately effective. Even

at the first sight, one can see that the QC filter might not be so much effective in the compilation approach, since compiled unification is significantly faster than interpreted unification. Of course, performing the QC has a cost. Measurements made on the LKB, TDL (Krieger and Schäfer, 1994), and PET systems revealed that in case of interpreted-like parsing with LinGO-like grammar, it is worthwhile to pay for the QC test, since most/many unifications fail even after the rules' combinatorial filter was applied. But in the case of compiled parsing, the trade-off to be made between the time required by the QC test (expressed as a function of the number of failure paths to be checked) on one hand, and the speed of the unification procedure on the other hand is dramatically narrowed. A compiled form of the QC filter as will be presented here is proven able to “enlarge” again this trade-off area to speed up parsing.

Basically, we will show how QC-vectors $\{root(\psi.\pi_1), root(\psi.\pi_2), \dots, root(\psi.\pi_n)\}$ can be computed in two stages. The first one is done once for all at compilation time — we call it QC pre-computation —, and it is completed at run-time by the second-stage computation, according to specific circumstances. The basis for this “two-stage” computation of QC-vectors resides in the facts that *i.* the order in which one rule's arguments will be processed is known at grammar preprocessing/compilation time, and *ii.* for any (in general not known in advance) feature structure ψ which will be involved in the QC test, we know that ψ will be an instance of (i.e., subsumed by) a certain feature structure Ψ fully known at compilation time ($\psi \sqsubseteq \Psi$). Formally, for any QC feature-path π , the $QC_\pi(\psi) = root(\psi.\pi)$ value will be computed by applying at run-time a function ρ to a certain argument $preComp(\Psi, \pi)$, computed at compilation time:

$$QC_\pi(\psi) = \rho(preComp(\Psi, \pi)).$$

2.1 Can the (interpreted) QC test be accommodated into (compiled) parsing in LIGHT?

Let us first analyse the way the QC was conceived in the interpreting setup of the LKB and TDL/PAGE systems for parsing with HPSG-like grammars:

- as soon as a phrase is parsed, a ‘passive’ QC-*vector* is computed for its associated feature structure ψ . This QC-vector is defined as $\{root(\psi.\pi_1), root(\psi.\pi_2), \dots, root(\psi.\pi_n)\}$. If one of the paths π_i is not defined for ψ , then the *i*-th component in the computed QC-vector is taken by definition \top , the top element in the grammar's sort hierarchy;

- every m -ary rule is associated with m ‘active’ QC-vectors; in the case of a binary rule φ , we have first a ‘key’ QC-vector $\{root(\varphi'.\pi_1), root(\varphi'.\pi_2), \dots, root(\varphi'.\pi_n)\}$, where $\varphi' = \varphi.KEY-ARG$, namely the sub-structure corresponding to the head/key argument in the FS representing the rule;
- before trying to apply the rule φ to a candidate key argument ψ , i.e., before unifying ψ with φ' , the QC pre-unification test does $root(\psi.\pi_i) \wedge root(\varphi'.\pi_i)$ for $i = \overline{1, n}$, that means the conjunction of the corresponding components of the two QC-vectors. If the conjunction result is always consistent (i.e., not \perp), the system unifies ψ with φ' , and if this unification succeeds, then the system produces a new, ‘active’ QC-vector, corresponding to the next argument to be parsed. If the current rule is a binary one, this new active QC-vector is what we call the ‘complete’ QC-vector, corresponding to $\varphi'' = \varphi.NON-KEY-ARG: \{root(\varphi''.\pi_1), root(\varphi''.\pi_2), \dots, root(\varphi''.\pi_n)\}$, where ϕ is what φ has become after $\varphi' = \varphi.KEY-ARG$ has been unified with ψ .

As already mentioned, the main *problem* that arose when we tried to integrate the QC pre-unification test with the LIGHT compiler was its integration with the previously included main optimisation: the specialised compilation of rules. While the ‘key’ QC-vector φ' can be completely computed at grammar compilation/loading time, computing the QC-vector for $\varphi'' = \varphi.NON-KEY-ARG$ is not immediately possible simply because the $\varphi.NON-KEY-ARG$ structure does not effectively exist on the heap. This happens because in the LIGHT system syntactic rules are represented as feature structures, and their application is done in a bottom-up manner. In order to eliminate unnecessary copying, when dealing with LinGO-like grammars (working with only binary and unary rules), we have specialised each rule’s execution into *i.* key/head-corner mode application, and *ii.* complete mode application. This distinction between two different modes for one (binary) rule application — together with the FS environment-based sharing facility — allows for an incremental construction of the feature structure representing a phrase, in such a way that, if completion is finally not possible, then no space (otherwise needed in an interpreter framework) for constructing the FS corresponding to the rule’s complement/non-key argument is wasted. This strategy of incremental parsing in LIGHT is simple and elegant, due the use of open FSs as in the OSF constraint theory (Ait-Kaci and Podelski, 1993), in contrast with closed records used in the appropriateness-based approach (Carpenter, 1992) underlying other LinGO-parsing systems.

Table 1.1. The QC-vectors computed for a rule φ in the compilation approach.

QC test	'key' QC-vector	'complete' QC-vector	'passive' QC-vector
compilation time:	$preComp(\varphi') =$	$preComp(\varphi'')$	$preComp(\varphi)$
run time:	$= QC(\varphi')$	$QC(\phi)$	$QC(\phi')$

Specialised rule computation provided LIGHT a factor of speeding up of 2.75 on the CSLI test-suite.

Note that even in the hyper-active head-corner parsing approach proposed by (Oepen and Carroll, 2000), in which an indexing schema is used to minimise the copying of possibly unnecessary parts of a rule's FS (notably the non-key argument and the LHS of the rule), one initial full representation of the rule's FS must be constructed before applying the rule in order to fill its key-argument. In LIGHT a full FS representation of a rule is obtained only after the rule arguments have been successfully unified with FSs already present on the heap.

In order to solve the above *problem* — namely, that the computation of the 'complete' QC-vector is prevented by the missing representation of the non-key argument — we proposed firstly a rather *naive solution*: we relaxed the QC-test for the complete/non-key argument by checking the QC-vector of the candidate argument ϕ against the QC-vector computed for φ .NON-KEY-ARG. As this last QC-vector is more general than the one computed for φ'' — in the sense that if both φ .NON-KEY-ARG. π and φ'' . π exist, then $root(\varphi$.NON-KEY-ARG. $\pi) \preceq root(\varphi''$. $\pi)$ in the sort hierarchy — we were entitled to use it for QC.² However, using this method, the speed up effect of the QC test on parsing the CSLI test-suite with LIGHT was not significant.

The method we actually adopted was to compile (the computation of) the QC-vectors. For instance, in the interpreted approach for a binary rule one has to compute three QC-vectors: one 'key' QC-vector at the loading/pre-processing time, a 'complete', and finally a 'passive' one at the run-time. (These QC-vectors are shown on the bottom line in the somehow synoptic table 1.1.) Instead, in the compilation approach we compute five QC-vectors, among which three are computed at compilation time and two at run-time, the last two building upon the pre-computed ones. The quick check compilation will be presented in detail in the next two subsections. In the notation used in the table 1.1, ϕ is what φ became after the key argument (φ') was unified with ψ , the FS of a passive item, and ϕ' is what ϕ became after the non-key argument (φ'') was unified with ψ' the FS of another passive item.

