Scaling up the abstract machine for unification of OSF-terms to do head-corner parsing with typed unification grammars

Liviu-Virgil Ciortuz

DFKI, LT Lab, Stuhlsatzenhausweg 3
Saarbrücken, Germany.
E-mail: ciortuz@dfki.de.

Abstract. We present

- the major enhancements added to the AM for unification of OSF-terms [2] so to be able to use it (as desired) for both typed (à la [7]) and non-typed feature structures, and
- essential ideas in the design of the control level (above unification) added to this AM to do efficient head-corner bottom-up chart-based parsing [15] [27] on large unification grammars, in particular on LinGO [11], the wide coverage HPSG grammar for English developed at CSLI, University of Stanford.
- In conclusion of this work, we propose the definition of a simple logic language called Light, in the framework of CLP(OSF), that underlies LinGO(-like) HPSG grammars. Deduction in Light is a head-corner version of bottom-up parsing-oriented deduction [26].

1 Introduction

Aït-Kaci, Podelski and Goldstein have stated in [5] that OSF-theory unification is more general not only than OSF-term ($\psi - term$) unification but also than typed feature structure unification [7].

The Light AM described here stands for OSF-theory unification in order- and type-consistent OSF-theories. This class of OSF-theories, formally defined in the beginning of Section 2 is slightly larger than that of systems of typed feature structures, whose definition is assumed here in the form synthesized by [10] and restated in Appendix. Building on the OSF AM design presented in [2], we show what constructions have exactly to be added in order to perform typed feature structure unification. A number of AMs for typed unification were developed in the past ([9], $\mathcal{A}$MALIA [29] and LiLFeS [20]) but all of them have been derived from WAM, and did not elaborated on (the specific difference wrt) the OSF AM. The relation between all these abstract machines is given in Figure 1.

Light AM was subsequently enhanced with a control level for head-corner deductive parsing [15] [27], thus generalizing the chart-based parsing-oriented control level in $\mathcal{A}$MALIA and putting under compiled form the hyper-active parsing idea [22].

The compiler system resulted was tuned and tested on a large-scale HPSG [25] grammar for English, namely LinGO [11]. Good performance numbers advocate for the continuation of this work, namely incorporation of the (compiled form of) quick-check [19] and parsing with packed feature structures [22].

When we enhanced the OSF AM so to do parsing with LinGO, we translated this grammar (or even more general: the conjunctive positive subset of TDL [17]) into a logic language called Light,[1] a CLP(OSF) language in which the set of predicate symbols is empty.[2]

In this paper, that can be thought as concentrating on some basic issue of the Light's operational semantics, we will show that upgrading the OSF AM so to be able to perform OSF-theory unification was not difficult; moreover, the original setup of the OSF AM suggested interesting optimizations for

[1] Light stands for LIGHT — Logic, Inheritance, Grammars, Heads, and Types. The analogy with the name of LIFE — Logic, Inheritance, Functions and Equalities — a language based on the OSF constraint system [4] is evident.

[2] For the CLP schema see [14].

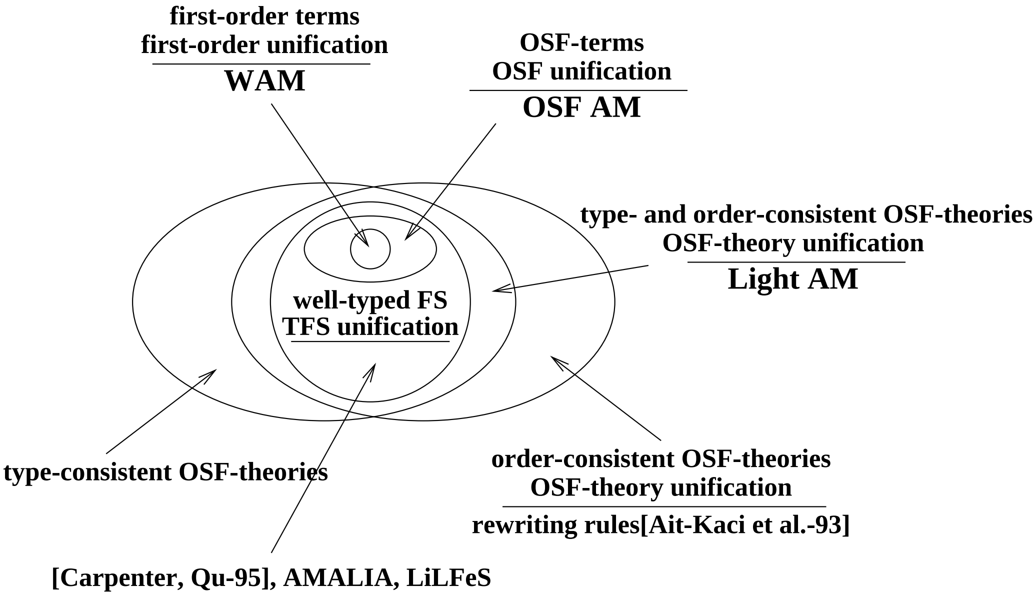

Fig. 1. The relation between Light AM and other AMs for compiled FS unification.

our system, that proved to be very effective, compared with a machine designed especially for typed feature structures like LiLFeS.[3] On the CSLI test suite (derived from the Hewlett-Packard test suite), we obtained the following measurements, given in seconds/sentence:[4]

system	*optimization*	*frequency*	*RAM*	*average parsing time*
PET	quick-check	400MHz	4GB	0.04
Light AM	without quick-check	400MHz	4GB	0.06
PET	without quick-check	400MHz	4GB	0.09
LiLFeS	quick-check	336 MHz	6GB	0.11

It might be surprising that an AM not intended initially for unification of typed feature structures behaves (so) well on large typed grammars. Our explanation is that *i.* partiality, in the *weak* well-typed form of LinGO [5] goes well together with the OSF/Light AM's capability to deal with open spare

[3] LiLFeS [20] is the only other existing compiler that was tested on LinGO. It is based on an AM [18] that enhanced the AM devised previously for unification of typed feature structures [9].

[4] The PET system is reportedly the fastest one running LinGO [24]. While using the same translation of LinGO into an OSF-theory, and incorporating certain of the techniques developed for LKB and Light as acknowledged in [6], PET's development status is more advanced than Light simply because it is an interpreter and normally the development of compilation techniques require considerably more time than an interpreter implementation. The good performance of PET wrt our system (only 50% slower when quick-checking is turned off) is explained by the integration of the Tomabechi's unifier [28]. The main idea from Tomabechi's unification algorithm concerns delaying "write" (stream) operations until unification is known to succeed; if unification fails, then those write operations are cancelled because they became useless. An optimization (already designed) for Light AM will be implemented in the near future, following this idea.

