

# On specialised compilation of rules for head-corner bottom-up chart-based parsing with unification grammars

Liviu Ciortuz

CS Department, University of York  
Heslington, York, YO10 5DD, UK.  
E-mail: ciortuz@cs.york.ac.uk.\*\*

**Abstract.** This paper presents the compilation approach for head-corner bottom-up chart-based parsing implemented in our ABC Light system. Compilation in ABC Light is done via an abstract machine which substantially expands OSF AM, the abstract machine designed for OSF-unification.

The central concept in our approach to compiled parsing with feature-based unification grammars is the specialised compiled form of rules, which is obtained via transformation of the abstract code generated by the OSF AM for rules represented as feature structures. Specialised compilation through abstract code transformation ensures modularity and compactness of the OSF→C compiler component in the ABC Light system.

## 1 Introduction

Significant progress has been achieved during the last couple of years in the area of efficient processing with feature-based grammars. A recent work [17] presented some of the most advanced results concerning parsing with large-scale HPSG grammars [19], notably the LinGO grammar [11] for English developed at CSLI, University of Stanford. In the meantime, while still under development, our ABC Light compiler [7], designed to do head-corner parsing with feature constraint-based grammars, provided on LinGO parsing results which are competitive with the best results reported in [17].

With respect to the the parsing issue itself, the systems dealing with LinGO-like grammars divide into two categories:

1. Systems which implement a language (extension) in which one or different suitable parsers can be written:

- LiLFes implements a Prolog/LOGIN-like control level above unification;
- ALE compiles typed feature structures into Prolog terms and uses (almost) for free the Prolog environment as a host layer, in which the user can test and extend different (eventually already available) parsing strategies.

The advantage in this approach is the flexibility the user has to choose the most suitable parser for his needs.

2. LKB [9], TDL/PAGE [15], PET [5] as interpreters on one side, and *AMALIA* [22] [23] and ABC Light as compilers on the other side have opted for built-in parsers. The advantage is (presumably) the speed up due to the dedicated implementation of the parser; the disadvantage: as soon as the user wants to change/adjust the parser, he has to modify the system's source code.

---

\*\* This is the extended version of the paper "On specialised compilation of rules in unification grammars" published in the Proceedings of the International Workshop on Parsing Technologies (pp. 209-212), held at Beijing University, China, 17-19 October, 2001. The implementation side of the work here reported was done while the author was with the LT Lab of the German Research Center for Artificial Intelligence (DFKI) in Saarbrücken, Germany.

ABC Light is a compiler that implements Light, a simple but interesting CLP(OSF) language [12] [13]. Light stands for LIGHT — Logic, Inheritance, Grammars, Heads, and Types.<sup>3</sup> Unification in Light is OSF-theory unification on order- and type-consistent theories (which slightly extend well-typed<sup>4</sup> systems of feature structures [6]). Deduction in Light is a head-corner version of bottom-up chart-based [14] [21] parsing-oriented deduction [20]. Compilation in ABC Light is done via an abstract machine called Light AM, which substantially expands the abstract machine designed for OSF-unification [2] (we will call it OSF AM in the sequel). The code produced by Light AM is further translated down into C in a similar manner to the `wamcc` approach [10]. (Extending the OSF AM so to perform OSF-theory unification was the first main task in building our system [7].)

Like *AMALIA*, the ABC Light system has specialised abstract instructions to implement the (compiled) parsing. The parser we implemented for Light is significantly more general than that in *AMALIA*:

- it is a head-corner bottom-up chart-based parser (*AMALIA*’s parser is a simple bottom-up chart-based one);
- it uses feature structure (FS) sharing to save space and time needed for parsing;
- it also integrates the so-called quick-check technique [16] to reduce the unification time for rule arguments, while benefiting from statistics results computed on test suites.

The above last two optimisations are presented in detail in [8]. The two main sections of this paper are concerned with the specialised compilation of rules in ABC Light by means of program transformation, starting from the abstract code of the feature structures representing rules (Section 2), and respectively the overall conception of the parsing control level in the ABC Light compiler system (Section 3). A final, *evaluation* paragraph provides measurements obtained with ABC Light on the CSLI test suite.

