

The Quick Check Pre-unification Filter for Typed Grammars: Further Advances

Liviu Ciortuz*

CS Department, University of Iași, Romania
E-mail: ciortuz@infoiasi.ro

Abstract. We present several significant improvements in the implementation of the quick check pre-unification filter [7] [10], and the potential way in which the design of the quick check [15] [17] can be further extended. We analyse the effect of these extensions on LinGO, the large-scale HPSG grammar for English [14]. Although these developments were done for the compiler system LIGHT [8], most of them are transferrable to non-compilation based systems.

1 Introduction

The so called quick check (henceforth QC) pre-unification filter for feature structure (FS) unification was introduced by [15]. For two FSs whose unifiability is to be checked, QC is performed as a pair-wise consistency test on the root sorts of certain substructures. These substructures in the two FSs are identified as the values of certain paths (henceforth QC-paths), which have been previously determined as the most probable to lead to unification failure.

QC is considered the most important speed-up technique in the framework of non-compiled FS unification [17]. [6] offers the first view on compiling QC and integrating it into LIGHT [8][11], a compilation-based parsing system for unification grammars. [7] and [10] show that, for a large (typed-)unification grammar like LinGO, the wide-coverage HPSG grammar for English [14], the set of QC-paths is essentially a subset of a larger, interesting set of paths, the so called GR-paths.

GR stands for Generalised Reduction, a machine learning technique that, while using a training test suite, identifies the set of paths that contribute (in one or several steps) to unification failure. These are the GR-paths. If we retain only these paths in the FSs representing the grammar rules, then the grammar get “reduced” (we will say: GR-reduced), and on the training test suite it will yield the same results (up to the underlying FSs) as the original grammar. For other test suites or corpora, the parsing can be naturally split into two steps: a first step which uses the rules in the GR-reduced form (and therefore possibly over-generates), and a second step which eliminates the over-generated parses by type-checking [4], using the full form of the rule FSs. [10] measures the simple combination of the two techniques — QC filtering, and GR-based two step parsing — showing that together, they lead to a 56% speed-up in the LIGHT parsing system on the CSLI test suite.

The present work further investigates the compiled form of the QC pre-unification filter in two interesting respects:

1. the computation of QC-vectors (i.e. the root sorts of QC-paths values) is speeded-up through

* This paper was published in the Scientific Annals of “Al.I.Cuza” University of Iași, Computer Science series, 2004, pag. 36–50.

- a second compilation phase which eliminates much of the redundancy that appears in on-line computation of QC-path values;²
 - the “personalised” application of the QC test:
 - the order in which QC-vectors are traversed during the (pre-unification) sort consistency test is made rule-dependent;
 - it can be determined at compilation time that certain QC-values (inside the QC-vectors associated to FSs) will never be used, therefore their computation could be skipped, leading to additional saved time.
2. exploring the virtues and limits of the QC filter by
- examining some alternatives to pre-unification filter, which in fact can be seen as complements to the (now ‘classical’) QC test: consistency sort check on coreferenced paths, and pre-unification type-checking;
 - combining QC with (syntactic) rule filtering, another optimisation included in the design of a (compiled or non-compiled) chart-based bottom-up active parsing system with typed grammars: delaying the construction of one rule’s LHS substructure for a (successfully created) passive item until it is needed for building another item.³
 - performing the QC test using GR-paths instead of the classical QC-paths, and storing the GR-path values so to easily retrieve the values of X variables employed by the *READ* instructions in the compile code of rule FSs.

Concerning the first point from above, to be detailed in Section 2, we will show that those developments yielded for the LIGHT compiler system a further 9% speed-up w.r.t. the average (full) parsing time on the CSLI test suite. The second point from above will be presented in Section 3 and Section 4. We will see that the design of the LinGO grammar appears as not using/exploiting some of the interesting speed-up capabilities that can be generally inferred (at compilation or pre-processing time) for typed-unification grammars. Interestingly, most of the optimisations presented here can be migrated to interpreter-like systems, for instance PET [3], LKB [12] and TDL/PAGE [16].

2 Improving the (Compiled) Quick Check

This section starts with a (rather) formal definition for the quick check [15] [17]. Then it briefly presents the compiled version of QC [7] [10]. Its newly added refinements constitute the main objective of this section.