The *quick check* [19] is a technique for early detection of feature structure unification failure. We elaborated the design of *compiled quick check*, and its implementation will be completed in the near future.

The above measurements for PET were obtained without taking into account packing of feature structures [23]. U. Callmeier provided the numbers for PET; the LiLFeS numbers were (to be) published in [20]. Unfortunately we haven't the possibility to run LiLFeS on our 400MHz 4GB Sun Sparc station; the estimated performance of LiLFeS on this machine for the CSLI test suite is 0.09 sec/sentence.

[5] The *weak* well-typed form is obtained from the totally well-typed form by relaxation through unfilling [13]

feature structures, and then *ii.* sharing of feature structures in the specialized compiled form of rules is highly beneficial,[6] overcoming the advantage of direct feature retrieval inside feature frames on which LiLFeS and its predecessors rely. More detailed:

- three design improvements are required in order to make the OSF AM capable of unifying well-typed feature structures. These changes will be presented in the second section of this paper; one of them refers to the run-time unification function *osf-unify*, the remaining ones concern two of the OSF AM's abstract instructions.
- the so-called *two-streams optimization* incorporated into the design of the OSF AM proved to be highly beneficial for speeding up the parsing via
  - a sensible reduction of the size of the abstract code into which the input grammar is translated through maximal appropriate value *READ*-relaxation, and
  - specialized compilation of (unification grammar) rules, that in fact defines the parsing-oriented control level upon unification of feature structures.

  The first of these optimizations will be presented at the end of Section 2, the second one is the subject of Section 3.

In the Appendix of this work we propose the definition of the simple constraint logic language Light, a subset of CLP(OSF), that underlies LinGO(-like) HPSG grammars. The semantics of Light is suitable for head-corner bottom-up char-based parsing-oriented deduction [26].

2 From OSF-unification to typed feature structure unification: What makes the difference?

The input (LinGO-like) grammar for our system is translated into the form of a (Light) OSF-theory,[7] and then it is *expanded*. Expansion, described in the Appendix, is a type inference technique ([7], chapter 6) that puts the input grammar into an equivalent order- and type-consistent form that, depending on input, is also well-typed.[8] During expansion, the appropriate constraints — appropriate feature domain and values — are computed for the input grammar.

An OSF-theory — or more simply a set of OSF-terms — $\{\Psi(s)\}_{s\in\mathcal{S}}$, where $<\mathcal{S}, \prec, \wedge>$ is a lower semi-lattice,[9] is *order-consistent* if $\Psi(s) \sqsubseteq \Psi(t)$ for any $s \preceq t$, where $\sqsubseteq$ denotes subsumption. An OSF-theory is *type-consistent* if for any non-atomic subterm ψ of a $\Psi(t)$, if the root sort of ψ is s, then $\psi \sqsubseteq \Psi(s)$.

The operations described in this section are related to the dynamic typing — achieved by *on-line type expansion*, a form of "filling" (the appropriate type constraints when needed), according to [7] terminology, as we used to call it — that ensures the soundness of well-typed unification.

During parsing/unification,

- leaf nodes of a feature structure may become framed, i.e, they can get associated features,
- non-leaf nodes may become more specific, i.e., their sorts can be replaced by some others, situated deeper in the sort hierarchy, and therefore they had to satisfy more elaborated constraints, characterizing the new sort.

(see Appendix). The totally typed version of LinGO's translation into C had more than 33.2MB as source code (compiled: 72MB) while the weak typed version is 14.4MB (compiled: 27.6MB, reduced to 26.6MB by *maximal appropriate value READ-relaxation*, presented at the end of Section 2).

[6] Specialized compilation of HPSG rules in LinGO brought our system's parsing performance from 0.27 seconds per item for the CSLI test suite down to 0.10 (without quick check). As shown in the above figures, this result was subsequently improved even more.

[7] Work done by the precompiler implemented by U. Callmeier, starting from our specifications.

[8] This form is minimal in certain conditions, that limit type recursion to leaf nodes. For a detailed discussion on these issues see the Appendix.

[9] That implies that, for any $s, s' \in \mathcal{S}$ there is a unique glb(s, s') in $\mathcal{S}$. This glb is denoted $s \wedge s'$.

This is the reason why the *osf-unify* procedure (in fact only the *bind-refine* function called by *osf-unify*) and the *intersect-sort* and *test-feature* abstract instructions in the design of the AM for unification of OSF-terms [2] are enhanced as described in the sequel.

Note:

> Due to the reduced length of this paper, some familiarity with the AM data structures as presented in [2] is assumed. However, in order to make the presentation easier, Figure 2 illustrates what a feature structure like that in Figure 10 looks like when represented on the OSF/Light AM's basic structures.
>
> There are two stacks — one of *cells*, the other of feature *frames*. The first one, most important, will be usually called simply the AM's heap. Each cell stores information about a node in a feature structure: the sort (index in the symbol table) – SORT, the feature frame (index in the frame heap) – FTAB, and the coreference – CREF, containing the index of a heap cell. Implicitly, if a heap cell c has the address/index value i in the heap, then the CREF field of c is i. If the feature structure whose root node is c gets instantiated/coreferenced to another, more specific feature structure, whose root node has the index value j, then the content CREF field of c becomes j. While not shown explicitly in Figure 2, a frame in the frame_heap has in fact two subcomponents: nf – the number of feature actually stored in that frame, and features – the address of a a chunk of feature-value pairs in a global array reserved for this purpose.
>
> The Light AM uses an optimized representation for non-empty (*cons*) lists and difference lists. Assuming that they (and all their descendants) have only two features — FIRST and REST, their values will be stored in the FTAB and CREF fields.[10] Also, the current implementation of Light AM assumes that the *nil* sort has no descendants.[11] Therefore a *nil*-sorted cell is built at the address 0 of the heap, and all built lists' end will normally point to this address. All unification algorithms implemented for Light AM are designed so not to affect the CREF filed of this "reserved" cell.[12]

The *bind-refine* function corresponds primarily to equation constraint propagation simply by affecting one heap cell's CREF field. In our version (Light AM) shown in Figure 3 it does two additional things:

- in view of further backtracking — and later, during parsing, for feature structure sharing — *bind-refine* stores in the machine's trail the information on the heap, affected through binding; this is exactly what the two push_TRAIL operations are for.
- *on-line-expansion* is applied if required by the conditions explained above and formalized in the expansionCondition function.