## 2 Specialised compilation of head-corner (binary) parsing rules: basic idea and exemplification

Specialised compilation design for unification grammar rules in ABC 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.<sup>5</sup>

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.)

Compared to the general setup of compiled unification of feature structures, specialised compilation of rules adds two important “ingredients”:

- feature structure sharing (as a means to eliminate or reduce feature structure copying);
- incremental treatment of rules’ arguments, i.e. interleaving arguments’ processing with (parsing-oriented) control operations.

<sup>3</sup> The analogy with the name of LIFE — Logic, Inheritance, Functions and Equalities — a well-known constraint logic language based on the OSF constraint system [4] is evident. LIFE was intended (but not limited) to be used in NLP applications [3]. However, up to our knowledge, no large-scale application was implemented in LIFE.

<sup>4</sup> An order- and type-consistent set of FSs is more general than a well-typed since it supports: *i.* open FSs, *ii.* the definition of a “canonical” set of so-called appropriateness constraints (automatically inferred), and *iii.* the restriction of the type unfolding condition  $\phi \sqsubseteq \Phi(t)$  to non-terminal nodes in the (typed) FSs, where  $\sqsubseteq$  denotes FS subsumption.

<sup>5</sup> Prior measurements done with CHIC/ago, the development prototype of ABC Light, have shown that head-corner chart-based parsing with LinGO is approximately twice as fast as simple chart-based parsing.

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 arguments’ unification and the construction of the rule’s mother/*LHS* feature structure.

More technically, our abstract program transformation task is made easier when the following facts are taken into account:

- the design of OSF AM — the abstract machine for OSF-term unification due to Aït-Kaci and Di Cosmo, integrates the “two-stream” optimisation that coordinates the *read* and *write* sequences of instructions performing the unification task;
- we upgraded that design with abstract instructions specialised in list processing; therefore the “slots” were the (parsing) control sequences must be placed are clearly identifiable by the places were abstract instructions in the input code implement the construction of the rule’s argument list.

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

- the *key mode*: unify the rule’s key argument with the feature structure corresponding to a *passive item*;<sup>6</sup> this feature structure is found on the heap at the address stored in the register Q. If success is reported, then the needed coreferences (more precisely: the values of the 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:<sup>7</sup> 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. The key mode is indicated by the value -1 for

<sup>6</sup> All lexical items are passive items; non-lexical passive items are obtained during the parsing process, as shown in the sequel.

<sup>7</sup> Note that 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 HPSG are sort-underspecified (usually *sign-* or even *Top-sorted*), because 1. the aim of lexicalized 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 [21]), we override here the term *complete*, and make it generalise both ‘scan’ and ‘complete’ notions as defined in [21].

the register E.<sup>8</sup> 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 demonstrated that binary rules are perfectly convenient for expressing sophisticated HPSG knowledge. Generalisation to rules of arbitrary length is not difficult.<sup>9</sup> 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.

In a feature-based unification grammar, a binary rule having a context-free backbone as

$$LHS \rightarrow ARG1 \ ARG2 \tag{1}$$

can be represented naturally as a feature structure in which a dedicated feature — here called *ARGS* — takes as value the list of arguments specified in the right hand side (*RHS*) of the above context-free backbone of the rule. For instance, to the following *vp* rule, which is taken from the sample unification grammar given in [21]

$$\begin{aligned} vp &\rightarrow \star verb \ np & (2) \\ vp.HEAD &\doteq verb.HEAD \\ vp.SUBJECT &\doteq verb.SUBJECT \\ verb.OBJECT &: np. \end{aligned}$$

one can associate the feature structure in the Figure 1. In (2), the key argument of the *vp* rule is the *verb*, marked with  $\star$ . The symbol # introduces coreferences. Lists are presented in sugared notation  $\langle \dots \rangle$ .

```

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

```

Fig. 1. The OSF-term associated to the rule (2)

Among all the equivalent OSF-terms/feature structures that could describe the rule (2), we have chosen the above one in a particular way, in order to later ensure both an efficient parsing with the resulting code of the rule and an elegant way to obtain this code through program (abstract code) transformation. More precisely, in the current set-up, the feature structure describing a rule has to satisfy the following two *well-formedness conditions*:

- the *ARGS* feature is the first one among those associated to the rule’s root;
- every coreference has all associated (sort and feature) constraints listed at its first occurrence.