Let φ and ψ be two FSs defined over a lower semi-lattice of sorts Σ , and $\{\pi_1, \pi_2, \dots, \pi_n\}$ a set of feature paths. The QC test operates as it follows: if $\exists i \in \{1, 2, \dots, n\}$ such that $sort(\varphi.\pi_i) \wedge sort(\psi.\pi_i) = \perp$, then φ and ψ do not unify. (The notation $\varphi.\pi_i$ designates the substructure of the FS φ identified by the path π_i , while $sort(\varphi.\pi_i)$ identifies the sort associated to the root node of $\varphi.\pi_i$. The \wedge operator designates the greatest lower bound (glb) of two sorts in the semi-lattice Σ . The symbol \perp designates the inconsistent sort. The expression $sort(\varphi.\pi_i) \wedge sort(\psi.\pi_i) = \perp$ means that the two sorts are incompatible, i.e. their

² This optimisation concerns the QC-path values which are not known at compilation time. Those paths are empirically proven to form the majority among all QC-paths for the LinGO grammar (and therefore are very important) for the QC filter. Maximal common prefixes of those paths are computed at compilation time, thus avoiding the repeated traversal of a feature path.

³ Due to LinGO design, the LIGHT system operates only with binary and unary rules. The LHS substructure of a passive item will be built only if QC succeeds on the key (head) argument of a rule selected by the syntactic filter or (if no such QC test succeeds) if QC succeeds on the non-key argument for a previously created active item.

intersection is inconsistent in Σ . For a thorough view on such issues, the interested reader may consult [4] [2] [13].)

From the implementation point of view, since the FS φ usually takes part in several such QC tests, the sort values $sort(\varphi.\pi_i)$ for $i = \overline{1, n}$ are computed only once, normally immediately after the creation of φ . Therefore an application of the QC filter (or, simply, a QC test) is seen as pair-wise sort consistency check for two sort vectors QC_φ and QC_ψ , the so-called QC-vectors associated to φ and respectively ψ . (Formally: $QC_\varphi = [sort(\varphi.\pi_i)]_{i=\overline{1, n}}$.)

The above simple view on computing QC_φ is challenged in the [6] work, which shows that in a compilation approach to bottom-up deduction-based parsing with FSs, if the (very effective) Specialised Rule Compilation optimisation is used [5], then the computation of the QC-vectors has to be re(de)defined, simply because the (representation of the) FS φ does not exist on the heap.⁴

In this compilation setup, some components of QC_φ are known at compile time [$QC_\varphi]_i = sort(\varphi.\pi_i) = s_i$, other components are easily computable at the run time [$QC_\varphi]_i = sort(X_j)$, with j known at compile time ($X_j = \varphi.\pi_i$), while the remaining components are (more expensively) computable at run time [$QC_\varphi]_i = sort(X_j.\pi_i'')$, where $\varphi.\pi_i = X_j.\pi_i''$, with j and π_i'' known at compilation time.⁵ As a consequence, in the LIGHT compiler system, for a given FS, the QC-vector exists in two forms: a pre-computed one (gathering the informations that can be determined at compilation time), and a fully computed one (produced at run time, starting from the pre-computed form). Thorough details of the computation of QC_φ in the compilation approach, together with a comprehensive exemplification are given in [10].

Concerning the pre-computed form of a QC-vector, one will see that the last case in the above sketched definition/computation can be optimised through a second compilation phase. Indeed, it is often possible that two paths π_k and π_i have a common prefix. Compilation can easily identify the maximal common prefix π in such a case. As a consequence, the computation of $\varphi.\pi_i$ and $\varphi.\pi_k$ can be optimised, by computing only once the path value for π in φ . In LIGHT, the implementation of this optimisation is currently limited to those cases in which π_k is a prefix of π_i .

A further step we took in order to improve the compilation of QC-vector was to personalise, i.e. to make rule-sensitive, the application of QC tests. As presented above, the QC filter is assumed to operate on QC-paths in decreasing order of the probability to detect unification failure. That order is usually computed on the same corpus (or test suite) on which the QC-paths have been identified. It is obvious that for each rule argument, the order in which the QC-paths must be explored in order to get first the most probable failures (i.e. sort inconsistencies) may significantly differ from the globally (corpus-relative) computed order.

For the LinGO grammar, we used the CSLI test suite to compute the sets of (indices of) ordered QC-paths for each rule argument, and the respective failure frequencies. It is interesting to note that

- previously, with non-personalised QC the optimal number of QC-paths actually used in the QC filter in the LIGHT system was 43 (out of a total of 132 identified QC-paths on the CSLI test suite), most of the personalised sets of failure paths comprise at most 30 paths,
- only few such sets (4) have more than this (namely more than 90 paths!), and some sets are empty!⁶

⁴ Instead, φ is represented in the program by a function, which is the compiled form of φ .

⁵ More precisely, we take $\pi_i = \pi_i'\pi_i''$, and $\varphi.\pi_i = X_j$, such that π_i'' is minimal with these properties.