The *on-line-expansion* function is presented in Figure 4. This function has to propagate onto the feature structure recorded at the address addr on the heap the constraints associated with the type $\Psi(s)$, while checking for consistency. In the Light AM current implementation, this type is assumed to be compiled as a program term. Functions coding (program or query) terms are indexed; the index p of the program function corresponding to the type s is obtained by invoking the (simple) function program_id. The program function corresponding to the type s is applied to the feature structure recorded on the heap at the address addr by calling the (general-purpose) function program with the argument p.[13] Its effect

[10] U. Callmeier pointed out that this representation imposes some limitations on the list's sort signature, but they have practically no impact (on LinGO).

[11] This is a fair enough restriction that can be be imposed to the current version of LinGO, according to D. Flickinger (personal discussion).

[12] For simplicity reasons, all the pieces of code given in the sequel will be limited to the general representation of feature structures on the heap, ignoring the special cases raised by the optimized representation of non-empty lists and difference-lists.

[13] The program function performs the (compiled) unification of the $\Psi(\mathsf{s})$ term with the term found on the heap at the Q address.

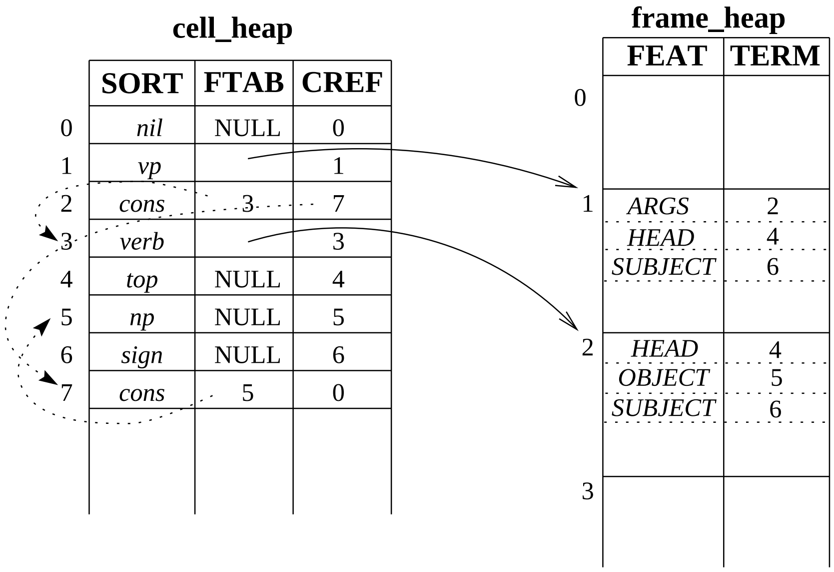

Fig. 2. The internal representation of the OSF-term in Figure 10.

```
boolean bind_refine( int d1, int d2, sort s )
{
    push_TRAIL( d1, LINK, heap[ d1 ].CREF );
    if heap[ d2 ].SORT ≠ s ∨
            (¬ isAtomicFS( d1 ) ∧ isAtomicFS( d2 )) {
        push_TRAIL( d2, SORT, heap[ d2 ].SORT );
        if OnLineExpansion ∧ expansionCondition( d1, d2, s ) {
            heap[ d2 ].SORT = s;
            return on_line_expansion( s, d2 ); }
        else
            { heap[ d2 ].SORT = s; return TRUE; }  }
    else { heap[ d2 ].SORT = s; return TRUE; }
}

boolean expansionCondition( int d1, int d2, sort s )
{
    if  ((¬ isAtomicFS( d1 ) ∨ ¬ isAtomicFS( d2 )) ∧
            heap[ d1 ].SORT ≠ s ∧ heap[ d2 ].SORT ≠ s) ∨
        (isAtomicFS( d1 ) ∧ ¬ isAtomicFS( d2 ) ∧ heap[ d2 ].SORT ≠ s) ∨
        (¬ isAtomicFS( d1 ) ∧ isAtomicFS( d2 ) ∧ heap[ d1 ].SORT ≠ s)
        return TRUE;
    else return FALSE;
}
```

Fig. 3. The new version of the bind_refine procedure.

```
boolean on_line_expansion( sort s, int addr )
{
    boolean result;
    int p = program_id( s ), oldQ = Q;

    Q = addr;
    saveXregisters();
    push( &programPDL, p );
    result = program( p );
    pop( &programPDL );
    restoreXregisters();
    Q = oldQ;

    return result;
}
```

Fig. 4. The on_line_expansion function.

is (in this particular case) exactly the unification of the feature structure rooted at addr with the type $\Psi(s)$.

What remains is to explain the actions that have to be done prior to, and respectively after the application of the program function:

- the Q register – that keeps the address of the feature structure against which unification is (to be) performed – is saved, and respectively, later is restored to the value it had prior to on-line expansion;
- the values of the X registers (in fact, in the Light AM implementation, only those different from the implicit values) are saved/restored;
- the program index p is pushed onto the programPDL stack before on-line expansion, and respectively popped out afterwards; the top element in this stack is checked by the abstract instructions *intersect-sort* and *test-feature* to avoid looping (that otherwise can occur) when applying the program function corresponding to the sort s.

Example 1. Let us consider the ψ-terms

$$\psi_1 = \text{a}[\ \mathsf{FEAT1} \quad \text{b}\ ],$$
$$\psi_2 = \text{a}[\ \mathsf{FEAT1} \quad \text{c}[\ \mathsf{FEAT2} \quad \mathit{bool}\ ]\ ],$$

while in the underlying OSF-theory, $b \wedge c = d$, the symbol $+$ is subsort of *bool*, and

$$\Psi(d) = \text{d}[\ \mathsf{FEAT2} \quad +\ ].$$

The glb OSF-unifier of ψ_1 and ψ_2 is

$$\psi_3 = \text{a}[\ \mathsf{FEAT1} \quad \text{d}[\ \mathsf{FEAT2} \quad \mathit{bool}\ ]\ ],$$

while the ($\Psi(d)$) OSF-theory unifier is

$$\psi_4 = \text{a}[\ \mathsf{FEAT1} \quad \text{d}[\ \mathsf{FEAT2} \quad +\ ]\ ].$$

It is obvious that, if ψ_1 and ψ_2 are represented on the heap, the *osf-unify* procedure using the new form of bind_refine, obtains as result ψ_4, by invoking on_line_expansion on the ψ_2.FEAT term, namely d[FEAT2 *bool*]. As soon as the root sort of this term was refined to d, it is transformed into d[FEAT2 +] by unification with $\Psi(d)$.