<sup>8</sup> When applying a binary rule in the complete mode, E will store the index of the environment corresponding to the key argument.

<sup>9</sup> Our system could however deal with arbitrary long rules, in a version that compiles rules as ordinary feature structures.

Note that the first well-formedness condition stated above ensures the partitioning of the abstract code into 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.

*Convention:* In order to simplify the general presentation of our specialised compilation technique, we will assume that, like in the above example, for any feature structure representing rule, the head/key argument is the first one in the rule’s *ARGS* list.<sup>10</sup>

The Light specialised form of the rule (2), obtained by transformation of the code shown in Figure 2 is given in Figure 3. It is exactly at the slots *S1*, ..., *S6* delimiting the areas *ARG1*, *ARG2*, and *LHS* that control sequences for doing parsing with this rule will be placed.<sup>11</sup> In Figure 3, the parsing control instructions are indented to the left, to be better visualised. Technically, new 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.<sup>12</sup> Also, environments will include information useful for the (compiled form of) quick-check filtering.

Apart from the (basic) fact that the parsing control instructions replaced *ARGS* list-oriented stuff at the control slots shown in Figure 2, five other transformations can be noticed:

— The *LHS-read* part was 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 were eliminated from the *LHS-write* area, and so was also the *write\_test* immediately above the *LHS-write* area.

— Since our strategy for compiled parsing follows strictly the order *RHS* (*ARG1* and then eventually *ARG2*) → *LHS*, then the code lines between the label R0 and the slot *S1* are no-longer needed; they were eliminated.

— The slot *S4* from Figure 2 was simply deleted (so it has no correspondent) in Figure 3, and the code between the labels W2 and W3 was eliminated, since it becomes a dead code (the entry point for *ARG1* is R1).

— The code between the labels W0 and W1 was moved after the “write” code for *ARG2* (actually empty in the compiled form of the *vp* rule).<sup>13</sup>

— The *ARGS* feature is “discarded” i.e., not created in the *LHS* code. This omission is supported by the Locality Principle in the HPSG theory [18] [19], and is adopted in the Light setup, as it was implemented in the other LinGO-parsing systems.

*Notes:*

1. The example presented above illustrates what a binary rule looks like under specialised compiled form. It represents however, a particular case of a binary rule, namely one in which

<sup>10</sup> In the general setup, as this convention cannot be hold, we proceed as it follows: A new feature *KEY-ARGS* is introduced, and for every rule, its value will be a list obtained from the rule’s *ARGS* list simply by duplicating it (i.e., by coreferring the elements) and then moving the key argument on the first position. The above specialised compilation work is performed in practice on *KEY-ARGS*, and not on *ARGS* as presented above. So, *KEY-ARGS* will be the first feature in the rule’s frame, and the whole (thus extended) feature structure has to satisfy the second well-formedness condition, relative to coreferences.

<sup>11</sup> It is easy to identify the slots in the non-specialised code by retrieving first the abstract instructions *test\_feature* (or *unify\_feature*), and *set\_feature* that deal with the *ARGS* feature, and subsequently the related list-specialised instructions — *test\_inst\_list*, *get\_first*, *get\_rest*, *unify\_first*, *test\_NIL\_rest*, *set\_list* — concerning the *ARGS* list itself and its components.

<sup>12</sup> Actually, the trail content will be saved in the (corresponding part of an) environment in a compressed form.