2.2 Pre-computing QC vectors

So far we have shown that

- the QC test acts as a pre-unification filter for rule application; in interpreted-like parsing with LinGO-like grammars, rules are represented as FSs, and therefore computing whether a given FS will match the argument of a rule is straightforward;
- what makes pre-computation of QC necessary is that specialised compilation of rules in LIGHT eliminates the presence (of the full representation) of rule FSs from the heap.

Note that — assuming like in the head/key-corner parsing (Kay, 1989) that the key argument is parsed always before the non-key/complement arguments — the ‘key’ QC-vector, as introduced in the previous subsection for a certain rule φ is unique for all key-mode applications of that rule. All the other computed QC-vectors depend on the actual application of φ i.e., on the already parsed/filled arguments. However, one can see all these QC-vectors as computable in two stages/components: *i.* a pre-computed/preliminary form of QC-vectors, which can be computed at compilation time independently of the FS that will be eventually unified with the arguments; and *ii.* the actual, form/content of QC-vectors will be filled at run time starting from the pre-computed forms, dependent on the already parsed arguments.

The QC test main idea as introduced in the previous subsection is easy to formalise: given a finite set of feature paths $\Pi = \{\pi_1, \dots, \pi_m\}$, and the feature structures ψ_1, ψ_2 , check whether

$$root(\psi_1.\pi_i) \wedge root(\psi_2.\pi_i) \neq \perp, \text{ for } i = 1, \dots, m.$$

It will be assumed by definition that $root(\psi.\pi) = \top$ if the feature path π is undefined for ψ (this fact will be denoted as $\psi.\pi \uparrow$).

It turns out that if $\pi' = f_1. \dots .f_j$ is the longest prefix of $\pi = f_1. \dots .f_n$ such that $\psi.\pi'$ is defined ($\psi.\pi' \downarrow$), and $root(\psi.\pi') = s_j$, then we can improve our definition of QC-values and take $root(\psi.\pi) = s_n$, where $s_{j+1} = \Psi(s_j).f_{j+1}$, $s_{j+2} = \Psi(s_{j+1}).f_{j+2}$, ..., $s_n = \Psi(s_{n-1}).f_n$, where $\Psi(s)$ is the type associated with the sort s in the input grammar. Of course, $\Psi(s).f$ has to be considered \top if f is not defined at the root level in $\Psi(s)$. Alternatively, we could try to expand ψ by local unfolding, i.e. unifying $\psi.\pi'$ with (a copy of) the type $\Psi(s_j)$ provided by the grammar. If necessary, further local expansion/unfolding can be done. Note that local expansion/unfolding provides more refined constraints, so it is good to use this information for the the pre-computation of QC-vectors. But it

$$preComp(\psi, \pi) = \begin{cases} s : \text{sort} & \text{if } root(\psi, \pi) = s, \text{ and} \\ & \psi.\pi' \notin \mathcal{X}, \text{ for any prefix } \pi' \text{ of } \pi; \\ & \text{at the run time } \underline{QC_{\pi}(\psi) = s}; \\ i : \text{int} & \text{if } \psi.\pi \downarrow, \psi.\pi = X_i, \text{ and} \\ & \psi.\pi' \in \mathcal{X}, \text{ for a prefix } \pi' \text{ of } \pi; \\ & \text{at the run time } \underline{QC_{\pi}(\psi) = \text{heap}[X_i].\text{SORT}}; \\ -j : \text{int} & \text{if } \psi.\pi \uparrow, \\ & \psi.\pi' \in \mathcal{X}, \text{ for a prefix } \pi' \text{ of } \pi, \\ & \pi' = f_1 \dots f_j \text{ is the longest prefix of } \pi \text{ such that} \\ & \psi.\pi' \downarrow, \text{ and } \psi.\pi = X_j; \\ & \text{at the run time } \underline{QC_{\pi}(\psi) = \text{heap}[X_j.f_{j+1} \dots f_n].\text{SORT}}. \end{cases}$$

Figure 1.1. $QC_{\pi}(\psi)$ as function of $preComp(\psi, \pi)$.

is not a good idea to involve these refined constraints in the computation of QC-vectors at run time, since it would consume additional time and space. At run time, the previous solution, based on appropriateness constraints is preferable.

We distinguish the following three cases in (pre-)computing the QC-vectors:

1. If ψ is a rule argument without containing references to preceding³ arguments — this is the case of the head-corner argument in head-corner chart-based parsing — then we define

$$preComp(\psi, \pi) = s : \text{sort}, \text{ where } s = root(\psi, \pi).$$

2. If ψ is a rule argument with references to substructures of the preceding arguments, \mathcal{X} is the set of all variables/tags in ψ which refer to preceding arguments (according to the parsing order), and $\pi = f_1 f_2 \dots f_n$ is a feature path, then assuming that the heap is the (main) data structure used for the internal representation of FSs, we define the values of QC-vectors stating from their pre-computed form ($preComp$) like in Figure 1.1. (The underlined expressions in Figure 1.1 are in fact thought as extended to $QC_{\pi}(\varphi) = \dots$, for any feature structure φ subsumed by ψ .) If — as it is the case in the current implementation of LIGHT — the computation of certain QC-vectors is delayed until really needed, then the actual values of the variables X_i, X_j representing addresses/indices of heap cells will have to be saved (together with those in the set \mathcal{X}) in the environment associated with the preceding argument (saved after it has been parsed), so to make them available to the current argument.

3. If ψ is the feature structure corresponding to a non-unary rule instance, the ‘passive’ QC-vector corresponding to that instance is defined in a similar way to the one detailed above, with the only one difference

```

sentence
[ ARGS < vp
  [ HEAD #1:verb
    [ AGREEMENT #3:agr ],
    OBJECT np ],
  #2:np
  [ HEAD noun
    [ AGREEMENT #3 ] ] ],
HEAD #1,
SUBJECT #2 ]

```

Figure 1.2. The OSF-term associated to a `sentence` rule.

Table 1.2. The active (‘key’ and ‘complete’) QC-vectors for the `sentence` rule.

QC-paths	‘key’ QC-vector	preComp	‘complete’ QC-vector
$\pi_1 = \text{HEAD}$	<i>verb</i>	<i>noun</i>	<i>noun</i>
$\pi_2 = \text{OBJECT}$	<i>np</i>	#2.OBJECT	\top
$\pi_3 = \text{HEAD.AGREEMENT}$	<i>agr</i>	#3	<i>3sg</i>

that \mathcal{X} is taken as the set of all variables/coreferences shared between the rule’s *LHS* and the arguments (*RHS*).

We mention that in the LIGHT system, a pre-computed QC-vector is stored as an array of tuples of the form (s, sort) , (i, int) , $(-j, \text{int})$, with $i \geq 0$, and $j > 0$, while at run-time a QC-vector is represented simply as an array of sorts.