The understanding of (the new form of) the *intersect-sort* abstract instruction,[14] presented in Figure 5, is quite easy now, due to the explanations given above, that apply also here. The only thing to mention

[14] The *intersect-sort* abstract instruction is in charge with refining the sort of a heap cell, by computing the glb of its current sort with a given sort s.

is that prior the application of on-line expansion, the newly computed sort is already stored in the addr cell on the heap, in order to avoid looping when applying on-line expansion. Obviously, if expansion fails, the old sort of the addr cell must be restored.

```
intersect_sort i:int, s:sort ≡
  {
    int addr = deref( X[ i ] ), p;
    sort old_sort = heap[ addr ].SORT;
    sort new_sort = glb( s, old_sort );

    if new_sort == ⊥
      fail = TRUE;
    else
      if old_sort ≠ new_sort {
        push_TRAIL( addr, SORT, heap[ addr ].SORT );
        if OnLineExpansion ∧ ¬ isAtomicFS( addr ) {
          heap[ addr ].SORT = new_sort;
          p = program_id( new_sort );
          if addr ≠ Q ∨ p ≠ top( &programPDL )
            fail = ¬on_line_ID_expansion( new_sort, addr );
          else heap[ d2 ].SORT = s; }
        else heap[ d2 ].SORT = s; }
      else ;
  }
```

Fig. 5. The enhanced intersect_sort abstract instruction.

```
test_feature i:int, feat:int, j:int, level:int, l:label, s:sort ≡
  {
    int addr = deref( X[ i ] ), p;
    int f = get_feature( addr, feat );

    if f ≠ FAIL
      X[ j ] = heap[ addr ].FTAB.features[ f ].VAL;
    else {
      new_sort = heap[ addr ].SORT;
      if OnLineExpansion ∧ new_sort ≠ s AND isAtomicFS( addr ) {
        p = program_id( new_sort );
        if addr ≠ Q ∨ p ≠ top( &programPDL ) {
          on_line_ID_expansion( new_sort, addr )
          f = get_feature( addr, feat );
          X[ j ] = heap[ addr ].FTAB.features[ f ].TERM; }
        else { D = level; goto W_label; } }
      else D = level; }
  }
```

Fig. 6. The enhanced test_feature abstract instruction.

Example 2. Let us consider $\psi_1, \psi_2, \Psi(d)$ and the sort signature like in Example 1. If ψ_2 is present on the heap and ψ_1 is compiled as a program term (like in Figure 7), then when executing the instruction intersect_sort $X1, b$, the computed new_sort will be d, which is different from b, therefore on_line_expansion will be applied, further adding the feature FEAT2, + at ψ_2.FEAT1.

```
                              R0:  intersect_sort X[0], a
                                   test_feature X[0], FEAT1, X[1], 1, W1, a
                                   intersect_sort X[1], b
                                   R1:  goto W2;
W0:  push_cell X[0]
     set_sort X[0], a
W1:  push_cell X[1]
     set_feature X[0], FEAT1, X[1]
     set_sort X[1], b
W2:
```

Fig. 7. Abstract program code for the term ψ_1 in the Example 2.

Concerning the (new form of the) *test-feature* abstract instruction — that basically has to test whether the feature structure rooted by the cell whose index is X_i has the feature feat defined (at root level), and if so, instantiate X_j to (the root address of) its value — we have to notice the introduction of the additional argument s (not present in the original, [2] version). It is needed to check whether on-line expansion must be applied: s is the sort value of the current/corresponding feature path for the type whose compiled code contains this test_feature instruction. If new_sort, the actual value of this feature is a more specific sort than s, then in view of adding further feature constraints at addr, on-line expansion must be applied.[15]

Example 3. Again, consider $\psi_1, \psi_2, \Psi(d)$ and the sort signature like in Examples 1 and 2. Now let ψ_1 be represented on the heap and ψ_2 be compiled as a program term (see Figure 8). The instruction test_feature $X1$, FEAT2, $X2$, 2, W2, c finds that the root sort of the substructure ψ_1.FEAT1 is d. (It replaced the value b, at the execution of the preceding instruction intersect_sort $X1$, c, that did not apply on_line_expansion since the ψ_1.FEAT1 substructure on the heap was not framed (yet)). Now, as $d \neq c$, on_line_expansion is invoked by the instruction test_feature, adding the feature FEAT2, + to the subterm ψ_1.FEAT1.

```
                              R0:  intersect_sort X[0], a
                                   test_feature X[0], FEAT1, X[1], 1, W1, a
                                   intersect_sort X[1], c
                                   test_feature X[1], FEAT2, X[2], 2, W2, c
                                   R1:  goto W3;
W0:  push_cell X[0]
     set_sort X[0], a
W1:  push_cell X[1]
     set_feature X[0], FEAT1, X[1]
     set_sort X[1], c
W2:  push_cell X[2]
     set_feature X[1], FEAT2, X[2]
     set_sort X[2], bool
W3:
```

Fig. 8. Abstract program code for the term ψ_2 in the Example 3.

At a first sight, our on_line_expansion function plays a similar role to Wintner's build_most_general_fs

[15] A better, less expensive work would be to apply at addr only the constraints associated with the value of the feature feat in the new_sort type. In the current status of development, the Light AM does not support this optimization. Nevertheless, it could be implemented easily enough using some technical ideas presented in [12].

function in $\mathcal{A}$MALIA, since they both achieve type checking through associated constraints filling.[16] However, the differences between build_most_general_fs and on_line_expansion are rather essential:

1. build_most_general_fs in $\mathcal{A}$MALIA is an *optimization* construct. If we eliminate it, we affect the system performances, but not its semantics/output. On the contrary, on_line_expansion is a *basic* construct for Light AM understood as an extension to the OSF AM that makes it capable of performing OSF-theory (and therefore type) unification. It may by no way be eliminated from the (well-typed) Light's operational semantics.

2. In $\mathcal{A}$MALIA, the type constraints' filling through build_most_general_fs always precedes unification on that (to-become-framed) feature structure. In Light, constraint propagation through on_line_expansion may be done even on an already framed feature structure, namely when its root sort becomes more refined during unification. Therefore the application conditions for the two functions differ in a non-trivial manner.

3. The main idea behind build_most_general_fs cannot be applied in the Light's more general setup. Filling of sub-type constraints can be simply delayed in $\mathcal{A}$MALIA because its input is ALE-like [8], allowing neither coreference/equality constraints on different feature paths, nor refining of type constraints in a substructure. The Light setup has no such restrictions, and therefore we have to apply all the constraints of the type $\Psi(s)$ to any (to-be) non-atomic substructure whose sort has been refined to s during unification. Memoization of all nodes that, when unfolded, could affect a certain node is impractical, since among these nodes are not only members of the compiled term that performs the current unification, but also other nodes (in the enveloping super-structure) whose unfolding/filling was priorly delayed.