<sup>13</sup> The code between W1 and W2 was dead in the design of the original OSF abstract machine; it has to be “revived” in this new, specialised approach for rules compilation.

```

R0:  intersect_sort X[0], vp           %ext. S1
      test_feature X[0], ARGS, X[1], 1, W1, vp
      intersect_sort X[1], cons       %S1
      test_inst_list X[1], 2, W2      %
      get_first X[1], X[2]           %
      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
R3:  get_rest X[1], X[6]              %S2
      intersect_sort X[6], cons       %
      test_inst_list X[6], 3, W7      %
      unify_first X[6], X[4]
      test_NIL_rest X[6]             %S3
R4:  unify_feature X[0], HEAD, X[3]   %LHS/read
      unify_feature X[0], SUBJECT, X[5] %
R5:  jump W8                          %

W0:  push_cell X[0]                    %dead code
      set_sort X[0], vp                 %
W1:  push_cell X[1]                    %S4
      set_feature X[0], ARGS, X[1]      %
      set_sort X[1], cons               %
W2:  push_cell X[2]
      set_sort X[2], verb
W3:  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]
      write_test 3, R3
W6:  push_cell X[6]                    %S5
      set_list X[1], X[2], X[6]         %
      set_sort X[6], cons               %
W7:  set_list X[6], X[4], NIL          %S6
      write_test 1, R4                 %LHS/write
      set_feature X[0], HEAD, X[3]      %
      set_feature X[0], SUBJECT, X[5]   %
W8:

```

**Fig. 2.** The OSF abstract code for the rule (2)

the second argument is previously coreferred, i.e, its index/tag occurs in the *ARG1* code area. The general case requires a few additional refinements that will be skipped here.

2. Particularities for unary rules: In the OSF (non-specialised) abstract code there are no *S2* and *S5* slots, and of course the *ARG2* (*read* and *write*) areas are missing. When running the specialised compiled code, things happen almost like in the key mode of binary rules, except that when unification (on the single argument) succeeds, then we enter directly the construction of the *LHS* feature structure, like in complete mode.

Full details on abstract program transformation for specialised rule computation are provided

```

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

                                     R0: % ARG1                               %CS1
                                     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                               %CS2
R3:  % ARG2
                                     restoreEnv E                           %CS3
                                     cond unify( X[4], Q ) = FALSE, Failure
R5: jump W0
                                     %CS4
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                               %CS5
     jump W8                                     %
     % ARG2
W0:  % LHS
     saveEnv NULL                               %CS6
     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:

```

**Fig. 3.** Specialised (Light) abstract code for the rule (2).

in [8]. (They are skipped here due to the lack of space.) In our Light AM implementation, the program transformations presented in this section are done “on fly”, i.e., for any rule the specialised, “ready for parsing” form is the only one to be actually produced by the (OSFC) compiler, if specialised compilation was demanded.

### 3 The parsing control level in ABC Light

A simplified version of the active bottom-up chart based (ABC) parser written in (Light) abstract code is presented in Figure 4. One can easily recognise several new abstract instructions. Among them there are two which are used in the main loop to build up parses: `K_corner` and `complete`. They are shown in C-like pseudo-code in Figure 5. The `build_key_item` and `build_complete_item` procedures transmit the execution control to the compiled rules, in *key-*, respective *complete-* mode. It is exactly from inside these procedures that rules are called. We will explain them later in detail.

A few words about the (general) abstract instruction that build up the parser in Figure 4:

```

Start: init_machine
      get_grammar
      create_lexicon
      init_parser
ParseLoop:
      cond empty( get_input( buffer ) ), Stop
      tokenize buffer, lexInput
      init_chart lexInput, chartHeap
ProcessItem:
      cond empty( agenda ), Print
      set item, chartHeap[ pop( agenda ) ]
      cond passive( item ), DirectComplete
      K_corner item, length( lexInput )
ReverseComplete:
      complete item, False
      goto EndProcess_item
DirectComplete:
      complete item, True
EndProcess_item:
      goto ProcessItem
Print: print_results chartHeap
      reset_machine
      goto ParseLoop
Stop:

```

Fig. 4. The ABC head-corner parser abstract code

- `init_machine`: allocates space for the main data structures of the Light abstract machine and initialises registers;
- `reset_machine`: resets registers in Light AM to initial/implicit values;
- `get_grammar`: initialises and computes the “key” QC-vectors used for pre-unification filtering (see the next section), and the “restrictors”; restrictors are names of certain features, that will be eliminated from a rule LHS FS after successful unification of rule arguments (such features are grammar and/or (HPSG) theory dependent);
- `create_lexicon`: builds up a hash table whose entries put words in correspondence with lexical feature structures (compiled as query terms) in the input Light grammar; this hash table will be used during the lexical analysis of the sentences to be parsed;
- `init_parser`: allocates memory for the chart, the agenda and the input buffer to be used during the parsing;
- `tokenize`: performs the morphological analysis on the input sentence;
- `init_chart`: for each morphological token, a certain lexical rule is applied to a specific lexical description (according to the lexicon), in order to build a full FS associated to a lexical token, and a lexical item will be built on the chart; the indices of all (passive) lexical items are registered onto the agenda;
- `print_results`: its name is self explaining.

The `K_corner` abstract instruction tests for a given passive item which are the potential rules to match (the FS corresponding to) this item against their key argument. For each successful application of such a rule, a new item is built on the chart. This item will be a passive one in the case of unary rules, and an active one otherwise. The `complete` abstract instruction works in two modes, according to the value of its second argument `isActive`. For active (i.e., incomplete) items, which were just/newly obtained as a result of a successful execution of the `K_corner` abstract instruction, the `complete` procedure works in “direct” mode, trying to produce new complete items, by matching the not-yet-filled position in the current item with complete items

already found on the chart. For complete items read from agenda, complete works in “reverse” mode, searching for active items on the chart, and trying to combine them with the current item.

```

K_corner (passive:item*, input_length:int) ≡
{
  for all non-lexical rules r in the grammar {
    E = -1;
    if filter( r, passive, 0 ) ∧
      ((isLeftExtendable( r ) ∧ passive->start >= r.arity-1) ∨
       (¬isLeftExtendable( r ) ∧ passive->end + r.arity-1 <= input_length))
      build_key_item( r, passive );
    else ; }
}

complete (it:item*, isActive:boolean) ≡
{
  chart_entry:c; ile:boolean;
  if isActive { % direct mode
    ile = item_left_extendable( it );
    for every passive item c on the chart {
      E = it->env;
      if filter( it->rule, c ) ∧
        ((ile ∧ c.end = it->start) ∨ (¬ile AND c.start = it->end))
        build_complete_item( it, &c );
      else ; } }
  else % reverse mode
    for every active item c on the chart {
      E = c->env;
      ile = item_left_extendable( c );
      if filter( c->rule, it ) ∧
        ((ile ∧ c.start = it->ends) ∨ (¬ile ∧ c.end = it->start))
        build_complete_item( &c, it );
      else ; }
}

```

Fig. 5. The key and complete parsing abstract instructions

As ABC Light emulates abstract code into C, every (FS representing a) rule is finally associated a C function which returns 0 if the rule fails and 1 if it was successfully applied. Also, an array is created at the compilation time, storing the addresses of the C functions which are the compilation result for FSs representing rules. A rule application is invoked by calling the function rule with two arguments: the rule index and the root address of the FS on which this rule has to be applied. At the end of a successful application of a binary rule in complete mode, the Q register contains the root of the newly created (mother) FS. Knowing this, the code for the functions build\_key\_item and build\_complete\_item shown in Figure 6 is now easily understandable.

The OSF→C compiler component of the ABC Light system computes (among other data) the *combinatorial filters* lexfilter and synfilter. These are multi-dimensional boolean arrays satisfying the property

$$filter[fs, r, n] = 1 \text{ iff the FS } fs \text{ unifies with the } n\text{-th argument of the syntactic rule } r.$$

```

build_key_item( int r, item *passive )
{
    item *new_item;
    if rule( r, passive->fs ) {
        new_item = build_new_item( passive, NULL, passive->fs );
        agenda = agenda ∪ { new_item }; } % put new_item on agenda
    else ;
}

build_complete_item( item *active, item *passive )
{
    restoreEnv( active->env );
    if rule( active->rule, passive->fs )
        build_new_item( passive, active, Q );
    else ;
}

```

**Fig. 6.** The `build_key_item` and `build_complete_item` procedures.

The FS *fs* is either one denoting a rule (in the case of `synfilter`), or a FS associated to a lexical entry (in the case of `lexfilter`).

