ON COMPILATION OF THE QUICK-CHECK FILTER FOR FEATURE STRUCTURE UNIFICATION

Liviu Ciortuz
Computer Science Department
University of York
Heslington, York, YO10 5DD, UK
ciortuz@cs.york.ac.uk

Abstract

The quick-check (QC) technique introduced by [11] is a highly effective optimisation technique for filtering out the (eventually unsuccessful) unification of feature structures. This paper presents the compilation of the QC filter as it was designed and implemented in the Light compiler system [4]. (Light stands for LIGHT — Logic, Inheritance, Grammars, Heads, and Types.[1])

Up to our knowledge, it is the first attempt to incorporate this pre-unification speed-up technique in a compiler system dealing with large-scale typed-unification grammars, and what makes this work interesting is that it proposes a compiled form of the QC test, which is significantly more elaborated than its original form.

This elaboration is motivated not only by the compilation's overall aim to get an increased speed up for parsing, the fact is that the original, simple, interpreted form of the QC filter cannot directly accommodate with other optimisation techniques for compiled parsing with typed-unification grammars, and this very fact made compilation of Quick-Check fully justified.[2]

0 Introduction

Significant progress has been achieved during the last couple of years in the area of efficient natural language processing with large-scale feature-based grammars. A recent work [15] presented some of the most advanced results concerning parsing with wide-coverage HPSG grammars [16], notably the LinGO grammar [8] for English developed at CSLI, University of Stanford. The QC pre-unification filter [11], one of the most remarkable speed-up techniques in this area is, up to our knowledge, used now in all but one of the systems able to cope with LinGO. (We named LKB [7], TDL [10], PET and Light; the exception — the LiLFeS system [12] — instead of using QC, employs a CFG filter which leads to practical results comparable to (the best of) the other systems but, up to our knowledge, it is much more memory consuming. For PET, the reportedly fastest system running LinGO, the author reports a factor of speed up of about 63% after the introduction of the QC test [2].)

The present paper deals with getting a compiled form of the QC filter, suitable for an elegant and efficient integration with compilers for unification-based grammars.

Making it simple, the idea behind the pre-unification QC test is the following:

Having got the knowledge about the most probable *failure paths* $\pi_1, \pi_2, ..., \pi_n$ in the application of parsing rules,[3] before doing the unification of a certain feature structure ψ_1 (representing a phrase)

[1] The analogy with the name of LIFE — Logic, Inheritance, Functions and Equalities — a well-known constraint logic language based on the OSF constraint system [1] is evident.

[2] The conception and implementation side of the work here reported was done while the author was employed at the LT Lab of the German Research Center for Artificial Intelligence (DFKI) in Saarbrücken, Germany.

[3] These most probable failure paths are identified by running the system (without QC) on a large corpus.

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., $root(\psi_1.\pi_i) \wedge root(\psi_2.\pi_i) \neq \bot$, where 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 $\bot$ is the bottom/inconsistent sort in the grammar's sort hierarchy. This 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 for such a path this compatibility test is not passed, then it follows immediately that the feature structures ψ_1 and ψ_2 don't 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. Here follows a first explanation:

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, doing the QC test costs (it is finally a waste in case of actual successful unification). Measurements done first on the LKB, TDL, PET systems revealed that in case of interpreted-like parsing with LinGO-like grammar, it is worth 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 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 side, and the speed of the unification procedure on the other side is dramatically narrowed. A compiled form of the QC filter as will be presented here is proven able to "enlarge" again this trade area for speeding up the parsing/deduction.

Basically, this paper shows how QC-vectors $\{root(\psi.\pi_1), root(\psi.\pi_2), ..., root(\psi.\pi_n)\}$ can be computed in two stages, the first one done once for all at the compilation time (we call it QC pre-computation), completed at the run-time by the second one according to specific circumstances. The basis for this "two-step" computation of QC-vectors resides in the facts that *i.* the order in which one rule's arguments will be processed is known at the 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 the compilation time ($\psi \sqsubseteq \Psi$).