2.2.1 Example.

Let us consider — adapted from (Sikkel, 1997) — a simple rule made of a context-free backbone $s \rightarrow np *vp$ augmented with feature constraints like in Figure 1.2. (The $*$ sign marks the rule’s head/key argument.) Suppose that we want to consider the failure paths $\pi_1 = \text{HEAD}$, $\pi_2 = \text{OBJECT}$, $\pi_3 = \text{HEAD.AGREEMENT}$. The ‘key’ QC-vector and the two ‘complete’ QC-vectors — the *preComp* form and respectively the final form — are shown in the table 1.2. The *preComp* QC-vector is shown in a more intuitive form than in the formalisation given in Subsection 2.2. The final, complete QC-vector corresponds to the (expected) analysis of the sentence *The cat catches a mouse*. Note that in the complete QC-vector, the value for π_2 is \top since the FS corresponding to the noun phrase *a mouse* does not have the OBJECT feature defined.

<pre> vp [ARGS < catches [HEAD #7:verb [AGREEMENT #5:3sg], OBJECT #6:np [ARGS < a [HEAD det], mouse [HEAD #4:noun [AGREEMENT 3sg]] >, HEAD #4], SUBJECT #8:sign [HEAD top [AGREEMENT #5]]], #6 >, HEAD #7, SUBJECT #8] </pre>	<table style="width: 100%; border-collapse: collapse;"> <thead> <tr style="border-top: 1px solid black; border-bottom: 1px solid black;"> <th style="text-align: left; padding: 2px;"><i>paths</i></th> <th style="text-align: left; padding: 2px;"><i>QC-vector</i></th> </tr> </thead> <tbody> <tr> <td style="padding: 2px;">π_1</td> <td style="padding: 2px;"><i>verb</i></td> </tr> <tr> <td style="padding: 2px;">π_2</td> <td style="padding: 2px;">⊥</td> </tr> <tr style="border-bottom: 1px solid black;"> <td style="padding: 2px;">π_3</td> <td style="padding: 2px;"><i>3sg</i></td> </tr> </tbody> </table>	<i>paths</i>	<i>QC-vector</i>	π_1	<i>verb</i>	π_2	⊥	π_3	<i>3sg</i>
<i>paths</i>	<i>QC-vector</i>								
π_1	<i>verb</i>								
π_2	⊥								
π_3	<i>3sg</i>								
<pre> np [ARGS < the [HEAD det], cat [HEAD #9:noun [AGREEMENT 3sg]] >, HEAD #9] </pre>	<table style="width: 100%; border-collapse: collapse;"> <thead> <tr style="border-top: 1px solid black; border-bottom: 1px solid black;"> <th style="text-align: left; padding: 2px;"><i>paths</i></th> <th style="text-align: left; padding: 2px;"><i>QC-vector</i></th> </tr> </thead> <tbody> <tr> <td style="padding: 2px;">π_1</td> <td style="padding: 2px;"><i>noun</i></td> </tr> <tr> <td style="padding: 2px;">π_2</td> <td style="padding: 2px;">⊥</td> </tr> <tr style="border-bottom: 1px solid black;"> <td style="padding: 2px;">π_3</td> <td style="padding: 2px;"><i>3sg</i></td> </tr> </tbody> </table>	<i>paths</i>	<i>QC-vector</i>	π_1	<i>noun</i>	π_2	⊥	π_3	<i>3sg</i>
<i>paths</i>	<i>QC-vector</i>								
π_1	<i>noun</i>								
π_2	⊥								
π_3	<i>3sg</i>								

Figure 1.3. The parses corresponding to the *vp catches a mouse* and the *np the cat*, and the computed ‘passive’ QC-vectors.

One can easily see that the *vp catches a mouse*, as shown in Figure 1.3, passes the QC test with the key QC-vector presented in the table 1.2. Then the *np FS* shown in Figure 1.3 for the noun phrase *the cat* passes the QC test in conjunction with the complete QC-vector in the table 1.2, but the (slightly different) FS for *the cats* wouldn’t, due to an (AGREEMENT) inconsistency on the path π_3 (*non-3sg* vs. *3sg*). The sorts *non-3sg* and *3sg* are both assumed to be subsorts of *agr*.

2.3 From pre-computed QC to compiled QC

After getting the *preComp* vectors at compilation time, we must find the right place to put together *i.* the QC-vectors computation, and *ii.* the QC test within the compiled rule’s code or, alternatively, into the sequence containing a call to the rule’s application.

Let us consider ψ the FS corresponding to a rule and φ the FS (corresponding to a passive item) to be unified with the next-to-be-parsed argument. For LinGO, which deals only with binary and unary rules,

1. for the rule's head-corner/key argument, (*i.*) its QC associated vector is computed at compile time, and (*ii.*) the QC test can be compiled as a sequence of conditional statements of the form

if (glb(s_π , $QC_\pi(\varphi) = \perp$) return FALSE;

where $s_\pi = QC_\pi(\psi.\text{KEY-ARG}) = preComp(\psi.\text{KEY-ARG}, \pi)$ is known at compile time.

2'. if ψ is binary rule, and (after the QC test) φ unifies successfully with the rule's head-corner argument, then before building (and saving) the corresponding environment, we have to (*i.*) compute the QC-vector for the non-head-corner argument:⁴

set $QC_\pi(\psi')$, t_π

where

$\psi' = \psi.\text{NON-KEY-ARG}$, and

$$t_\pi = \rho(preComp(\psi', \pi)) = \begin{cases} s & \text{if } preComp(\psi', \pi) = s : \text{sort}; \\ \text{heap}[X_i].\text{SORT} & \text{if } preComp(\psi', \pi) = i : \text{int}, i \geq 0; \\ \text{heap}[\text{path}(\pi, j, X_j)].\text{SORT} & \text{if } preComp(\psi', \pi) = -j : \text{int}, j > 0 \end{cases}$$

and $path(\pi, j, \psi')$ computes the value for the path $f_{j+1} \dots f_n$ inside the FS ψ' , starting from the node X_j . (Like in the previous subsection, $\pi = f_1 \dots f_n$.)

2''. if ψ is a binary rule, and φ is a candidate for its non-head-corner argument, before restoring the environment for the item corresponding to φ , we have to (*iii.*) perform the QC test, in fact a sequence of conditional statements of the following form, one for each QC path π :

if (glb($QC_\pi(\psi.\text{NON-KEY-ARG})$, $QC_\pi(\varphi) = \perp$) return FALSE;

Note that $QC_\pi(\psi.\text{NON-KEY-ARG})$ was already computed (see 2').

3. if the rule ψ was successfully completed, then we have to (*i.*) compute the 'passive' QC-vector for the newly created item/FS: we proceed like above (2''), with the single difference that instead of $preComp(\psi', \pi)$ we have to consider $preComp'(\psi, \pi)$, where $preComp'$ is computed similarly to $preComp$, but taking \mathcal{X} as the set of all variables used in the rule's arguments (as already noticed at the point 3 of the previous subsection, when we presented the pre-computed QC-vectors).