The filter test in the `K_corner` and `complete` procedures looks up into the (lexical or syntactic) filter tables to decide whether the FS associated with the passive item under consideration potentially unifies with the key (respectively non-key) argument of a certain rule. The index of this rule is *r*, the first argument of `K_corner`, respectively the rule field of the current active item in `complete`. In order to distinguish whether it has to check for the key, respectively the non-key argument, the filter function (whose code is shown in Figure 7) tests the content of the register *E*. The rule field in the passive item passed as argument to the filter function makes it possible to decide whether this passive item is a lexical or a non-lexical one. According to this information, either the `lexfilter` or the `synfilter` is looked up.

```

boolean filter( int r, item *passive )
{
    int filter_table( passive->rule < 0 ? lexfilter : synfilter );
    int arg = ( E == -1 ? RULES[ r ].key_arg : RULES[ r ].arity-1 - RULES[ r ].key_arg );
    return filter_table[ passive->rule ][ r ][ arg ];
}

```

**Fig. 7.** The combinatorial filter function.

`RULES` is an array computed at the compilation time, synthesising information about the syntactic rules in the input Light grammar: the rule name, arity, key-argument position and its key quick-check vector.

As an *example* of rule application, when the system will parse the input sentence “*The cat catches a mouse*”, also taken from [21], the *vp* rule will be applied once in key mode unifying *ARG1* with the feature structure given as lexical description for the verb *catches*, and then — after parsing the noun phrase *a mouse* — in complete mode, unifying *ARG2* with the feature structure describing this noun phrase.

More detailed: When parsing the phrase “*catches a mouse*”, the execution flow is as it follows:

- Before entering the rule in key mode, the register **E** is set to -1, and **Q** is instantiated to the root address (on the heap) for the feature structure corresponding to the lexical description of the verb *catches*.

- When entering the *vp* rule code, the set of coreference indices (**corefs**) is instantiated to {3, 4, 5}, since **X[3]**, **X[4]**, **X[5]** in the abstract code correspond to #1, #3, #2 respectively in Figure 1.

- Then, as **E** is -1, the sequence of instructions corresponding to *ARG1* — namely from the label **R0** to **R2** and **W3** to **W6** — is executed, leading to unification of the verb-rooted feature structure in Figure 1 with the term corresponding to the lexical entry *catches*.

- At **W6**, a fresh environment is created, incorporating

- the coreferences: the values of **X[3]**, **X[4]**, **X[5]**, namely the indices of the heap cells rooting the feature structures #1, #3, #2 in Figure 1, and
- the history of changes done on the heap during unification, namely new sort, feature and equality constraints (corresponding respectively to the fields **SORT**, **FTAB**, **CREF**).

- When exiting the *vp* rule execution in key mode (**jump W8**), these changes on the heap will be undone, leaving the feature structure of the verb *catches* exactly as it was before unification (with the first argument of the rule **vp**), to be further available for “sharing” by any other rule.

- Later on, after having had parsed the input sequence *a mouse*, and before entering again the *vp* rule, now in complete mode, the index (different than -1) of the above created environment is stored in the register **E**, and the content of the register **Q** is set to the heap address of the feature structure characterising the noun phrase *a mouse*.

- Now, reentering the *vp* rule,  $E \neq -1$  dictates the execution of the *ARG2* sequence (**R3**–**R5**). At **R3**, the environment of index **E** is restored, meaning that **X[3]**, **X[4]**, **X[5]** point again to the heap representations of the terms tagged by #1, #3, #2 in Figure 1, and the verb argument in the same figure is instantiated to its glb with the feature structure for *catches*, exactly as it was at the end of the *vp* rule application in key mode. Then the feature structure rooted by the cell **X[4]**, (corresponding to #3 in Figure 1), is unified with the feature structure characterising the noun phrase *a mouse*.

- As this last unification succeeds, a new environment is created (with an empty set of coreferences), corresponding to the instantiation of the FS rooted at #4:np in Figure 1 to its glb with the feature structure for the noun phrase *a mouse*.

