ON SPECIALISED COMPILATION OF RULES IN UNIFICATION GRAMMARS

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

One central concept in our approach to compiled parsing with feature-based unification grammars in the Light system [2] is the specialised compiled form of rules, which is obtained via transformation of the abstract code generated by the OSF AM [1] for rules represented as feature structures.

Like AMALIA [10], the Light system has specialised abstract instructions to implement the (compiled) parsing. But the parser we implemented for Light is significantly more general than that in AMALIA: 1. it is a head-corner bottom-up chart-based parser (AMALIA's parser is a simple bottom-up chart-based one); 2. it uses feature structure (FS) sharing to save space and time needed for parsing; 3. it also integrates the so-called quick-check technique [7] to reduce the unification time for rule arguments, while benefiting from statistics results computed on test suites. We briefly present here the first optimisation mentioned above. The second optimisation is presented in detail in [3], while the third one makes the object of the [4] paper.

Specialised compilation design for unification grammar rules in Light must be done in such a way that their application be suitable and efficient for the active bottom-up chart-based head-corner parsing [6].[1] Compared to the general setup of compiled unification of feature structures, specialised compilation of rules adds an important "ingredient": the incremental treatment of rules' arguments, i.e. interleaving arguments' processing with (parsing-oriented) control operations.

The technique we chose in order to obtain the specialised compiled form of rules — assuming that they are represented as feature structures — is *program transformation*. Starting from the abstract code delivered by the OSF/Light AM compilation of the feature structure representing a rule in "program" mode, we will upgrade it with specialised control sequences for the rule's application. Thus, our specialised rule compilation task consists mainly in defining specialised *control instructions*, and simple *transformation actions* on the abstract code. When executed, these actions basically insert into abstract code certain sequences of control instructions. These control instructions will trigger (from within the parser) the unification of rule arguments (with feature structures associated to passive items) and the construction of the rule's mother/*LHS* feature structure.

In Light AM there are two possible *application modes* for (compiled) grammar rules:

— the *key mode:* unify the rule's key argument with the feature structure corresponding to a *passive item*.[2] If success is reported, then the needed coreferences (more precisely: the values of the abstract

[1] To differentiate the notion of head in HPSG from that used for head-corner parsing, we adopt the convention proposed by LinGO developers to use the term *key* instead of *head* for parsing, therefore in the sequel we will use the terminology *key-corner* parsing. The notion of *head* will be reserved for HPSG/linguistics usage.

[2] All lexical items are passive items; non-lexical passive items are obtained during the parsing process, as shown in

machine's X registers whose indices are coreferences) and the changes made on the heap during unification are saved in a newly created *environment*. The index of this new environment, stored in the register E will be transmitted to the parser, and it will record E's value in a newly created *active item*;

— the *complete mode* (only for non-unary) rules: restore the environments corresponding to the already parsed/instantiated arguments of the rule and unify one of the "active" (i.e., not yet instantiated/parsed) rule arguments with the feature structure corresponding to a *passive item*. If unification succeeds, then a new environment is created as above; moreover, if after successful unification the argument list is exhausted, then a feature structure corresponding to the left hand side (*LHS*) of the rule is constructed on the heap, and a passive item is registered on the chart, otherwise we register an active item. If unification fails, then the changes done on the heap during argument unification will be undone.

The switch between the two possible modes for rule application is done by examining the register E when calling for the rule's application. It will be -1 for the key mode. When applying a binary rule in the complete mode, E will store the index of the environment corresponding to the key argument.

Remark: In order-sorted (i.e., inheritance based) feature grammars, the distinction between the two main operations 'scan' and 'complete' (by which the input string is consumed) is no longer possible, since the root sort of the arguments in the *RHS* of a rule can have — and in HPSG usually have! — as subsorts both lexical (i.e., terminal) symbols and phrase (i.e., non-terminal) symbols. It is often the case that arguments in the rules' *RHS* in lexicalized grammars like HPSG are sort-underspecified (usually *sign*- or even *Top*-sorted), because 1. the aim of building such grammars is to come up with a very limited number of rules (or better: rule schemata) and 2. their selection during parsing is determined mainly by checking the satisfiability of the associated feature constraints. This makes impossible/impractical the prediction (of the symbol to be tried/parsed next) as usually defined in the parsing theory. Therefore, apart from accepting here the *head-corner* item deduction (as given by the unification grammar parsing schemata in [9]), we override here the term *complete*, and make it generalise both the 'scan' and 'complete' notions as defined for instance in [9].

Note that in certain conditions, saving the trail in a new environment may be postponed. The specialised compilation of rules in the current implementation of Light AM is limited to binary and unary rules since LinGO [5] — the large-scale HPSG grammar for English implemented at CSLI, University of Stanford — demonstrated that binary rules are perfectly convenient for expressing sophisticated HPSG knowledge. Generalisation to rules of arbitrary length is not difficult. (Our system could however deal with arbitrary long rules, in a version that compiles rules as ordinary feature structures.) In the sequel, when not otherwise explicitly stated, we will refer to binary rules, because their treatment is of course more elaborated than that of unary rules.