2.4 Compiled QC — Evaluation and possible improvements

When running the LinGO grammar on the CSLI test-suite without the Quick Check pre-unification filter, the LIGHT system took 0.07 sec/sentence. With Quick Check turned on, LIGHT registered 0.04

sec/sentence. The compiled QC filter in LIGHT thus provided a speed-up factor of 42%. The tests were run on a SUN Sparc server at 400MHz. The optimal set of failure paths contained 43 paths with lengths between 2 and 14 features. As expected this factor is lower than the speed-up factor of simple, interpreted QC (63% for PET) because compiled unification is already significantly faster than interpreted unification. In other words, one has to keep in mind that in LIGHT the specialised compiled form of rules already significantly speeds up parsing, before applying the QC filter. However this factor can be further increased by implementing the *improvements* outlined below. (Some of those improvements apply also to interpreter-like parsing systems.)

The run-time QC test can be incorporated into the functions “encapsulating” the rules compiled code as a sequence of if statements. This would have the following advantages, which further improve QC-filter efficiency:

- tests like $\top \wedge \text{root}(\phi)$, which in fact correspond to paths that are not fully defined in the argument being currently checked, may be eliminated since they always succeed;
- when using appropriateness constraints (Carpenter, 1992), tests on $s \wedge \text{root}(\phi.f)$ may be eliminated if s is the maximal appropriate sort for the feature f ;
- certain parts in the *preComp* vectors overlap; subject to the failure paths’ order, the definition of these vectors can be improved so to eliminate duplicate work:

if $\text{QC}_\pi(\psi) = \text{heap}[X_j.f_{j+1} \dots .f_k.f_{k+1} \dots .f_n].\text{SORT}$, and $\text{QC}_{\pi'}(\psi) = \text{heap}[X_j.f_{j+1} \dots .f_k.f'_{k+1} \dots .f'_m].\text{SORT}$, then $\text{QC}_{\pi'}(\psi)$ can be computed as $\text{heap}[Y_l.f_{j+l} \dots .f_k.f'_{k+1} \dots .f'_m].\text{SORT}$, where Y_l is the last Y definable variable in the sequence $Y_1 = X_j.f_1, \dots, Y_k = Y_{k-1}.f_k$ is the sequence used to compute $\text{QC}_\pi(\psi)$;⁵

- the sort *glb* tests (represented by the if statements) can be re-ordered, depending on the applied rule and the type of the filtered argument, because most probable failure paths at the grammar-level are not necessarily most probable failure paths for each rule and argument.

Indeed, one of the main *criticisms* that can be made of the QC-filter technique in the form presented in the beginning of this subsection — and used as such in the LKB, TDL and PET systems — is that it is a grammar-level technique, in the sense that the QC-paths to be tested are grammar+corpus deduced, but they are not “personalised” at the

rule and argument level. However, one can compute such QC-vectors so to be rule+argument dependent. A disadvantage still remaining is that the QC-vectors associated with a passive item (completed rule) must contain all paths addressed by those rules and arguments for which that completed rule/passive item is a potential candidate. Therefore it is unlikely that the dimensionality of the “personalised” QC-vectors would be significantly reduced.

To our knowledge, the compilation schema we presented here for quick check is the first attempt to incorporate this pre-unification speed-up technique in a compiler system dealing with large-scale unification grammars. The QC compilation is by no means HPSG dependent. Moreover, it is basically independent of the variant of (order-sorted) feature constraint logics that supports parsing (which in turn calls unification). In LIGHT we used as logic background the order- and type-consistent OSF-theories (Ciortuz, 2002b) for which OSF-theory unification is equivalent to unification of typed FSs defined by (Carpenter, 1992). Therefore, the technique here presented can be applied to other systems dealing with typed-unification grammars like AMALIA (Wintner and Francez, 1999) and LiLFeS (Miyao et al., 2000).

3. Generalised Rule Reduction

This section presents Generalised Reduction (GR), a learning technique for speeding up parsing typed-unification grammars, based on generalising feature values. GR eliminates as many as possible of the feature constraints (FCs) from the type feature structures (FSs) while applying the criterion of preserving the parsing results on a given, training corpus. For parsing with GR-restricted rule FSs, and for checking the correctness of obtained parses on other corpora, one may use a form of FS unification which we call two-step unification. We will report results of GR application on the LinGO English grammar.

Below we formally define generalised reduction and present the idea of 2-step unification. Then Section 3.1 presents our GR algorithms, and Section 3.2 reports the measurements we did in getting and using GR-versions of the LinGO grammar. Section 3.3 outlines further work, about doing quick check on generalised reduction paths.

From the logical point of view, FSs can be viewed as positive OSF-clauses, which are finite sets of atomic OSF-constraints (Aït-Kaci and Podelski, 1993), namely sort constraints, feature constraints and equation constraints. Therefore the generalisation of FSs can be logically achieved through elimination of some atomic constraints from the FS. The approach we followed in developing both the generalisation and

specialisation procedures for learning feature values in typed-unification grammars is basically the one proposed by Inductive Logic Programming (ILP) (Muggleton and Raedt, 1994). We have adapted/applied the main ILP ideas to OSF-logic, the feature constraint (subset of the first-order) logic underlying such grammars (Aït-Kaci and Podelski, 1993), (Carpenter, 1992), (Aït-Kaci et al., 1997). We showed in (Ciortuz, 2002d) how one can improve the linguistic competence — i.e., enhancing the coverage — of a given typed-unification grammar by guided generalisation, using parsing failures.

Generalised reduction is a restricted form of FS generalisation that eliminates as many as possible of the feature constraints from a type FS of a given unification grammar while applying an *evaluation criterion* for maintaining the parsing results on a large corpus/test-suite. Note that although GR explicitly eliminates only feature constraints, it may be the case that sort and/or equation constraints associated with the value of an eliminated feature constraint get implicitly eliminated through FS normalisation (Aït-Kaci and Podelski, 1993).

We make the observation that the notion of *restriction* in the HPSG literature designates the elimination of certain (few) features following the application of a parsing rule. Generalised reduction extends this operation on arbitrary chosen features (of course, ensuring the parsing correctness on the given training corpus). We prefer the name *reduction* for this operation, since from the logical point of view, eliminating one feature constraint from a FS logically generalises that FS, while its set of feature paths indeed gets *restricted*. However, we will denote the result of applying GR to a FS as the *GR-restricted form* or the *GR-learnt form* of that FS. Alternatively, one may think of generalised reduction as *relative unfilling*. Indeed, *expansion* for typed-unification grammars — the procedure that propagates both top-down in the type hierarchy and locally in the typed FS the constraints associated to types in the grammar — is usually followed by *unfilling* (Gerdemann, 1995), a feature constraint removal technique which is corpus-independent. Generalised reduction may be thought as a corpus/test-suite relative unfilling technique.

GR is shown not only to improve the performance of the given grammar, since it maximally reduces the size of the (rule) FSs in grammar, but also adds an interesting improvement to the parsing system design. As measurements on parsing the CSLI test-suite have shown that on average only 8% of the items produced on the chart during parsing constitute part of the full parses, one can devise unification as a two-step operation. Parsing with *two-step unification* will *i.* use the GR-learnt form of the grammar rules to produce full parses, and *ii.* eventually com-

plete/check the final parses using the full (or, rather: complementary) form of rule FSs.