- Finally, the *vp* mother/*LHS* feature structure is constructed, by executing the **W0**–**W8** sequence. After exiting the *vp* rule, changes on the heap due to environment restoring and unification are undone.

## Evaluation and Conclusion

In the ABC Light compiler which implements Light, a simple CLP(OSF) language suitable to process large-scale HPSG-like typed grammars, unification is “controlled” by a parsing-oriented level (that corresponds to the SLD-resolution level in the classical WAM [1]); in this respect our approach — head-corner active bottom-up chart-based parsing — is more general than the simple bottom-up chart-based parsing in *AMALIA*[23]. Also, the specialised compilation of rules by transformation of the OSF AM abstract code ensures modularity and compactness of the OSF→C compiler component in the ABC Light system.

Without the pre-unification filter *quick-check*, on the CSLI test suite, ABC Light scored 0.07 sec/sentence (while PET— known as the fastest system running LinGO [17], and currently supported/used by several commercial companies — reported 0.11 sec/sentence). The tests were run on a SUN Sparc server at 400MHz. With *quick-check* turned on, the ABC Light system registered 0.04 sec/sentence, a performance slightly better than that of PET. PET’s good performance w.r.t. ABC Light is due to the incorporation of graph unification library procedures and to the fact that computations in PET are highly localised. This second optimisation made PET about 40% faster; we plan to apply it soon for ABC Light, along with the overall polishing.

## References

1. H. Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, Cambridge, MA, 1991.
2. H. Ait-Kaci and R. Di Cosmo. Compiling order-sorted feature term unification. Technical report, Digital Paris Research Laboratory, 1993.
3. H. Ait-Kaci and Patrick Lincoln. LIFE—a natural language for natural language. In *T.A. Informations*, 30(1-2), pages 37–67. 1989. Paris, France.
4. H. Ait-Kaci and A. Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16:195–234, 1993.
5. 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.
6. B. Carpenter. *The Logic of Typed Feature Structures – with applications to unification grammars, logic programs and constraint resolution*. Cambridge University Press, 1992.
7. L. 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, 2000. Downloadable from <http://ling.osu.edu/~dm/events/esslli00/proceedings/index.html>.
8. L. Ciortuz. LIGHT — a feature constraint language applied to parsing with large-scale HPSG grammars. Unpublished technical report, The German Research Center for Artificial Intelligence (DFKI), Saarbruecken, Germany, 2001.
9. A. Copestake. *The (new) LKB system*. CSLI, Stanford University, 1999.
10. D. Diaz and P. Codognet. WAMCC: Compiling Prolog to C. In *Proceedings of ICLP'95, the 12th International Conference on Logic Programming*, Tokyo, Japan, 1995. MIT Press.
11. 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.
12. J. Jaffar and J.-L. Lassez. Constraint logic programming. In *The 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, January 1987.
13. J. Jaffar and M. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19(20):503–582, May-July 1994.
14. M. Kay. Head driven parsing. In *Proceedings of the 1st Workshop on Parsing Technologies*, pages 52–62, Pittsburg, 1989.
15. H.-U. Krieger and U. Schäfer. TDL – A Type Description Language for HPSG. Research Report RR-94-37, The German Research Center for Artificial Intelligence (DFKI), 1994.
16. 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.
17. 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.
18. C. Pollard and I. Sag. *An information-based syntax and semantics*, volume I. CSLI Publications, Stanford, 1987.
19. C. Pollard and I. Sag. *Head-driven Phrase Structure Grammar*. CSLI Publications, Stanford, 1994.
20. S.M. Shieber, Y. Schabes, and F. Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming*, pages 3–36, 1995.
21. N. Sikkil. *Parsing Schemata*. Springer Verlag, 1997.
22. S. Wintner. *An Abstract Machine for Unification Grammars*. PhD thesis, Technion – Israel Institute of Technology, Haifa, Israel, 1997.
23. S. Wintner and N. Francez. Efficient implementation of unification-based grammars. *Journal of Language and Computation*, 1(1):53–92, 1999.