Remark: Formally, for any QC feature-path π, the $\mathrm{QC}_\pi(\psi) = root(\psi.\pi)$ value will be computed by applying at the run-time a function ρ to a certain argument $preComp(\Psi, \pi)$, computed at the compilation time:

$$\mathrm{QC}_\pi(\psi) = \rho(preComp(\Psi, \pi)).$$

The three sections of the present paper deal respectively with 1. explaining the problems we encountered when we tried 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 [5]; 2. getting the compiled (incomplete, "pre-computed") form of the QC-vectors followed by a simple example; 3. computing their run-time, completed form, and suggestions for improvements. A final, *evaluation* paragraph provides figures on the compiled QC efficiency, namely the measurements done for LinGO by running the Light system on the CSLI test suite both with and without the QC filtering.

1 Can the (interpreted) QC test be accommodated into the (compiled) parsing in Light?

Let us first analyse the way the QC was conceived in the interpreting setup (of the LKB and PAGE/TDL 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 (FS) ψ. This QC-vector is defined as $\{root(\psi.\pi_1), root(\psi.\pi_2), ..., root(\psi.\pi_n)\}$. 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 m "active" QC-vectors; in the case of a binary rule φ, we will have first a "key" QC-vector $\{root(\varphi'.\pi_1), root(\varphi'.\pi_2), ..., root(\varphi'.\pi_n)\}$, where $\varphi' = \varphi.\text{KEY-ARG}$, 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 presumptive 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 $\bot$), 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 $\varphi'' = \phi.\text{NON-KEY-ARG}$: $\{root(\varphi''.\pi_1), root(\varphi''.\pi_2), ..., root(\varphi''.\pi_n)\}$, where ϕ is what φ has become after $\varphi' = \varphi.\text{KEY-ARG}$ has been unified with ψ.

As already mentioned in the introductory section, the main *problem* that we've got when we tried to integrate the QC pre-unification test with the Light compiler was its accommodation with the previously included main optimisation: the specialised compilation of rules. While the "key" QC-vector φ' can be thoroughly computed at the grammar compilation/loading time, computing the QC-vector for $\varphi'' = \phi.\text{NON-KEY-ARG}$ is not immediately possible simply because the $\phi.\text{NON-KEY-ARG}$ structure does not effectively exists (on the heap). Let us detail this issue:

In Light, 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 one 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 sharing (environment-based) 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 — which is simple and elegant due the use of open records/FSs as in the OSF constraint theory [1], in contrast with closed records used in the appropriateness-based approach [3] underlying other LinGO-parsing systems — provided us a factor of speeding up of 2.75 on the test suite provided by the CSLI, University of Stanford.

Note that even in the hyper-active head-corner parsing approach proposed by Oepen and Carroll [14], 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 were 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 $\varphi''.\pi$ exist, then $root(\varphi.\text{NON-KEY-ARG}.\pi) \preceq root(\varphi''.\pi)$ in the sort hierarchy —, we were entitled to use it for QC. However, in this way the speed up effect of the QC test with Light when parsing the CSLI test suite with the LinGO grammar was not significant. (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 $\varphi' = \varphi$.KEY-ARG was unified with ψ, the FS corresponding to a passive item.)

The second, actual solution we proposed was to compile (the computation of) the QC-vectors. It will be presented in the next sections. Basically, the idea is that instead of computing for instance for a binary rule three QC-vectors like in the interpreted approach — one "key" QC-vector at the loading/pre-processing time, a "complete", and finally a "passive" one at the run-time[4] —, in the compilation approach we will compute five QC-vectors, among which three are computed at the compilation time and two at the run-time, the last two building upon the pre-computed ones.