The second/late unification step performs in fact *consistency checking* with respect to the full form of rule FSs. From this point of view it plays a symmetrical role to the quick check pre-unification filter presented in Section 2. From an implementation point of view, we can first emulate the second/late unification step as type-checking, via FS unfolding. As further work, we suggest that different techniques may be used to apply more efficiently the unification’s second phase.

3.1 Generalised Reduction of Rule FSs — Algorithms

We present two algorithms for generalised reduction of types in unification grammars. The first one, called A below is a simple, non-incremental one. From it we derived a second, incremental algorithm (B) that we have optimised. Both algorithms have roughly the same kind of input and output.

Input: \mathcal{G} , a typed-unification grammar, Θ a test-suite i.e., a set of sentences annotated by a *parsing evaluation* function;

Output: a more/most general grammar than \mathcal{G} obtained by generalised reduction of FSs, and producing the same parsing evaluation results as \mathcal{G} on Θ .

The measurements provided below use the following *parsing evaluation criterion*: the number of full parses, the number of attempted unifications and the number of successful unifications must be preserved for each sentence in the test-suite, after the elimination of a (selected) feature constraint. Due to the large size of both the LinGO grammar and the test-suite used for running GR, this parsing evaluation criterion is a good approximation of the following “tough” criterion: the set of actual parses (up to the associated FSs) delivered for each sentence by the two grammar versions must be exactly the same. We make the remark that after applying GR on the LinGO using the CSLI test-suite, this “tough” criterion was satisfied. The parsing evaluation criteria to be used in getting the GR-restricted form of a grammar must actually be combined with one regarding memory/space resources exhaustion: if for a given sentence, after elimination of a feature constraint the resources initially allocated for parsing are exhausted, then that feature constraint is kept in the grammar.

Although in our experiments only the rule types in \mathcal{G} were subject to generalised reduction, the formulation of the two GR algorithms can be extended to any FS in the input grammar. Applying GR to the type FSs

Procedure A:

```

for each rule  $\Psi(r)$  in the grammar  $\mathcal{G}$ 
  for each feature constraint  $\varphi$  in  $\Psi(r)$ 
    if removing  $\varphi$  from  $\Psi(r)$ 
      preserves the parsing (evaluation) results
        for each sentence in the test-suite  $\Theta$ 
          then  $\Psi(r) := \Psi(r) - \{\varphi\}$ ;

```

Figure 1.4. A simple, non-incremental Generalised Reduction procedure.

used in type-checking is easy, but if parsing correctness must be ensured — i.e., elimination of over-generalised parses is required in the end — then things get more complicated than for rule FSs. Extending GR to lexical entries is even more demanding if lexical rules are used.

3.1.1 A simple Generalised Reduction procedure.

A simple procedure for deriving the generalised reduction of a grammar is given in Figure 1.4. Basically, each feature constraint (FC) in the grammar’s rule FSs is tested for elimination. If removing a constraint φ does not change the parsing (evaluation) results on the test-suite Θ , then φ will be eliminated from \mathcal{G} , for the purpose of constructing the grammar’s GR-restricted form. The algorithm constructs monotonic generalisations of the grammar \mathcal{G} . From the logical point of view, $\mathcal{G}_n \models \mathcal{G}_{n+1} \models \dots \models \dots \models \mathcal{G}_1 \models \mathcal{G} = \mathcal{G}_0$.

Obviously, the GR result is dependent on the order in which feature constraints are processed: the elimination of the constraint φ can block the elimination of the constraint φ' if φ is tried first, and vice-versa. Therefore usually there is no unique most general GR form for a given grammar. In the actual implementation of the GR algorithms we first worked on the elimination of feature constraints from key arguments (in the decreasing order of their “usage” frequency on the given test-suite), then from non-key arguments (according to the same kind of frequency), and finally from rule LHS sub-structures. Inside a “reduction partition” (key argument, non-key argument, LHS structure) feature constraints were tested for elimination following the bottom-up traversal order of the acyclic rooted graph representing the rule FS.

We have to *remark* that the first two **for**’s in the procedure A can be combined into a single **for** iterating over the set of all feature constraints $\langle r, \varphi \rangle$ in the grammar’s rules ($\varphi \in \Psi(r)$).

Procedure B:

```

do 0.  $\mathcal{G}_0 = \mathcal{G}$ ,  $i = 0$ ;
   1. Apply the procedure A on a sentence  $s$  from  $\Theta$ ;
      let  $\mathcal{G}_{i+1}$  be the result
   2. eliminate from  $\Theta$  the sentences for which  $\mathcal{G}_{i+1}$ 
      provides the same parsing results as  $\mathcal{G}$ 
until  $\Theta = \emptyset$ .

```

Figure 1.5. An improved, incremental Generalised Reduction procedure.

3.1.2 An improved, incremental GR procedure.

The simple GR algorithm presented in Figure 1.4 can be substantially improved by:

- reversing the two **for**'s, namely the one identified in the above *remark* and the one iterating over the sentences in the test-suite. The advantage is that those sentences which become/are correctly parsed by \mathcal{G}_i — an intermediate generalised form of \mathcal{G} — need not to be tried again when computing \mathcal{G}_{i+1} ;
- getting a first GR version of the input grammar by running the simple procedure A on one or more sentences, and subsequently improving it iteratively.

The improved procedure is given in Figure 1.5.

It should be noted that the set of FCs to be tried for elimination in procedure A called at step 1 — see the last remark following the presentation of the algorithm A — is provided by $\mathcal{G} - \mathcal{G}_i$, therefore it becomes smaller each time around the loop. Indeed, from the logical point of view, $\mathcal{G} = \mathcal{G}_0 \models \dots \models \mathcal{G}_{n+1} \models \mathcal{G}_n \dots \models \mathcal{G}_2 \models \mathcal{G}_1$. Viewed as set of constraints, $\mathcal{G} = \mathcal{G}_0 \supset \dots \supset \mathcal{G}_{i+1} \supset \mathcal{G}_i \dots \supset \mathcal{G}_2 \supset \mathcal{G}_1$ therefore $\mathcal{G} - \mathcal{G}_{n+1} \subset \mathcal{G} - \mathcal{G}_n$.

We have made a number of *improvements* to the algorithm in our implementation:

- At step 2, the elimination of sentences from Θ is done in a lazy manner: initially Θ is sorted according to the number of failed unifications per sentence when using \mathcal{G} , — therefore the sentence s chosen at the step 1 may be considered the first sentence in Θ_i — and subsequently only the first sentences from Θ_i which are correctly parsed by \mathcal{G}_{i+1} are eliminated. (Θ_{i+1} starts with the first sentence in Θ_i which causes over-parsing with \mathcal{G}_{i+1} . Obviously, we take $\Theta_0 = \Theta$.) The reason for lazy elimination of sentences from Θ is the significant time consumption caused by over-parsing

and/or exhaustion of allocated resources that may occur — very frequently, in the beginning — for some of the the remaining sentences in Θ due to the (premature) elimination of certain constraints in preceding loops in procedure B.