⁶ We found that out of a total of 91 rule arguments in LinGO, for 13 of them QC doesn't detect any sort inconsistency on the CSLI test suite.

$$\begin{aligned}
& QC_r = \emptyset \text{ (the empty set)} \\
& \text{for each rule } i, \text{ and for each argument } j \text{ of the rule } i \\
& \quad \text{if } \text{syn_filter}(r, i, j) = \text{TRUE} \\
& \quad \quad QC_r = QC_r \cup QC_{ij}
\end{aligned}$$

Fig. 1. The procedure computing QC_r , the set of (only) those QC-paths which are eventually used in QC tests involving FSs produced by the rule r .

Therefore, the order in which the QC-test is applied was subsequently naturally altered: If Q_{ij} is the set of personalised QC-paths for the rule i and the argument j , the sort consistency check is applied in decreasing order of $[\mu(QC_{ij})]$, the array of frequencies of failures produced by the paths in QC_{ij} .

Furthermore, using the QC_{ij} ordered sets of QC-path indices associated to rule arguments, for each rule r we produced the set QC_r consisting of (only those) QC-paths which are eventually used in QC tests involving FSs produced by the rule r as shown in Figure 1. There, *syn_filter* denotes the syntactic rule filter, obtained at grammar’s compilation time, while *syn_filter*(r, i, j) = *TRUE* denotes the fact that the (FS associated to the) rule r is compatible with the argument j of the rule i . It is obvious that for FSs produced by the (full) application of the rule r , the run-time computation of the associated QC-vectors can be limited to the paths whose indices are found in the QC_r set.

One more optimisation related to the QC test: let ψ be a lexical FS, and φ , the FS representing the key (i.e. first treated) argument of rule i ; if ψ is proven at the compilation time (i.e., when computing the lexical filter) to be compatible with φ , then we know for sure that the corresponding QC test is superfluous and therefore can be eliminated.

The interested reader will see that all these optimisations, with the exception of the first one, can be easily adapted to non-compilation-based systems by adequately pre-processing the grammar. (The first optimisation however can be adopted by such systems in the same, less general form that it is currently implemented in LIGHT.)

Figure 2 shows the combined effect that the above presented optimisations incorporated into the LIGHT system had on the GR-reduced form of LinGO grammar, when using the CSLI test suite.⁷ First, we visualised the average parsing times we obtained for LinGO, in two versions: with QC-paths selected as usually (as most frequent failure paths), and then using GR-paths as QC-paths. Then, we showed the effect of all the above presented optimisations. All these (three kinds of) parsing times have been recorded for an increasing number of GR-paths used QC-paths. The minimal average parsing time, 15.163 msec/sentence, — obtained by LIGHT on the the CSLI test suite using GR-paths as QC-paths and including all the optimisations presented above —, is 8.71% better than 16.610, the best time recorded for the previous (one-phase, non-personalised) QC compilation approach.

We have to note that using GR-paths as QC-paths to perform the QC test on the GR-reduced form of a grammar may have the disadvantage that some frequent unification failures may appear outside (i.e. on extensions of) GR-paths, and therefore they will not be checked during the QC test. Conversely, performing QC on the proper QC-paths on the GR-reduced form of the grammar may suffer from the drawback that some (usually few) of the QC-paths are not GR-paths, and therefore the corresponding sort values in the QC-vectors are not

⁷ We consider that the effect that QC has on the average parsing time is more appropriately measured on the GR-reduced form of the used grammar, since non-GR paths by definition do not contribute to unification failures (on the training test suite).

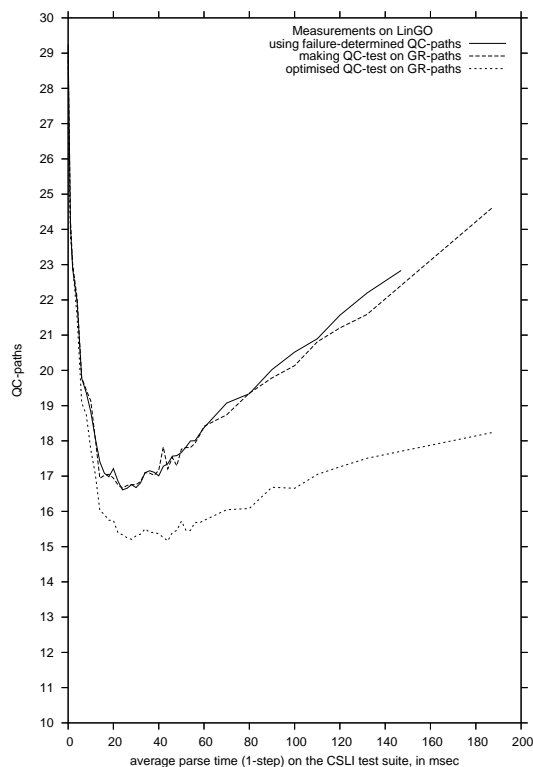


Fig. 2. Comparing the effect that the one-phase compiled QC — first on corpus-relative QC-paths and then on GR-paths — and respectively the two-phase compiled, personalised QC had on the average parsing time registered for the GR-reduced form of LinGO on the CSLI test suite, using the LIGHT system.

computable at compilation time.