QC test	"key" QC-vector	"complete" QC-vector	"passive" QC-vector
compilation-time	$preComp(\varphi') =$	$preComp(\varphi'')$	$preComp(\varphi)$
run-time	$= \text{QC}(\varphi')$	$\text{QC}(\phi)$	$\text{QC}(\phi')$

Figure 1: The QC-vectors computed for a rule φ in the compilation approach.

In the notation used in Figure 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.[5]

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 full representation) of rule FSs from the heap.

Note that — assuming like in the head/key-corner parsing [9] that the key argument is parsed always before the non-key/complement arguments — the "key" QC-vector, as introduced in the previous section for a certain rule φ is unique for all key-mode application 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

[4] These QC-vectors are shown on the bottom line in the (somehow) synoptic table in Figure 1.

[5] A more suggestive for the two run-time computed QC-vectors notation would be perhaps $\text{QC}(\varphi'', \psi)$ and $\text{QC}(\phi, \psi')$.

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

Figure 2: $\text{QC}_\pi(\psi)$ as function of $preComp(\psi,\pi)$.

pre-computed/preliminary form of QC-vectors, which can be computed at the 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 the run time starting form the pre-computed forms, dependent on the already parsed arguments.

The QC test idea as introduced in the previous section is very simple: given a finite set of feature paths $\Pi = \{\pi_1, ..., \pi_m\}$, and the feature structures ψ_1, ψ_2, check whether

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

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 $\psi.\pi\uparrow$).

Important Remark: Actually, if $\pi' = f_1.\ \ldots\ .f_j$ is the longest prefix of $\pi = f_1.\ \ldots\ .f_n$ 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 to 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 (for the QC-vectors), so it is good to use it at compile time, but it is not recommendable at run time, because it would consume additional time and space. At the run time, the previous solution, based on appropriateness constraints is preferable.[6]

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

1. If ψ is a rule argument without containing references to precedent[7] arguments — this is the case of the first/head-corner argument in simple/head-corner chart-based parsing — then we define

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

2. If ψ is a rule argument with references to substructures of the precedent arguments, $\mathcal{X}$ is the set of all variables/tags in ψ which refer to precedents arguments (according to the parsing order), and

[6] Our claim that this *Remark* invalidates the the opinion of PET's author [2] who stated that partial expansion [6] is reducing the quick-check's efficiency.

[7] Here the term "precedent" is used in the sense of the parsing order.

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

Figure 3: The OSF-term associated to a `sentence` rule.

failure paths	key QC-vector	preComp	complete QC-vector
π_1 = HEAD	*verb*	*noun*	*noun*
π_2 = OBJECT	*np*	#2.OBJECT	$\top$
π_3 = HEAD.AGREEMENT	*agr*	#3	*3sg*

Figure 4: The active ("key" and "complete") QC-vectors for the `sentence` rule.

$\pi = f_1 f_2 \ldots f_n$ 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 2.

Important Remark: According to the Remark made in the Introduction, in Figure 2 the underlined expressions are in fact extended to $\mathrm{QC}_\pi(\varphi) = \ldots$, for any feature structure φ subsumed by ψ.

Note that if, as 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 to the precedent 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 that $\mathcal{X}$ is taken as the set of all variables/coreferences shared between the rule's *LHS* and the arguments (*RHS*).

In the Light system, a pre-computed QC-vector is stored as an array of tuples of the form (s, sort), (i, int), ($-j$, int), with $i \geq 0$, and $j > 0$, while (at run-time) a QC-vector is represented simply as an array of sorts.