- Only FCs from rules involved in the parsing of the sentence s at step 1 must be checked for elimination. If the sentence exhausted the allocated resources for parsing, this optimisation does not apply, because it is not possible to tell in advance which rules might have been used if exhaustion had not been reached.
- A “preview” test which decides whether a whole rule partition is immediately learnable is highly effective in improving the running time of the GR procedure with LinGO. This test means eliminating from $\mathcal{G} - \mathcal{G}_i$ in a single step all the constraints in a partition (an argument or LHS substructure) for the rules previously identified as contributing to parsing the sentence s . At rule level, identifying the FCs that must be retained in \mathcal{G}_{i+1} may be further speeded up through halving the FC search space.
- As the feature constraints in a rule FS are implicitly ordered by their position in the rooted acyclic graph representing the rule, another optimisation is possible: if all ancestors of a FC have been approved for elimination, than that FC may be eliminated immediately, assuming that its value is not coreferenced in a subsequent partition (argument of LHS substructure), and this fact that can be determined in the preparation of the GR application.

In our current implementation of procedure B, the *exhaustion* of resources allocated for parsing is controlled by a sentence-independent criterion. Currently, the allocated resources allow for the complete parsing of all sentences in the test-suite. But using such a criterion has the following drawback: if procedure A causes strong over-parsing (possibly looping) for a sentence, then this fact is detected eventually only after *all* resources are exhausted. We suggest a better, more punctual criterion for evaluating the consumption of allocated resources for parsing: *i.* initially, for each sentence in the given test-suite register the parsing time using \mathcal{G} ; *ii.* if during the generalised reduction, parsing a sentence using the current generalised form of rules consumes more than n times the initially registered parsing time without finishing, then consider that sentence as a resource exhausting one.

Incorporating all but the last improvement mentioned above, the procedure B required 61 minutes on a Pentium III PC at 933MHz running

Table 1.3. Comparison between the results of applying the two GR procedures on LinGO, the CSLI test-suite.

<i>GR procedure</i>	<i>A</i>		<i>B</i>	
FC reduction rate: average	58.92%		56.64%	
GR vs. total feat. paths in rule arg.	234/494	(47.36%)	318/494	(64.38%)
<i>average parsing time, in msec.</i>				
using full-form rule FSs			21.617	
1st-step unification (reduction %)	16.662	(22.24%)	16.736	(22.07%)
emulated 2-step unification (red. %)	18.657	(13.12%)	18.427	(14.20%)

Red Hat Linux 7.1. (All measurements reported in the next subsection were done also on that PC.) We expect that this amount of time will be approximately halved by using a parser which incorporates quick check; our GR procedure cannot use rules in Specialised Rule Compilation (SRC) format, and the non-SRC parser in LIGHT currently does not use QC.

Because the test-suite for training is processed incrementally only by algorithm B, we refer to it as the *incremental* GR algorithm, while algorithm A is called the *non-incremental* one.

3.2 Generalised Reduction — Measurements and Comparisons

When running GR algorithm A, the reduction of the number of FCs in rule argument sub-structures of LinGO was impressive: 61.38% for key/head arguments and 61.85% for non-key arguments (see Table 1.3). The number of feature paths in rule argument sub-structures was reduced from 494 to 234, revealing that the decisive contribution to unification failures for LinGO on the CSLI test-suite is restricted to less than half of the feature paths in the arguments. We will call those feature paths *GR-paths*.

This result complements the view provided by the QC technique: only 8.5% of the total feature paths in the arguments are responsible for most of the unification failures during parsing with LinGO. As expected, most of the (43) QC-paths we are using in LIGHT for LinGO are among the GR-paths identified by the procedure A; the QC-paths which are not GR-paths are extensions of maximal GR-paths.

While the average number of FCs eliminated from LinGO by GR algorithm B and the parsing performance on the resulting grammar version are only slightly different than those provided by algorithm A, the num-

Table 1.4. A snapshot view on reduction of memory usage when parsing the CSLI test-suite with the GR-restricted form of LinGO.

	<i>full-form rules</i>	<i>GR-restricted rules</i>	
		<i>first-step unification</i>	<i>emulated 2-step unif.</i>
heap cells	101614	38320 (62.29%)	76998 (24.23%)
feature frames	60303	30370 (49.64%)	58963 (02.22%)

Table 1.5. Results of running the parameterised GR-procedure B on the CSLI test-suite; n designates the number of sentences used in the inner (A) loop.

n	<i>running time</i>	<i>FCs reduction rate</i>	<i>GR-paths</i>
1	1h 27min	56.64	318
2	1h 36min	59.38	200
4	2h 27min	59.42	187
8	4h 03min	59.39	187

ber of GR-paths retained in rule arguments is significantly higher in the B case than in the A case. This difference is explained by test-suite fragmentation/atomisation on which the design of the procedure B was based.

The table 1.4 presents a snapshot of the most needed memory resources (and the corresponding percentage reduction) when parsing with full-form rule FSs compared with GR-restricted rule FSs. The graph in Figure 1.6 presents the evolution of the average rule reduction rate for obtaining the LinGO GR-version using algorithm B. Figure 1.7 illustrates the reduction of parsing time for the sentences in the CSLI test-suite when running the LinGO grammar with the GR-restricted rules. One can see in this last figure — especially for the sentences requiring many unifications — that although the number of unifications is increased, the total parsing time is reduced.

Algorithm B may be generalised so as to provide the inner loop (the call to the procedure A) with not only one but several (n) sentences which are incorrectly parsed by \mathcal{G}_{i+1} . If so, the processing time for getting the GR-version of the given grammar will increase. However, as the table 1.5 shows, this is a convenient way to get empirically a smaller number of computed GR-paths.⁶

We tested a GR-version of LinGO (produced using the CSLI test-suite) on the *aged* test-suite also provided with LinGO. This test-suite

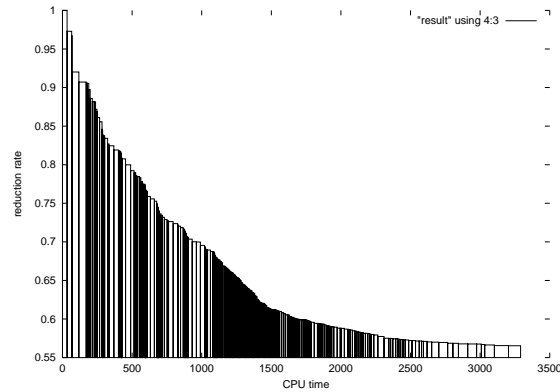


Figure 1.6. Procedure B: rule reduction rate vs. CPU time consumption on LinGO/CSLI.