Regarding the plots in Figure 2 we have to remark that:

1. The decline in parsing efficiency which appears when using GR-paths as QC-paths instead of usual QC-paths (obtained by taking the most frequent failure paths when parsing the same test suite as the one used to get the GR form of the grammar) is practically not significant: going up from 16.610 to 16.662 means only a 0.31% slowing down factor.
2. The minimal average parsing time (15.163 msec/sentence) was obtained for a greater number of QC-paths in the two-phase QC compilation approach (44 vs. 24 used before), mainly because the newly embedded optimisations enable a faster exploration of QC-paths.
3. There is a rapid decrease of the average parsing time w.r.t. the number of QC-paths involved for each of the two (main) cases. For instance, for the first 6 QC-paths, the average parsing time is reduced by 29.39% in our last QC version! (The total reduction factor due to the QC filter is 43.90%.)
4. Beyond the optimal number of used QC-paths, the speed performance decreases slowly, but the deterioration is smaller in the two-phase QC compilation approach, again because less time is spent on building the QC-vectors.

The next section will show a natural way to extend the definition of the QC pre-

unification filter given in the beginning of the present section, and then we will examine the sensitivity of the LinGO grammar to this new extension.

3 Extending the Quick Check Definition/Area

If two FSs do not unify, this is due to at least one sort inconsistency in the logical formula representing the conjunction of the two FSs. Saying it in another way: unification failure for (typed) FSs is always the effect of incompatibility of certain pairs of sorts. The ‘classical’ form of the QC pre-unification filter (as presented in the precedent section of this paper) only searches for sort incompatibility on mutual paths, i.e., $sort(\varphi.\pi) \wedge sort(\psi.\pi) = \perp$, where \perp is the inconsistent sort.⁸ But failure in typed FS unification may stem from two other sources: run-time type checking and coreferenciation.

3.1 Coreferenced Based Quick Check

We will show that the early identification of unification failure due to (incompatible) coreferences is not difficult.

Suppose that the FS φ is going to be unified with ψ , and that φ contains the coreference $\varphi.\pi \doteq \varphi.\pi'$.⁹ In this setup, if for a certain path η it happens that $sort(\varphi.(\pi\eta)) \wedge sort(\psi.(\pi\eta)) \wedge sort(\psi.(\pi'\eta)) = \perp$, then certainly φ and ψ are not unifiable.¹⁰ A “mild” form of the newly introduced ‘coreference-based’ QC involves only the FS ψ : if $\pi\eta \doteq \pi'\eta$ in φ and $sort(\psi.(\pi\eta)) \wedge sort(\psi.(\pi'\eta)) = \perp$, then φ and ψ are not unifiable.¹¹ In the sequel, when referring to coreference-based QC, we will mean its mild form, if not otherwise stated.¹²

There is no a priori reason why, on certain typed grammars, coreference-based sort inconsistency would not be more effective in ruling out FS unification than sort inconsistency on mutual paths. Moreover, the integration of the two forms of QC is not complicated. However, up to our knowledge no system parsing LinGO-like grammars included the above newly presented form of (coreference-based) pre-unification QC test. We will later provide an explanation in the remaining part of this section, where we will evaluate the effectiveness of the (mild form of) coreference-based QC pre-unification filter on LinGO.

⁸ For LinGO, when running the CSLI test suite, the average number of such sort inconsistencies per each attempted unification which doesn’t pass the QC-filter test is 1.99 when using the first 32 QC-paths (2.12 if all GR-paths effective as QC-paths were used).

⁹ The \doteq symbol denotes the identity of the two (coreferred) substructures, $\varphi.\pi$ and $\varphi.\pi'$.

¹⁰ Note that in this ‘coreference-based’ QC, the sort intersection $sort(\varphi.(\pi\eta)) \wedge sort(\psi.(\pi\eta))$ denotes the result of ‘classical’ QC on the $\pi\eta$ path. Unfortunately, the current implementation of QC in the LIGHT system does not store the values computed by the classical QC test. If we would do so, then (when adding coreference-based QC) the QC-computed sort value for $\pi\eta$ would be further intersected with $sort(\psi.(\pi'\eta))$.

¹¹ Note that the ‘classical’ QC involves at ‘elementary’ level (i.e. for each pair of sorts) two distinct FSs and only one path. The mild form of ‘coreference-based’ QC defined above involves a single FS and (at ‘elementary’ level) two distinct paths.