4. While build_most_general_fs is able to atomize (and delay as much as possible) the construction of feature structures during unification, on_line_expansion on leaf nodes does simple filling, i.e., full type unfolding. On non-leaf nodes, filling is also complete if type constraints' checking succeeds.

To summarize, $\mathcal{A}$MALIA's build_most_general_fs strategy is: do lazy (breadth first) type construction for further feature constraint checking during unification, while Light AM's on_line_expansion strategy is: during unification do (depth-first) type checking for sort-refined framed (or, to-be framed) substructures.

Note: The lazyness of our *on_line_expansion* mechanism might be increased if the calls to on_line_expansion are stacked, and their execution is done only if unification succeeds.

A first and simple *optimization* added to the (compiler that generates code for) OSF AM when dealing with typed grammars is the *READ-relaxation of appropriate values*. If for a feature f, its maximal *appropriate* value is t, then in the abstract code of any subterm for which f is defined and takes the value t, the corresponding *READ*-stream instruction *intersect-sort* i, t can be deleted, since well-typedness ensures that if the term against which unification is performed has f defined, then its value is t or a subsort of t. This optimization applies to well-typed feature structures ([7], see Appendix), for which the maximal value for any feature f is given as $Approp(f,\ Intro(f))$.

For example, in LinGO, the type

```
png[ PN    pernum,
     GEN   gender ]
```

is coded as shown in Figure 9. The underlined intersect_sort instructions are eliminated, since maximal appropriate values for PN and GEN are respectively pernum and gender.

[16] Unfortunately, the neither definition of this function, nor the conditions under which it is applied were specified explicitly in [29], section 3.4.1 and [30], section 5.1.

```
                        R0:  intersect_sort X[0], png
                             test_feature X[0], PN, X[1], 1, W1, png
                             intersect_sort X[1], pernum
                        R1:  test_feature X[0], GEN, X[2], 1, W2, png
                             intersect_sort X[2], gender
                        R2:  goto W3;
W0:  push_cell X[0]
     set_sort X[0], png
W1:  push_cell X[1]
     set_feature X[0], PN, X[1]
     set_sort X[1], pernum
     write_test 1, R1
W2:  push_cell X[2]
     set_feature X[0, GEN, X[2]
     set_sort X[2], gender
W3:
```

Fig. 9. Example of relaxed code wrt maximal appropriate values for features.

3 Compilation of head-corner bottom-up chart-based parsing: Basic constructs

Specialized compilation design for rules must be done in such a way that their application be suitable and efficient for the active bottom-up chart-based head-corner parsing.[17] Compared to the general setup of compiled unification of feature structures, specialized 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 control operations.

Above them, and even more important is the fact that in Light AM there will be two possible *application modes* for (compiled) unification grammar rules:

- the *key mode:* unification of the rule's key argument with the feature structure corresponding to a *passive item*; the matching substitution obtained through unification (if it succeeds) is saved in a newly created *environment*;
- the *complete mode* (for non-unary) rules: restore the environments corresponding to the already parsed/instantiated arguments of the rule and unify one of the active arguments of the list with the feature structure corresponding to a *passive item*; if unification succeeds, then a new environment is created (as above), and if 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.

The switch between the two modes is done by examining the register K in the beginning of the rule's application. This "key" environment register will be set to 0 before applying a unary rule, and also before the application of a binary rule in key mode.

Initially, the lexical characterizations of the words in the input sentence are stored (as passive items) on the chart.

[17] Prior measurements have shown that active head-corner parsing is approximately twice as fast as passive parsing on LinGO. To differentiate the notion of head in HPSG from that used for head-corner parsing, we adopt the proposition 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 specialized 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. Generalization to rules of arbitrary length is not difficult.[18]

The technique we chose to obtain the specialized compiled form of rules (represented as feature structures) is program transformation. Starting from the abstract code delivered by OSF "program" compilation of the feature structure representing a rule, we will upgrade it with specialized control sequences for the rule's application. Thus, our specialized rule compilation task consists mainly in defining specialized *control instructions*, and simple *transformation operations* on abstract code, inserting the specific control sequences for the arguments' unification and the construction of the mother/LHS feature structure. Due to the limited length of this paper, we will concentrate on the specialized compilation by *program transformations* matter itself, and we will give (only) an intuitive description for newly required control abstract instructions.

Henceforth, in this section, when not otherwise explicitly stated, we will refer to binary rules, because their treatment is of course more elaborated. 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 [27]

$$\begin{aligned} &vp \rightarrow \star verb \;\; np \\ &\quad vp.\mathsf{HEAD} \doteq verb.\mathsf{HEAD} \\ &\quad vp.\mathsf{SUBJECT} \doteq verb.\mathsf{SUBJECT} \\ &\quad verb.\mathsf{OBJECT} \;:\; np. \end{aligned} \tag{2}$$

one can associate the feature structure in the Figure 10. (The *verb*, marked with $\star$, is the key argument of the rule. The symbol # introduces coreferences. Lists are presented in sugared notation < ... >.)

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

Fig. 10. The OSF-term associated to the rule (2)

.

Among all the equivalent OSF-terms/feature structures that can describe the rule (2), we have chosen the one above 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*:

[18] Our system can however deal with arbitrary long rules, in a version that compiles rules as ordinary feature structures.

– 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.

Note that the first condition stated above ensures the top-to-down partitioning of the areas $ARG1$, $ARG2$, and LHS, 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.

```
                                         R0:  intersect_sort X[0], vp
                                              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]
                                              unify_feature X[0], SUBJECT, X[5]
                                         R5:  jump W8
W0:   push_cell X[0]
      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
      set_feature X[0], HEAD, X[3]
      set_feature X[0], SUBJECT, X[5]
W8:
```

Fig. 11. The OSF abstract code for the rule (2)

.

Convention: In order to simplify the general presentation of our specialized compilation technique, we will assume first that, like in the above example, for any rule feature structure, the key argument is the first one in the rule's *ARGS* list. Later on, we will show how we deal in practice, when this convention is no longer assumed.

The OSF "program" code that implements (unification for parsing with) the feature structure given in Figure 10 is shown in Figure 11. It is easy to identify our list-specialized instructions that deal with the *ARGS* feature: *test_inst_list, get_first, get_rest, unify_first, test_NIL_rest, set_list*, and additionally *set_sort* and *intersect_sort*, when the second argument is *cons*. It is exactly at the "slots" formed by these instructions that control sequences for parsing with this rule will be placed.