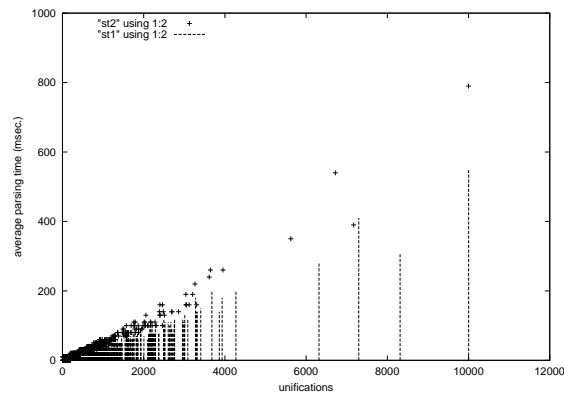


Figure 1.7. Comparisons: unifications vs. parsing time on the CSLI test-suite using LIGHT on LinGO: the full version vs. the GR-restricted version using 1st-step unification.

requires on average 4.65 times more unifications per sentence than the CSLI test-suite. The CSLI test-suite has 1348 sentences with an average length of 6.38 tokens and ambiguity 1.5 readings. For the *aged* test-suite, the average length is 8.5 and the ambiguity 14.14. Of the 96 sentences in the *aged* test-suite, 59 were correctly parsed, 3 exhausted the allocated resources, and 34 were over-parsed. The average parsing

precision for the sentences non-exhausting the allocated resources was 83.79%. In order to produce a GR-version of LinGO using the *aged* test-suite for training, procedure B needed 4h 44min, and the reduction rate was 65.60%. However, it took only 38min to further improve the previously obtained GR-version of LinGO (on the CSLI test-suite), using the *aged* test-suite, and the rule FS reduction rate went down only by 3.1%.

We investigated the effect of changing the ordering of the test-suite Θ for the GR algorithms. Applying algorithm A only to the most complicated sentence in the CSLI test-suite resulted in a 18% reduction in the number of FCs inside rule FSs. We obtained the same reduction rate running 30 iterations using the simplest sentences in the CSLI test-suite. Further generalising with the next (slightly less) complicated sentences in Θ will also take a lot of time, but may not significantly improve the grammar/rules reduction rate.

3.3 Further Work — Quick-check on GR-paths

As an alternative to “personalising” the quick check test for each rule as we proposed in Section 2.3, we can consider doing pre-unification filtering on GR-paths (instead of QC-paths). This would require the construction of GR-vectors associated with rules and their arguments. For LinGO, the size of GR-vectors would be several times larger than that of QC-vectors (187 vs. 43). We estimate however that the number of actual sort intersections during pre-unification tests is significantly lower than currently done by the quick check, as the size of rule argument FSs is greatly reduced by GR. The immediate advantage in doing pre-unification filtering on GR-paths is that the sort compatibility checks will fully “cover” the rule argument FSs. Moreover, the (compiled) computation of GR-vectors can be done directly by the abstract machine for FS unification (Ciortuz, 2002c), by augmentation of abstract instructions. Note that the few QC-paths which extend maximal GR-paths can be “injected” into the GR-restricted form of rules.

4. Conclusion

This paper presented our study of two interesting classes of feature paths inside rule feature structures of LinGO, a large-scale HPSG grammar for English.

The first class consists of those paths which most probably lead to unification failure during parsing a given corpus. The fact that there are only a relatively small number of such paths responsible for most of the unification failure during parsing enables the use of the quick

check pre-unification filter; hence the name QC-paths attributed to these paths (Malouf et al., 2000). Usually, prior to the application of the QC-test, the values of QC-paths in a given FS are stored in a (QC-)vector. Our contribution consists of the adaptation of this technique to a situation in which the computation of QC-vectors must precede the construction of the FS itself. Such a situation appears in parsing with compiled typed-unification grammars using the specialised rule compilation technique (Ciortuz, 2001a), as is the case with the LIGHT system (Ciortuz, 2002b). We show that a two-stage computation of QC-vectors is both feasible and effective, firstly getting a pre-computed/off-line form of the QC-vectors, and secondly computing their final/on-line form.

The second class of feature paths we have explored are generalised reduction paths in rule arguments. These are paths which contribute to unification failure. In the early stage of a rule application (via FS unification) in bottom-up parsing, checking constraints which are not on GR-paths may be delayed. Eventually the constraints are checked only on full, successful parses. For a LinGO-like grammar in which the percentage of GR-paths among all feature paths in rule arguments is high, applying generalised reduction and eventually two-step unification leads to a significant speed-up (supplementing the one provided by quick check), and an impressive reduction of memory usage. In the LIGHT system, to the 42% speed-up factor provided by QC on the LinGO grammar, GR may add a subsequent 22%. We described two GR algorithms for rule FSS, firstly a simple, non-incremental one, and then an optimised, incremental one. A procedure for automatic detection of cycles in parsing with generalised forms of the input typed-unification grammar would further improve the GR algorithms. Integrating linguistic knowledge and testing GR on other grammars and test-suites will provide additional insights into the use of this learning method.

Note that most QC-paths are GR-paths; the exceptions are extensions of maximal GR-paths. Generalised reduction limits the early unification stage/step to just the GR-paths, delaying the completion of unification and only applying it (eventually) to full parses. The quick check is a pre-unification filter; more exactly, in our design it operates in connection with the early/first unification step. Therefore the quick check and generalised reduction techniques are orthogonal. The second/late unification step acts as a post-parsing filter, and so it is symmetric (with respect to parsing) to the quick-check pre-unification filter.

Acknowledgments

The conception and implementation of the work on compilation of quick check was done while the author was employed at the Language Technology Lab of the German Research Center for Artificial Intelligence (DFKI) in Saarbrücken, Germany. The generalised reduction technique and the two papers (Ciortuz, 2001b) and (Ciortuz, 2002a) on which the present work is based were entirely elaborated while the author was supported by an EPSRC grant in the framework of an EPSRC ROPA project at the Computer Science Department of the University of York.

Thanks go to Ulrich Callmeier who implemented the interpreted form of the QC filter for CHIC/ago,⁷ the development prototype of the LIGHT compiler. Thanks to Stephan Oepen who pointed me to the precursor work of Maxwell and Kaplan. I wish to express special thanks to Professor Hans Uszkoreit for the support I received in order to get LIGHT designed and implemented during my employment at the Language Technology Lab of DFKI — the German Research Center for Artificial Intelligence in Saarbrücken, Germany. I am grateful also to Professor Steven Muggleton, Dr. Dimitar Kazakov, Dr. Suresh Manandhar, Dr. James Cussens and Dr. John Carroll who, in different ways, made possible my work on generalised reduction.

Notes

1. The LIGHT acronym stands Logic, Inheritance, Grammars, Heads, and Types. The analogy with the name of LIFE — Logic, Inheritance, Functions and Equalities — a well-known constraint logic language based on the OSF constraint system (Aït-Kaci and Podelski, 1993) is evident.

2. We used here the notation priorly established: φ is the FS associated to the rule which is being applied, and φ'' is obtained from φ after $\varphi' = \varphi.\text{KEY-ARG}$ was unified with ψ , the FS corresponding to a passive item.

3. Here the term “preceding” is used in the sense of the parsing order.

4. This QC-vector will be stored within the active item corresponding to the head-corner argument and will be used for the QC test at the rule’s completion attempt.

5. The $Y_1 = X_j.f_1, \dots, Y_k = Y_{k-1}.f_k$ sequence might not be entirely computed.

6. The running time of all GR algorithms can be significantly improved if, in the beginning of the GR application, all feature constraints not used when parsing the training test-suite are eliminated from the initial grammar. (Unfortunately, the current version of the LIGHT system cannot identify these unused FCs.) However, eliminating these FCs from the start will bias the set of GR-paths to be computed. In general, FCs initially not used in parsing the training test-suite may however appear in the GR-restricted grammar version.