¹² As stated above, coreference-based QC concerns coreferences in one rule argument. We have to notice that apart that case, in a rule FS coreferences may involve paths in different arguments. In case of binary rules (as in LinGO) that means coreferencing substructures between the key argument, and the non-key/complement argument: $\varphi.\pi \doteq \psi.\pi'$. (The key argument is the first to be treated among all arguments of a rule.) On the GR-reduced form of the LinGO grammar, we found 61 such cross-argument coreferences. One has to note that the exploration of this second coreference case is implicitly covered by the classical QC (i.e. both in QC-vector computing and the QC-test itself).

<i>rule</i>	<i>classical QC</i>	<i>coref.-based QC</i>
9	41,533	24,399
11	49,180	2,418
28	59,390	13,515
30	39,586	10,069

Fig. 3. Comparison between the number of failures detected by classical QC vs. coreference-based QC on key rule arguments. (On arguments and rules not shown in the table, coreference-based QC scores 0.)

On the GR-reduced form LinGO we identified 12 pairs of non-cross argument coreferences inside rule arguments (at LinGO’s source level). Interestingly enough, all these coreferences occur inside key arguments, belonging to only 8 (out of the total of 61) rules in LinGO.

To perform coreference-based QC, we computed the closure of this set Λ of coreference paths. The closure of Λ will be denoted $\bar{\Lambda}$. If the pair π_1 and π_2 is in Λ , then together with it will be included in $\bar{\Lambda}$ all pairs of QC-paths such that $\pi_1\eta$ and $\pi_2\eta$, where η is a feature path (a common suffix to the two newly selected paths).¹³ For the GR-reduced form of LinGO, the closure of Λ defined as above amounted to 38 pairs. It is on these pairs of paths that we performed the coreference-based QC test.

Using all these coreference paths pairs, 70,581 unification failures (out of a total of 2,912,623 attempted unifications) were detected on the CSLI test suite. Only 364 of these failures were not detectable through classical QC. When measuring the “sensitivity” of coreferenced-based QC to individual rule arguments, we found that out of a total of 91 rule arguments in LinGO only for 4 rule arguments the coreference-based QC detects inconsistencies, and the number of these inconsistencies is far much lower than those detected by the classical QC on the same arguments. The comparative numbers are shown in the Table 3. None of the pairs of coreference paths exhibited a higher failure detection rate than the first ranked 32 QC-paths. If one would work with 42 QC-paths, then only 4 of the pairs of coreference paths would score failure detection frequencies that would qualify them to be taken into consideration for the (extended form of) QC-test.

We think that the explanation for these modest numbers stems from the limited preference/need of LinGO designers for using non-cross argument coreferences.

As a conclusion, it is clear that for LinGO, running the coreference-based QC test is virtually of no use. For other grammars (or other applications involving FS unification), one may come to a different conclusion, if the use of non-cross argument coreferences balances (or outnumbers) that of cross-argument coreferences.

3.2 Type checking Based Quick Check

Failure of run-time type checking — the third potential source of inconsistency when unifying two typed FSs — is in general not so easily/efficiently detectable at pre-unification time, because this check requires calling a type consistency check routine which is much more expensive than the simple sort consistency operation.

¹³ In fact we proceed in a more elaborated manner: The elements of Λ are tuples (r, i, π_1, π_2) , where r is a rule (index) and i is the index of one of its arguments. The closure of Λ is computed as follows: if $(r, i, \pi_1, \pi_2) \in \Lambda$, then together with it in $\bar{\Lambda}$ will be included all tuples $(r, i, \pi_1\eta, \pi_2\eta)$ where $\pi_1\eta, \pi_2\eta$ are within the set of (pre-selected) QC-paths.

While exploring the possibility to filter unification failures due to type-checking, the measurements we did using LinGO (the GR-reduced form) on the CSLI test suite resulted in the following facts:

1. Only 137 types out of all 5235 (non-rule and non-lexical) types in LinGO were involved in (either successful or failed) type-checks.¹⁴ Of these types, only 29 types were leading to type checking failure.¹⁵
2. Without using QC, 449,779 unification failures were due to type-checking on abstract instructions, namely on `intersects_sort`; type-checking on `test_feature` acts in fact as type unfolding. When the first 32 QC-paths (from the GR-set of paths) were used (as standard), that number of failures went down to 92,447. And when using all 132 QC-paths (which have been detected on the non GR-reduced form of LinGO), it remained close to the preceding figure: 86,841.
3. For QC on 32 paths, we counted that failed type-checking at `intersect_sort` occurs only on 14 GR-paths. Of these paths, only 9 produced more than 1000 failures, only 4 produced more than 10,000 failures and finally, for only one GR-path the number of failed type-checks exceeded 20,000.