Technically, new abstract instructions — *saveEnv* and *restoreEnv* will be used at/by the control sequences placed (via meta-interpretation) at the slot places $S1, ..., S6$. An *environment* is a couple of *i*. a set of coreference indices, and *ii*. a trail copy (that registers the changes done on the heap during unification).[19]

The new specialized form of the rule (2), obtained by transformation of the code in Figure 11 is given in Figure 12. There, the control instructions are indented to the left, to be clearly visualized. Apart from the fact that they replaced ARGS list-oriented stuff at the control slots shown in Figure 10, four 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.
- 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 `W0` and `W1` was moved after the "write" code for $ARG2$ (actually empty).[20]
- the *ARGS* feature is "discarded" from, i.e., not created in the *LHS* code. This omission, perfectly reasonable for practical reasons, is supported by the Locality Principle in HPSG, and will be adopted in the current approach.

Note: The example presented above illustrates what a binary rule looks under specialized compiled form. It represents however, a particular case of binary rule, namely one in which the second argument is previously corefered, i.e, its index/tag occurs in the $ARG1$ code area. The general case requires a few additional refinements that will be skiped here.

As an *example* of rule application, when the system will parse the input sentence *"The cat catches a mouse"*, also taken from [27], 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 K is set to 0, and Q is instantiated to the root address (on the heap) for the feature structure characterizing the verb *catches*.

• When entering the *vp* rule code, the set of coreference indices (corefs) is instanciated to $\{3, 4, 5\}$, since `X[3], X[4], X[5]` in the abstract code correspond to `#1, #3, #2` respectively in Figure 10.

• Then, as K is 0, the sequence $ARG1$ of instructions — namely from the label `R0` to `R2` and `W3` to `W6` — is executed, leading to unification of the *verb*-rooted feature structure in Figure 10 with the term corresponding to the lexical entry *catches*.