7. CHIC stands for Compiling Hpsg Into C. The CHIC/ago name must be pronounced exactly like Chicago.

References

- Ait-Kaci, H. and Di Cosmo, R. (1993). Compiling order-sorted feature term unification. Technical report, Digital Paris Research Laboratory. PRL Technical Note 7, downloadable from <http://www.isg.sfu.ca/life/>.
- Ait-Kaci, H. and Podelski, A. (1993). Towards a meaning of LIFE. *Journal of Logic Programming*, 16:195–234.
- Ait-Kaci, H., Podelski, A., and Goldstein, S. (1997). Order-sorted feature theory unification. *Journal of Logic, Language and Information*, 30:99–124.
- Callmeier, U. (2000). PET — a platform for experimentation with efficient HPSG processing techniques. *Journal of Natural Language Engineering*, 6 (1) (Special Issue on Efficient Processing with HPSG):99–108.
- Carpenter, B. (1992). *The Logic of Typed Feature Structures – with applications to unification grammars, logic programs and constraint resolution*. Cambridge University Press.
- Ciortuz, L. (2001a). Compilation of head-corner bottom-up chart-based parsing with unification grammars. In *Proceedings of the IWPT 2001 International Workshop on Parsing Technologies*, pages 209–212, Beijing, China.
- Ciortuz, L. (2001b). On compilation of the Quick-Check filter for feature structure unification. In *Proceedings of the IWPT 2001 International Workshop on Parsing Technologies*, pages 90–100, Beijing, China.
- Ciortuz, L. (2002a). Learning attribute values in typed-unification grammars: On generalised rule reduction. In *Proceedings of the 6th Conference on Natural Language Learning (CoNLL-2002)*, Taipei, Taiwan. Morgan Kaufmann Publishers and ACL.
- Ciortuz, L. (2002b). LIGHT — a constraint language and compiler system for typed-unification grammars. In *Proceedings of the 25th German Conference on Artificial Intelligence (KI-2002)*, Aachen, Germany. Springer-Verlag.
- Ciortuz, L. (2002c). LIGHT — another abstract machine for feature structure unification. In Flickinger, D., Oepen, S., Tsujii, J., and Uszkoreit, H., editors, *Collaborative Language Engineering*. CSLI Publications, The Center for studies of Language, Logic and Information, Stanford University.
- Ciortuz, L. (2002d). Towards inductive learning of typed-unification grammars. In *Proceedings of the 17th Workshop on Logic Programming*. Dresden Technical University.
- Flickinger, D. P., Copestake, A., and Sag, I. A. (2000). HPSG analysis of English. In Wahlster, W., editor, *Verbmobil: Foundations of Speech-*

- to-Speech Translation*, Artificial Intelligence, pages 254–263. Springer-Verlag, Berlin Heidelberg New York.
- Gerdemann, D. (1995). Term encoding of typed feature structures. In *Proceedings of the 4th International Workshop on Parsing Technologies*, pages 89–97, Prague, Czech Republik.
- Kaplan, R. M. and Bresnan, J. (1983). Lexical-functional grammar: A formal system for grammatical representation. In Bresnan, J., editor, *The Mental Representation of Grammatical Relations*, pages 173–381. MIT Press.
- Kay, M. (1989). Head driven parsing. In *Proceedings of the 1st Workshop on Parsing Technologies*, pages 52–62, Pittsburg.
- Kiefer, B., Krieger, H.-U., Carroll, J., and Malouf, R. (1999). A bag of useful techniques for efficient and robust parsing. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics*, pages 473–480.
- Krieger, H.-U. and Schäfer, U. (1994). TDL – A Type Description Language for HPSG. Research Report RR-94-37, German Research Center for Artificial Intelligence (DFKI).
- Malouf, R., Carroll, J., and Copestake, A. (2000). Efficient feature structure operations without compilation. *Journal of Natural Language Engineering*, 6 (1) (Special Issue on Efficient Processing with HPSG):29–46.
- Maxwell III, J. and Kaplan, R. (1993). The interface between phrasal and functional constraints. *Journal of Computational Linguistics*, 19, Number 4:571–590.
- Mitsuishi, Y., Torisawa, K., and Tsujii, J. (1998). HPSG-Style Under-specified Japanese Grammar with Wide Coverage. In *Proceedings of the 17th International Conference on Computational Linguistics (COLING)*, pages 867–880.
- Miyao, Y., Makino, T., Torisawa, K., and Tsujii, J. (2000). The LiLFeS abstract machine and its evaluation with the LinGO grammar. *Journal of Natural Language Engineering*, 6 (1) (Special Issue on Efficient Processing with HPSG):47–61.
- Muggleton, S. and Raedt, L. D. (1994). Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19,20:629–679.
- Müller, S. (1999). *Deutsche Syntax deklarativ. Head-Driven Phrase Structure Grammar für das Deutsche*. Number 394 in Linguistische Arbeiten. Max Niemeyer Verlag, Tübingen.
- Nagata, M. (1992). An empirical study on rule granularity and unification interleaving: toward an efficient unification-based parsing system. In *Proceedings of COLING-92*, pages 177–183.

- Ninomyia, T., Makino, T., and Tsujii, J. (2002). An indexing scheme for typed feature structures. In *Proceedings of the 19th International Conference on Computational Linguistics: COLING-2002*.
- Oepen, S. and Callmeier, U. (2000). Measure for measure: Parser cross-fertilization. Towards increased component comparability and exchange. In *Proceedings of the International Workshop on Parsing Technologies IWPT-2000*, pages 183–194, Trento, Italy.
- Oepen, S. and Carroll, J. (2000). Performance profiling for parser engineering. *Journal of Natural Language Engineering*, 6 (1) (Special Issue on Efficient Processing with HPSG: Methods, Systems, Evaluation):81–97.
- Pereira, F. (1985). A structure-sharing representation for unification-based grammar formalisms. In *Proceedings of the 23rd meeting of the Association for Computational Linguistics*, pages 137–144, Chicago, Illinois.
- Pollard, C. and Sag, I. (1994). *Head-driven Phrase Structure Grammar*. CSLI Publications, Stanford.
- Shieber, S. (1985). Using restriction to extend parsing algorithms for complex feature-based formalisms. In *Proceedings of the 23rd Annual Meeting of the ACL*, pages 145–152, Chicago, Illinois.
- Shieber, S. M., Uszkoreit, H., Pereira, F. C., Robinson, J., and Tyson, M. (1983). The formalism and implementation of PATR-II. In Bresnan, J., editor, *Research on Interactive Acquisition and Use of Knowledge*. SRI International, Menlo Park, Calif.
- Siegel, M. (2000). HPSG analysis of Japanese. In *Verbmobil: Foundations of Speech-to-Speech Translation*, pages 264–279. Springer Verlag.
- Sikkel, N. (1997). *Parsing Schemata*. Springer Verlag.
- Tomabechi, H. (1992). Quasi-destructive graph unification with structure-sharing. In *Proceedings of COLING-92*, pages 440–446, Nantes, France.
- Uszkoreit, H. (1986). Categorical Unification Grammar. In *International Conference on Computational Linguistics (COLING'92)*, pages 498–504, Nancy, France.
- Wintner, S. and Francez, N. (1999). Efficient implementation of unification-based grammars. *Journal of Language and Computation*, 1(1):53–92.