The numbers given at the above second point suggest that when trying to extend the ‘classical’ form of QC towards finding all/most of failures, a considerably high number of type inconsistencies will remain in the FSs produced during parsing, even when we use all (GR-paths as) QC-paths. Most of these inconsistencies are due to failed type-checking. And as shown, neither the classical QC nor its extension to (non-cross argument) coreference-based QC is able to detect these inconsistencies.

The first and third points from above say that in parsing the CSLI test suite with LinGO, the failures due to type checking tend to agglomerate on certain paths. But due to the fact that type-checking is usually time-costly, our *conclusion*, like in the case of non-cross argument coreference-based QC, is that extending the classical QC by doing a certain amount of type-checking at pre-unification time is not likely to improve significantly the unification (and parsing) performances on LinGO.

If for another type-unification grammar

- the number of failed type-checks is much higher compared to the number of sort inconsistencies (on mutual or coreferenced paths), and
- the types involved in those failed type-checks are not very elaborated,

one can extend (or replace) the classical QC test with a *type-check QC filter procedure*. Basically, after identifying the set of paths (and types) which most probably cause failure during type-checking, that procedure works as shown in Figure 4.¹⁶

What we put in evidence in this section is a comparative view between different *kinds* of inconsistency in typed FS unification and the way they overlap in the (extended) design of the QC pre-unification filter. As in the precedent section, the non-classical (i.e., coreference-based and type-checking) alternatives to the QC filter here presented can be adapted to systems embedding non-compiled unification. The next section will present the combination of the QC filter with other optimisations which have been incorporated into the LIGHT system, and will analyse their effect on LinGO.

¹⁴ For 32 QC-paths: 122 types, for 132 QC-paths: also 122.

¹⁵ For 32 QC-paths and 132 QC-paths: 24 and 22 respectively.

¹⁶ Note that the check in Figure 4 is done on ψ since φ is assumed to be in compiled form, and therefore we can know in advance the outcome of the alternative test

$\Psi(s).\pi \wedge \text{GR}_\varphi[i] = \perp$ or type-checking $\varphi.\pi_i$ with $\Psi(s)$ fails.

```

s = GR $\varphi$ [i]  $\wedge$  GR $\psi$ [i];
if s  $\neq$  GR $\varphi$ [i] and s  $\neq$  GR $\psi$ [i] and
   $\varphi.\pi_j$  or  $\psi.\pi_j$  is defined for a certain non-empty path  $\pi_j = \pi_i\pi$ , an extension of  $\pi_i$ ,
  such that  $\pi_j$  is a QC-path,
then
  if  $\Psi(s).\pi \wedge$  GR $\psi$ [i] =  $\perp$ , where  $\Psi(s)$  is the type corresponding to s,
  or type-checking  $\psi.\pi_i$  with  $\Psi(s)$  fails
  then  $\varphi$  and  $\psi$  do not unify.

```

Fig. 4. The core of a type-checking specialised compiled QC sub-procedure.

4 On QC Effectiveness on LinGO: The Limits

Two optimisations of the LIGHT system to which the LinGO grammar was proven as having no (or only very reduced) sensitivity are:

1. making the QC test precede the (delayed) creation of the LHS substructures in successful rule applications.
2. using look-up tables to retrieve the values of the X variables used by certain *READ* abstract instructions (in the compiled for of one rule argument) directly from the candidate FS, without having (in most cases for the GR-reduced form of rules) to explore the feature tables.

The first optimisation is possible in LIGHT due to the specialised compilation of rules [5]. While implementing this optimisation proved useful in getting a sensible reduction of memory consumption, it provided almost no gain in the parsing speed. The reason for this lack of speed-up effect is explained by the following facts:

- i.* When we allow the QC test to be applied immediately after the successful unification of arguments (with candidate FSs), we must save the actual values of coreferences so to later retrieve them for the (delayed) creation of the LHS partition of the rule FS.
- ii.* The time cost for that save-and-restore work proves to be comparable to the (fast) application of the *WRITE* abstract instructions in the (compiled) rule LHS.

One could expect that for a grammar which has significantly larger LHS rule substructures and fewer coreferences (among LHS and the RHS) than LinGO, this optimisation will have a sensible effect.

We would like to add that in our view this optimisation is obtained for free in those (interpretation-based) systems which employ hyper-active parsing [20][19][21]. This is because they delay the copying of rule feature substructures until they are really needed in other, newly-created structures, and therefore the application of the QC filter precedes the completion of the new rule FS instance.