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

[20] This code was dead in the design of the OSF abstract machine; it has to be "revived" in this new, specialized approach for rules compilation.

```
vp:   set corefs, { 3, 4, 5 }
      cond K != 0, jump R3

                                   R0: % ARG1
                                      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
                                   R3: % ARG2
                                      restoreEnv K
                                         cond unify( X[4], Q ) = FALSE , Failure
                                   R5:jump W0
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
    jump W8
    % ARG2
W0: % LHS
    saveEnv NULL
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. 12. Specialized (Light) abstract code for the rule (2)

.

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

- the coreferences: the values of `X[3], X[4], X[5]`, namely the indeces of the heap cells rooting the feature structures `#1, #3, #2` in Figure 10, 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, to be 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 (not null) of the above created environment is stored in the register K, and the content of the register `Q` is set to the heap address of the feature structure characterizing the noun phrase *a mouse*.

- Now, reentering the *vp* rule, $\mathsf{K} \neq 0$ dictates the execution of the $ARG2$ sequence (`R3-R5`). At `R3`, the environment of index K 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 10, and the `verb` argument in the same figure is instanciated 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 10), is unified with the feature structure characterizing the noun phrase *a mouse*.

• As this last unification succeds, a new environment is created (with an empty set of coreferences), corresponding to the instation of `#4:np` in Figure 10 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.

The *key&complete* mechanism of (binary) rule application in our implementation of Light is optimized through delayed environment saving, that eliminates the need for environment restoring in certain cases.

Comming back to discuss the *convention* we made above, assuming that the key argument of a rule is the first one in the *ARGS* list, 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., corefering the elements) and then moving the key argument on the first position. The above specialized 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.

Note: In our implementation, the program transformations presented in this section are done "on fly", i.e., for any rule the specialized, "ready for parsing" form is the only one to be delivered by the (Light into C) compiler, if specialized compilation was demanded.

4 Contribution

We extended the design of the AM for unification of (non-typed) OSF-terms so that

– it can be used to perform unification of typed feature structures;[21] in this view, our machine can be seen as inheriting/generalizing from both [2] and [30], the later one being dedicated for typed feature structures.
– unification in our machine is "controled" by a parsing-oriented level (that corresponds to the SLD-resolution level in the classical WAM [1]); in this respect our strategy — head-corner active bottom-up chart-based parsing — is more elaborated/general than the simple bottom-up chart-based parsing in AMALIA [30], making the performances of our system substantially better those of the LiLFeS compiler [20]. The compiled (specialized) form of rules is obtained via transformation of the code generated (with the "two-streams" optimization) by the OSF AM for rules represented as feature structures.

Acknowledgements: Thanks go to Shuly Wintner for observations on several draft versions of this paper. (Of course, the remaining faults are only mine.) Ulrich Callmeier contributed to the debugging work on CHIC, the development prototype of ABC Light.

An enhanced version of this paper — elaborating notably on parsing issues in Light AM, since they could not be fully developed here — will be made available as a DFKI research report.

[21] Switching between the two "regimes" — typing and respectively non-typing — makes the new machine useful for both kinds of tasks. In particular, we use it in "non-typing" regime during compilation, unfolding and type inference for the input (non-typed) grammars, and in "typing" regime while parsing with typed grammars.

References

1. H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* The MIT Press, Cambridge, MA, 1991.
2. 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/.
3. H. Aït-Kaci and R. Nasr. Login: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.
4. H. Aït-Kaci and A. Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16:195–234, 1993.
5. H. Aït-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. The MIT Press.
6. U. Callmeier. PET — a platform for experimentation with efficient hpsg processing techniques. *Journal of Natural Language Engineering, Special Issue on Efficient Processing with HPSG: Methods, Systems, Evaluation*, pages 99–108, 2000 (to appear).
7. B. Carpenter. *The Logic of Typed Feature Structures – with applications to unification grammars, logic programs and constraint resolution*. Cambridge University Press, 1992.
8. B. Carpenter and G. Penn. ALE: The Attribute Logic Engine. User's Guide. Technical report, Carnegie-Mellon University. Philosophy Department. Laboratory for Computational Linguistics, Pittsburgh, 1992.
9. B. Carpenter and Y. Qu. An abstract machine for attribute-value logic. In *Proceedings of the Fourth International Workshop on Parsing Technologies*, pages 59–70, 1995.
10. A. Copestake. Definitions of typed feature structures. *Journal of Natural Language Engineering, Special Issue on Efficient Processing with HPSG: Methods, Systems, Evaluation*, 2000 (to appear).
11. A. Copestake, D. Flickinger, and I. Sag. *A Grammar of English in HPSG: Design and Implementations.* Stanford: CSLI Publications, 1999.
12. D. Diaz and P. Codognet. WAMCC: Compiling Prplog to C. In *Proceedings of ICLP'95, the 12th International Conference on Logic Programming*, Tokyo, Japan, 1995. The MIT Press.
13. D. Gedermann. Term encoding of typed feature structures. In *Proceedings of the 4th International Workshop on Parsing Technologies*, pages 89–97, Prague, Czech Republik, 1995.
14. J. Jaffar and M. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19(20):503–582, May-July 1994.
15. M. Kay. Head driven parsing. In *Proceedings of Workshop on Parsing Technologies*, Pittsburg, 1989.
16. 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*, 1999.
17. 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.
18. T. Makino, K. Torisawa, and J. Tsujii. LiLFeS – practical unification-based programming system for typed feature structures. In *Proceedings of Natural Language Peocessing Pacific Rim 1997 (NLPRS'97)*, Phuket, Thailand, 1997.
19. R. Malouf, J. Carroll, and A. Copestake. Efficient feature structure operations without compilation. *Journal of Natural Language Engineering, Special Issue on Efficient Processing with HPSG: Methods, Systems, Evaluation*, 2000 (to appear).
20. 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, Special Issue on Efficient Processing with HPSG: Methods, Systems, Evaluation*, 2000 (to appear).
21. St. Mueller. The Babel system – An HPSG fragment for German, a parser and a dialog component. In *Proceedings of the Fourth International Conference on the Practical Application of Prolog*, pages 263–277, London, 1996.
22. S. Oepen and J. Caroll. Parser engineering and performance profiling. *Journal of Natural Language Engineering, Special Issue on Efficient Processing with HPSG: Methods, Systems, Evaluation*, pages 81–97, 2000 (to appear).
23. S. Oepen and J. Carroll. Ambiguity packing in hpsg — practical results. In *Proceedings of NAACL 2000*, to appear.
24. 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*, 2000 (to appear).
25. C. Pollard and I. Sag. *Head-driven Phrase Structure Grammar.* Center for the Study of Language and Information, Stanford, 1994.

26. S.M. Shieber, Y. Schabes, and F. Pereira. Principles and implementation of deductive parsing. *Jornal of Logic Programming*, pages 3–36, 1995.
27. N. Sikkel. *Parsing Schemata*. Springer Verlag, 1997.
28. H. Tomabechi. Quasi-destructive graph unification. In *Proceedings of the 29th meeting of the Association for Computational Linguistics*, pages 315–322, Berkeley, 1991.
29. S. Wintner. *An Abstract Machine for Unification Grammars.* PhD thesis, Technion – Israel Institute of Technology, 32000 Haifa, Israel, 1997.
30. S. Wintner and N. Francez. Efficient implementation of unification-based grammars. *Journal of Language and Computation*, 1(1):53–92, 1999.

5 Appendix: Light — a significant subset of CLP(OSF)

Abstract. Light is a simple subset of CLP(OSF) particularly interesting for NLP — it notably underlies LinGO, a large coverage HPSG grammar of English designed at Stanford University [11]. Starting from the AM for unification of OSF-terms [2], we implemented of a full-fledged system that compiles (and does efficient parsing with) Light logic grammars.