Technically, for the specialised compilation of a rule via program transformation a new feature *KEY-ARGS* is introduced, and its value will be a list obtained from the rule's arguments (*ARGS*) list simply by duplicating it (i.e., by coreferring the elements) and then moving the key argument on the first position. The feature structure describing a rule has to satisfy the following two *well-formedness conditions*: *i*. the *KEY-ARGS* feature is the first one among those associated to the rule's root, and *ii*. every coreference has all associated (sort and feature) constraints listed at its first occurrence. Note that the first well-formedness condition stated above ensures the partitioning of the abstract code into

the sequel.

the areas *ARG1*, *ARG2*, and *LHS* (all having both "read" and "write" parts), while the second one allows the removal of the *LHS-read* area in (the program transformation process that will produce) the new compiled form of the rule.

In the case of a binary rule, it is exactly at the slots *S1*, ..., *S6* delimiting the areas *ARG1*, *ARG2*, and *LHS* in the two-stream OSF abstract code of the rule's feature structure that control sequences for doing parsing with this rule will be placed. Newly designed abstract instructions — *saveEnv* and *restoreEnv* are used at/by the control sequences placed (via abstract code transformations) at the slot places *S1*, ..., *S6*. An *environment* is a couple of *i.* a set of indices corresponding to coreferenced X variables, together with their values (which represent indices/addresses of heap cells) and *ii.* a trail copy that registers the changes done on the heap during unification.[3] Also, environments will include information useful for the (compiled form of) quick-check filtering.

Example: Consider the next vp rule inspired by [9]. Its non-specialised (OSF) abstract code can be easily get following the guidelines in [1], while its specialised compiled form in Light is given in below.

```
vp
[ ARGS    < verb
            [ HEAD    #1,
              OBJECT  #3:np,
              SUBJECT #2:sign ],
            #3 >,
  HEAD    #1,
  SUBJECT #2 ]
```

```
vp:   set corefs, { 3, 4, 5 }
      cond E != -1, jump R3

                                R0: % ARG1                                    %S1
                                   set X[2], Q
                                      intersect_sort X[2], verb
                                      test_feature X[2], HEAD, X[3], 3, W3, verb
                                R1:   test_feature X[2], OBJECT, X[4], 3, W4, verb
                                      intersect_sort X[4], np
                                R2:   test_feature X[2], SUBJECT, X[5], 3, W5, verb
                                   jump W6                                    %S2
                                R3: % ARG2
                                   restoreEnv E                               %S3
                                      cond unify( X[4], Q ) = FALSE, Failure
                                R5:jump W0
                                      %S4
W3: % ARG1
      push_cell X[3]
      set_feature X[2], HEAD, X[3]
      write_test 3, R1
W4:   push_cell X[4]
      set_feature X[2], OBJECT, X[4]
      set_sort X[4], np
      write_test 3, R2
W5:   push_cell X[5]
      set_feature X[2], SUBJECT, X[5]
W6: saveEnv corefs                    %S5
    jump W8                           %
    % ARG2
W0: % LHS
    saveEnv NULL                      %S6
    set Q, H                          %
W1:   push_cell X[0]
      set_sort X[0], vp
W7:   set_feature X[0], HEAD, X[3]
      set_feature X[0], SUBJECT, X[5]
W8:
```

Apart from the (basic) fact that the parsing control instructions replace the *KEY-ARGS* list-oriented stuff at the control slots other transformations are done: 1. The *LHS-read* part is deleted, since it is no longer needed: once the two arguments unify (with two certain feature structures represented on the abstract machine's heap), we have to built/write the *LHS* feature structure; no "read" action is any longer needed. For the same reason, the *write_test* instructions are eliminated from

[3] Actually, the trail content will be saved in the (corresponding part of an) environment in a compressed form.

the *LHS-write* area. 3. The *ARGS* feature is "discarded" i.e., not created in the *LHS* code.[4] Other, interesting details on this abstract program transformation schema for parsing rules in unification grammars are provided in [3].

This strategy of specialised compilation of rules in Light provided us a factor of speeding up of 2.75 on the test suite provided by the CSLI, University of Stanford for the LinGO grammar.

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 conception and implementation side of the work here reported was done while the author worked at the LT Lab of the German Research Center for Artificial Intelligence (DFKI) in Saarbrücken, Germany, and he would like to express here his gratitude for the possibility he had to develop the Light system there.

References

[1] H. Aït-Kaci and R. Di Cosmo. Compiling order-sorted feature term unification. Technical report, Digital Paris Research Laboratory, 1993. PRL Technical Note 7, downloadable from http://www.isg.sfu.ca/life/.

[2] 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.

[3] 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).

[4] L.-V. Ciortuz. On compilation of the Quick-Check filter for feature structure unification. In *Proceedings of the IWPT 2001 International Workshop on Parsing Technologies*, Beijing, China, October 17–19, 2001.

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

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

[7] 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.

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

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

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

[4] This omission is supported by the Locality Principle in the HPSG theory [8], and is adopted in the Light setup, as it was implemented in the other LinGO-parsing systems.