The second optimisation seemed to be promising when we started to work with GR-reduced rules, because it can exploit a considerable amount of knowledge detectable at compilation time, under the form of a fine-grained correlation between FSs representing rules. More exactly, by computing the correspondence between the X variables in the compiled form of (rule) FSs [1] and the set of GR-paths, one can get before unification itself the values of many variables to be associated to substructures in the candidate FS. But preliminary

tests done with LIGHT on LinGO revealed that this optimisation would amount to a speed up of parsing on the GR-reduced form of LinGO no higher than 10%.¹⁷

Our opinion is that this disappointing figure is due to the fact that when parsing the CSLI test suite with the LinGO grammar, only few parse items (i.e., rule-produced FSs) are intensively used during further parsing.

Although it has no link to the QC filter, we mention here another optimisation that we attempted in LIGHT for LinGO: the elimination of copying feature-value pairs in the save-and-restore process that accompanies parsing. To eliminate copying we need to keep pointers to relate non-destructible ‘pages’/tables of feature-value pairs. Measurements on LinGO prove that the memory block copying (*memcpy*) operation in C is so efficient (at least on not very large frames, as it is the case of LinGO), that there is no gain in eliminating copying. (There is however a gain in the fact that non-destructive frames make data persistent, and it can be used for tracing the unification and parsing process, which is very useful.) Again, like most developments presented in the present paper, this optimisation too is not restricted to compilation-based unification.

5 Conclusions

It was widely known from the literature that, in parsing with LinGO-like grammars, FS unification is a much time consuming operation and a high percentage of tried unifications are doomed to failure.

From the procedural/computational perspective, finding in an efficient way the inconsistencies among two FSs depends on the order in which FSs are traversed. In the LIGHT system, both in the on-line unification procedure and in the compiled form of rules [1], the FSs to be unified are explored in depth-first manner. The same is true about the Tomabechi efficient unification algorithm [22].

The effectiveness of QC comes from empirically/statistically identifying a better way of traversing the FSs submitted to unification. While in previous works [15] [17] [7] [10] it was determined only as holistic property of the grammar (and the training test suite used for measuring the distribution of failure on feature paths), the work presented in Section 2 made this QC-determined traversal rule-and-argument sensitive. Together with other new improvements, this optimisation reduced the average parsing time on LinGO.

Concerning the issue of multiple (and overlapping) sources of inconsistency, we showed in Section 3 that it is merely a statistical fact (which has no linguistic nor unification theoretical bias!) that in parsing with LinGO if there is any inconsistency between two FSs, no matter which kind is it, then it is very probable that it will be (accompanied by) a sort inconsistency

¹⁷ For those preliminary tests

- we by-passed the compilation of the correspondence between X variables and GR-path indices in the compiled form of rules, then we computed the GR-path values at run time in a non-compiled manner. In the end, after we measured the parsing time, we subtracted the time consumed by this computation of GR-path values;
- the *READ* abstract instructions `test_feature` and `unify_feature` were extended so to first test the definite-ness of the X variables which eventually store the value of the f feature (given as an argument to the abstract instruction call). If X is defined (actually as an already computed/stored GR-path value), obviously there is no need to search its value through feature tables.

This 10% factor will decrease when counting the time needed at the run-time to store the GR-path values.

on mutual paths.¹⁸ Moreover, finding that failure is the cheapest way — compared to the two other possibilities: sort inconsistency on coreferenced non-mutual paths, and type checking —, and this explains why in all LinGO-running systems the inconsistencies are filtered out by sort consistency QC on mutual paths. Several improvements in the design of the LIGHT system concerning the combination of the QC filter with other optimisations — the syntactic rule filter and the memoization of X variables in the compiled rule FSs based on the GR-learned form of the grammar — were presented in Section 4.

On one hand the facts presented in this paper explain the success of the classical QC filter.¹⁹ On the other hand these facts (especially Section 3 and Section 4) should make us aware that if grammars (or FS-based applications) exhibit empirical properties different than those exhibited by LinGO-like HPSG grammars, we have to be prepared to plug in different filtering strategies, or even to switch to other, better suited systems. Indeed, if you use a grammar with a quasi-uniform distribution of unification failures on feature paths inside rule arguments, then you have to choose a compiler. In such a case a compiler is significantly (i.e., almost twice) faster than an interpreter as we have practically proven by measurements done when the LIGHT and PET [3] systems were still conceptually close in their main architecture [9].

Most of the optimisations presented in this paper can be easily adapted to non-compilation based parsers for LinGO-like grammars.

Acknowledgements

The LIGHT system was designed and implemented while the author worked as a researcher at the Geerman Research Institute for Artificial Intelligence (DFKI), Saarbrücken, Germany. The GR procedure was implemented and documented when the author was employed at the University of York in the framework of a ROPA (EPSRC) grant. Some roots of the present research results were seeded during a stay at the University of Wales, Aberystwyth, UK, thanks to a BBSRC grant. The measurements here reported were done using a Pentium III PC running Red Hat 7.1 Linux at 933 MHz.