Example

Let us consider — adapted from [17] — a simple rule made of a context-free backbone $s \rightarrow np * vp$ augmented with feature constraints like in Figure 3. (The $*$ sign marks the rule's head/key argument.) Suppose that we want to consider the failure paths π_1 = HEAD, π_2 = OBJECT, π_3 = HEAD.AGREEMENT. The "key" QC-vector and the two "complete" QC-vectors (the *preComp* form and respectively the final form) are shown in Figure 4. The *preComp* QC-vector is shown in a more intuitive form than in the formalisation given in Section 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

```
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 ]
```

paths	QC-vector
π_1	*verb*
π_2	$\top$
π_3	*3sg*

```
np
[ ARGS < the
         [ HEAD det ],
         cat
         [ HEAD #9:noun
                [ AGREEMENT 3sg ] ] >,
  HEAD #9 ]
```

paths	QC-vector
π_1	*noun*
π_2	$\top$
π_3	*3sg*

Figure 5:
The parses corresponding to the vp *catches a mouse* and the np *the cat*,
and the computed "passive" QC-vectors.

for π_2 is $\top$ since the FS corresponding to the noun phrase *a mouse* doesn't have the OBJECT feature defined.)

One can easily see that the vp feature structure corresponding to the verb phrase *catches a mouse*, as shown in Figure 5, passes the QC test with the key QC-vector presented in Figure 4.

Then the np FS shown in Figure 5 for the noun phrase *the cat* passes the QC test in conjunction with the complete QC-vector in Figure 4, 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 subsorts of *agr*.

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 (g|b(s_π, $\mathsf{QC}_\pi(\varphi) = \bot$) return FALSE;

where $s_\pi = \mathsf{QC}_\pi(\psi.\text{KEY-ARG}) = \mathit{preComp}(\psi.\text{KEY-ARG}, \pi)$ 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:[8]

set $\mathsf{QC}_\pi(\psi')$, t_π

where $\psi' = \psi.$NON-KEY-ARG and

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

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

$2''$. 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 ($ii.$) perform the QC test, in fact a sequence of conditional statements of the following form, one for each QC path π:

if (glb($\mathsf{QC}_\pi(\psi.$NON-KEY-ARG), $\mathsf{QC}_\pi(\varphi) = \bot$) return FALSE;

Note that $\mathsf{QC}_\pi(\psi.$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 section, when we presented the pre-computed QC-vectors).

Possible improvements

The 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 the QC-filter efficiency):

– tests like $\top \wedge root(\phi)$, which in fact correspond to paths that are not fully defined in the argument being currently checked, must be eliminated since they always succeed;

– also, when using appropriateness constraints [3], tests like $s \wedge root(\phi.f)$ 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 $\mathrm{QC}_\pi(\psi) = \mathsf{heap}[X_j.f_{j+1}. \ldots .f_k.f_{k+1}. \ldots .f_n].\mathsf{SORT}$, and $\mathrm{QC}_{\pi'}(\psi) = \mathsf{heap}[X_j.f_{j+1}. \ldots .f_k.f'_{k+1} \ldots .f'_m].\mathsf{SORT}$, then $\mathrm{QC}_{\pi'}(\psi)$ can be computed as $\mathsf{heap}[Y_l.f_{j+l}. \ldots .f_k.f'_{k+1} \ldots .f'_m].\mathsf{SORT}$, where Y_l is the last Y definable variable in the sequence $Y_1 = X_j.f_1, ..., Y_k = Y_{k-1}.f_k$ is the sequence used to compute $\mathrm{QC}_\pi(\psi)$;[9]

– the sort glb tests (represented by the if statements) can be reordered, 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.

[8] 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.

[9] Note that the $Y_1 = X_j.f_1, ..., Y_k = Y_{k-1}.f_k$ sequence might not be entirely computed.

Indeed, one of the main *critics* that can be addressed to the QC-filter technique in the form presented in the beginning of this section (and used as such in the LKB, PAGE and PET systems) is that it is a grammar-level devised 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 to a passive item (completed rule) must contain/cover 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 dimension of the "personalised" QC-vectors would be significantly reduced.)

Evaluation and conclusion

Without the Quick Check pre-unification filter when running the LinGO grammar on the CSLI test suite, the Light system scored 0.07 sec/sentence. With Quick Check turned on, Light registered 0.04 sec/sentence. The compiled QC filter in Light provided thus a speed-up factor of 37%. 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 — and already explained in the introductory section — 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. Otherwise said, one has to keep in mind that in Light the specialised compiled form of rules already speeds up significantly the parsing, before applying the QC filter. However this factor can be further increased by implementing the above mentioned *improvements*.