References

1. H. Ait-Kaci and R. Di Cosmo. Compiling order-sorted feature term unification. Technical report, Digital Paris Research Laboratory, 1993.
2. H. Ait-Kaci, A. Podelski, and S.C. Goldstein. Order-sorted feature theory unification. In Dalle Miller, editor, *Proceedings of the International Symposium on Logic Programming*, pages 506–524, Vancouver, 1993. MIT Press.
3. U. Callmeier. 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, 2000.
4. B. Carpenter. *The Logic of Typed Feature Structures – with applications to unification grammars, logic programs and constraint resolution*. Cambridge University Press, 1992.

¹⁸ For instance, it is worth to see that if no QC is performed at all, then out of the 2,912,623 unifications done during the parsing of the CSLI test suite, 1,122,843 (representing a surprisingly high percentage, 38.55%) are ruled out at the first failing type-checking.

¹⁹ In our view, it is due to QC and the very efficient Tomabechei unification algorithm with non-destructive FS sharing that the non-compilation based parsing systems became competitive with the few implemented compilers, like LiLFeS [18] and LIGHT.

5. L. Ciortuz. On 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, 2001.
6. L. Ciortuz. 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, 2001.
7. L. Ciortuz. Learning attribute values in typed-unification grammars: On generalised rule reduction. In D. Roth and A. van den Bosch, editors, *Proceedings of the 6th Conference on Natural Language Learning (CoNLL-2002)*, pages 70–76, Taipei, Taiwan, 2002. Morgan Kaufmann Publishers and ACL.
8. L. Ciortuz. LIGHT — a constraint language and compiler system for typed-unification grammars. In *KI-2002: Advances in Artificial Intelligence*, volume 2479, pages 3–17, Berlin, Germany, 2002. Springer-Verlag. Proceedings of the 25th Annual German Conference on AI, Aachen, Germany.
9. L. Ciortuz. LIGHT AM — another abstract machine for feature structure unification. In S. Oepen, D. Flickinger, J. Tsujii, and H. Uszkoreit, editors, *Efficiency in Unification-based Processing*, pages 167–194. CSLI Publications, The Center for the Study of Language and Information, Stanford University, 2002.
10. L. Ciortuz. On two classes of feature paths in large-scale unification grammars. In H. Bunt, J. Carroll, and G. Satta, editors, *New Developments in Parsing Technologies*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2004.
11. L. Ciortuz. *Parsing with Unification-Based Grammars — The LIGHT Compiler*. EditDan Press, Iasi, Romania, 2004.
12. A. Copestake. *The (new) LKB system*. CSLI, Stanford University, 1999.
13. A. Copestake. Definitions of typed feature structures. *Journal of Natural Language Engineering*, 6 (1) (Special Issue on Efficient Processing with HPSG), 2000.
14. Daniel P. Flickinger, Ann Copestake, and Ivan A. Sag. HPSG analysis of English. In Wolfgang Wahlster, editor, *VerbMobil: Foundations of Speech-to-Speech Translation*, Artificial Intelligence, pages 254–263. Springer-Verlag, 2000.
15. B. Kiefer, H-U. Krieger, J. Carroll, and R. Malouf. 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, 1999.
16. H.-U. Krieger and U. Schäfer. TDL — A Type Description Language for HPSG. Research Report RR-94-37, German Research Center for Artificial Intelligence (DFKI), 1994.
17. R. Malouf, J. Carroll, and A. Copestake. Efficient feature structure operations without compilation. *Journal of Natural Language Engineering*, 6 (1) (Special Issue on Efficient Processing with HPSG):29–46, 2000.
18. Y. Miyao, T. Makino, K. Torisawa, and J. Tsujii. 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, 2000.
19. S. Oepen and J. Carroll. Ambiguity packing in HPSG — practical results. In *Proceedings of the 1st Conference of the North American Chapter of the ACL*, pages 162–169, Seattle, WA, 2000.
20. S. Oepen and J. Carroll. Performance profiling for parser engineering. *Journal of Natural Language Engineering*, 6 (1) (Special Issue on Efficient Processing with HPSG: Methods, Systems, Evaluation):81–97, 2000.
21. S. Oepen and J. Carroll. Efficient parsing for unification-based grammars. In S. Oepen, D. Flickinger, J. Tsujii, and H. Uszkoreit, editors, *Efficiency in Unification-based Processing*, pages 195–225. CSLI Publications, The Center for the Study of Language and Information, Stanford University, 2002.
22. H. Tomabechi. Quasi-destructive graph unification with structure-sharing. In *Proceedings of COLING-92*, pages 440–446, Nantes, France, 1992.