Those improvements apply also to interpreter-like parsing systems, and the technique for compiling the QC filter here presented is in our opinion easily transferable to other compilers for parsing with unification-based grammars.

The outlined compilation schema for the Quick Check pre-unification test is by no means HPSG dependent. Moreover, it is basically independent of the variant of (order-sorted) feature constraint logics that supports parsing/deduction (which in turn calls unification). In Light we used as logic background the order- and type-consistent OSF-theories [5], a slightly more general class of typed feature structures than that (of appropriate FSs) defined by [3]. The technique here presented can therefore be applied to other systems dealing with typed-unification grammars like Amalia [18] [19] and LiLFeS.

Acknowledgements

Thanks go to Ulrich Callmeier for having had implemented the interpreted form of the QC filter for CHIC/ago,[10] the development prototype of the Light compiler. I wish to express special thanks to Professor Hans Uszkoreit for the kind 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 Saarbruecken, Germany.

This paper was written while the author was supported by an EPSRC grant in the framework of the ROPA project at the Computer Science Department of the University of York. The Light system is used in this new project to learn typed-unification grammars within the Inductive Logic Programming framework [13].

[10] CHIC stands for Compiling Hpsg Into C. The CHIC/ago name must pronounced exactly like Chicago.

References

[1] H. Aït-Kaci and A. Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16:195–234, 1993.

[2] 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.

[3] B. Carpenter. *The Logic of Typed Feature Structures – with applications to unification grammars, logic programs and constraint resolution.* Cambridge University Press, 1992.

[4] L.-V. Ciortuz. Scaling up the abstract machine for unification of OSF-terms to do head-corner parsing with large-scale typed unification grammars. In *Proceedings of the ESSLLI 2000 Workshop on Linguistic Theory and Grammar Implementation*, pages 57–80, Birmingham, UK, August 14–18, 2000.

[5] L.-V. Ciortuz. Compiling HPSG into C. Research report, The German Research Center for Artificial Intelligence (DFKI), Saarbruecken, Germany, and the Computer Science Department, University of York, UK, 2001. (In preparation).

[6] L.-V. Ciortuz. Expanding feature-based constraint grammars: Experience on a large-scale HPSG grammar for English. In *Proceedings of the IJCAI 2001 co-located Workshop on Modelling and solving problems with constraints*, Seattle, USA, August 4–6, 2001.

[7] A. Copestake. *The (new) LKB system.* CSLI, Stanford University, 1999.

[8] A. Copestake, D. Flickinger, and I. Sag. *A Grammar of English in HPSG: Design and Implementations.* Stanford: CSLI Publications, 1999.

[9] M. Kay. Head driven parsing. In *Proceedings of Workshop on Parsing Technologies*, Pittsburg, 1989.

[10] 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.

[11] 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.

[12] 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.

[13] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

[14] S. Oepen and J. Caroll. 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.

[15] S. Oepen, D. Flickinger, H. Uszkoreit, and J. Tsujii. Introduction to the special issue on efficient processing with HPSG: Methods, systems, evaluation. *Journal of Natural Language Engineering*, 6 (1), 2000.

[16] C. Pollard and I. Sag. *Head-driven Phrase Structure Grammar*. Center for the Study of Language and Information, Stanford, 1994.

[17] N. Sikkel. *Parsing Schemata*. Springer Verlag, 1997.

[18] S. Wintner. *An Abstract Machine for Unification Grammars*. PhD thesis, Technion – Israel Institute of Technology, 32000 Haifa, Israel, 1997.

[19] S. Wintner and N. Francez. Efficient implementation of unification-based grammars. *Journal of Language and Computation*, 1(1):53–92, 1999.