LIGHT — in the sequel written simply Light — is an abbreviation for Logic, Inheritance, Grammar, Heads, and Types. The similarity to the LIFE abbreviation (Logic, Inheritance, Functions and Equations) is in itself suggestive. Life is a successor of Login (Logic and Inheritance [3]), that led to the development of the logical feature constraint OSF theory, but while Life maintained relational constraints on terms like in Prolog and got a flavor of concurrence, the new, simpler language Light aims to build the bridge towards NLP/CL, first by picking up from CLP(OSF) only what's needed for (efficient) parsing with large scale type unification grammars, and then importing certain features from the existing (interpreter-like) systems implementing such grammars.

There are three main characteristics that individualize Light as a subset of CLP(OSF):

1. It doesn't use predicate symbols at all, simply because the selection of feature structures involved in parsing is done primarily on feature structures' root sort (and then the search is refined via checking of feature constraints).
2. Deduction in Light is done in a bottom-up manner, since rules used in heavily lexicalized grammars like HPSG [25] are usually left top-sorted,[22] and therefore the selection of possible candidates for unification is based mainly on feature constraint checking. We extended the AM defined by Aït-Kaci and Di Cosmo to a full-fledged AM-based compiler, in which unification is "controlled" by a strategy known in the NLP/CL literature as **a**ctive **b**ottom-up **c**hart-based head-corner parsing [27], [15]. This is why, when refering to our current implementation of Light we will call it ABC Light.
3. We consider that Light fills in a natural way the gap between OSF logic theory setup and the theory of (linguistic) types [7] as used in the HPSG theory.

5.1 Formal introduction to Light

Constraint logic definitions:

– In the sequel, the notions of sort constraint, feature constraint and equality constraint, OSF clause, normalized OSF clause (or ψ-term) over a signature $\Sigma =< \mathcal{S}, \mathcal{F}, \prec>$, matching/subsumption (denoted: $\sqsubseteq$), unification and greatest lower bound (glb) of two ψ-terms are like in the OSF theory [5]. Notations: *root*(ψ) and $\psi.f$ denote the sort of the root node in ψ, and respectively the value of the feature f at the root level in ψ. Also, $\preceq$ is the reflexive and transitive closure of $\prec$.

[22] Or, *sign*-sorted.

– An OSF-*theory* is a set of types $\{\Psi(s)\}_{s\in\mathcal{S}}$ such that $root(\Psi(s)) = s$. An *order-consistent* OSF-theory is one in which the following (*monotonicity*) condition is satisfied: $\Psi(s) \sqsubseteq \Psi(t)$ for any $s \preceq t$.
In the sequel, features will be understood as partially defined functions (instead of totally defined functions as in the OSF feature constraint system).
– An well-typed OSF-theory is an order-consistent theory in which the following conditions are satisfied for any $s, t \in \mathcal{S}$:

  i. if $f \in$ *Arity(s)* $\wedge f \in$ *Arity(t)*, then $f \in$ *Arity*$(s \wedge t)$;
  ii. for every subterm ψ in $\Psi(t)$, such that $root(\psi) = s$,
  if a feature f is defined for ψ, then $f \in$ *Arity*(s), and $\psi.f \sqsubseteq \Psi(s.f)$.

where *Arity(s)* is the set of features defined at the root level in the term $\Psi(s)$.
An OSF-term ψ satisfying the condition *ii* from above is (said) *well-typed* wrt the OSF theory $\{\Psi(s)\}_{s\in\mathcal{S}}$.

Notes:
1. The condition *i* implies that for every $f \in \mathcal{F}$ there is at most one sort s such that f is defined for s but undefined for all its supersorts. This sort will be denoted *Intro*(f), and will be called the *appropriate domain* on the feature f. Also, $root(\Psi(s), f)$, if defined, will be denoted *Approp*(f, s), and will be called the *appropriate value* on the feature f for the sort s. *Approp*$(f$, *Intro*$(f))$ is the maximal appropriate value for f. [23]
2. As a well-typed OSF-theory is an order-consistent one, it implies that *Arity(s)* $\supseteq$ *Arity(t)*, and *Approp(s, f)* $\preceq$ *Approp(t, f)* for every $s \preceq t$;
3. A stronger version for the condition *ii* would be: the feature f is defined (at the root level) for ψ iff $f \in$ *Arity*(s), and $\psi.f \sqsubseteq \Psi(s.f)$. In this case, the theory is said to be *totally well-typed*. Obviously, a totally well-typed OSF-theory is type-consistent (see the definition of type-consistency given in Section 2.)

– The notion of OSF-unification extends naturally to *well-typed* OSF-*unification* in an OSF theory $\{\Psi(s)\}_{s\in\mathcal{S}}$ which is well-typed. The well-typed glb of two feature structures ψ_1 and ψ_2 is the most general (wrt $\sqsubseteq$) well-typed feature structure subsumed by both ψ_1 and ψ_2. The well-typed glb of two feature structures is subsumed by the glb of those feature structures. Note that the role of the *on-line-expansion* function defined in the Section 2 of this paper is ensuring that the glb computed by *osf-unify* for two (well-typed) feature structures is refined/instantiated to a well-typed feature structure (by enforcing/checking the condition *ii*).
– A Light logic grammar will be defined as a particular well-typed OSF-theory: first, there will be used several "reserved" sorts in $\mathcal{S}$ (namely *top, sign, rule-sign, lexical-sign* and *start* — a descendent of *rule-sign*), and features (PHON, ARGS). Light Horn clause: ψ_0 :– $\psi_1\ \psi_2\ ...\ \psi_n$ $(n \geq 0)$, corresponding to the logical formula $\tilde{\forall}(\psi_0 \leftarrow \psi_1 \wedge \psi_2 ... \psi_n)$, where $\psi_i, i = 0, 1, 2, ..., n$ are ψ-terms, and $root(\psi_0) \preceq$ *rule-sign*. (Remark the lack of predicate symbols.) Light rule: a Light Horn clause with $root(\psi_0)$ a leaf node in the sort hierarchy $(\mathcal{S}, \prec)$, and all (LHS and RHS) constituent terms having the feature PHON defined.[24] (PHON is related to the consumed input sentence).
– The logical form associated to a term:
$\psi \equiv s[f_1 \rightarrow \psi_1, ..., f_n \rightarrow \psi_n] \Rightarrow Form(\psi) \equiv \forall X \exists X_1 ... \exists X_n ((X.f_1 \doteq X_1 \wedge ... \wedge X.f_n \doteq X_n) \leftarrow X : s)$. For terms representing rules: ψ :– $\psi_1 ... \psi_n$. $\Rightarrow \forall X \exists X_1 ... \exists X_n ((X.\text{ARGS} \doteq < X_1, ..., X_n > \wedge Form(\psi)) \leftarrow (X : s \wedge Form(\psi_1) \wedge ... \wedge Form(\psi_n)))$, where $X_1, ..., X_n$ identify the roots of $\psi_1 ... \psi_n$ respectively.

Remarks:

1. Any Light rule can be written as a single ψ-term, if we denote the right hand side (RHS) arguments as a list value of the (reserved) feature ARGS. Anywhere we refer to a rule as an OSF-term, we assume this understanding. In the sequel we make the perfectly reasonable assumption that a Light logic

[23] ABC Light uses a weaker version for the condition *i*: if $f \in$ *Arity(s)* $\wedge f \in$ *Arity(t)*, and $f \notin$ *Arity*$(s \wedge t)$, then $root(s.f) = root(t.f)$.
[24] For the moment, the Light language is not designed to deal with so-called ϵ-rules, i.e., whose RHS is empty.

grammar $\mathcal{G}$ contains for every sort s at most one clause whose head (left hand side) is s-rooted/sorted. It will be called the *s type* in $\mathcal{G}$, and will be denoted $\Psi(s)$.

The ABC Light system does the transformation of the input Light grammar into a minimally equivalent well-typed one, using a transformation schema called *grammar expansion*.

2. In ABC Light and in all other LinGO-processing existing systems, after each rule application, the argument list of the newly resulted LHS/mother features structure is cut off; this technique presented in [16] is called feature structure *reduction*. The track of the mother-daughter relation established during parsing is kept however in the *chart*, a basic structure of the parser.

Parsing-oriented derivation specific definitions:[25]

- *Generalized instance* of a rule in the Light grammar $\mathcal{G}$:
  Let $\psi_0\ :-\ \psi_1\ \psi_2\ ...\ \psi_n$ $(n \geq 1)$ be a clause in $\mathcal{G}$, and $\{i_1, i_2, ..., i_k\} \subseteq \{1, 2, ..., n\}$. If σ is an OSF-substitution over Σ,[26] then $\sigma\psi_0\ :-\ \phi_1\ \phi_2\ ...\ \phi_n$, where
  $$\phi_j = \begin{cases} \sigma\psi_j, & \text{if } j \in \{i_1, i_2, ..., i_k\}, \\ \psi_j, & \text{otherwise.} \end{cases}$$
  is a generalized instance of that given rule. If $\{i_1, i_2, ..., i_k\} = \{1, 2, ..., n\}$, then we call it a *passive* instance; otherwise it is an *active* instance.
- A *parse item* in $\mathcal{G}$: a generalized instance of a rule in $\mathcal{G}$, or a matching substitution of (i.e., term subsumed by) a *lexical entry* type (a Light clause whose root is a descendent of the *lexical-sign* sort but has no descendent sorts) whose variables were priorly renamed apart.
- A generalized instance, denoted as $(\sigma\ ;\ \psi_0 :- \psi_1\ \psi_2\ ...\ \psi_n\ ;\ \{i_1, i_2, ..., i_k\})$ can *combine* with the OSF-term ψ iff for a $j \in \{i_1, i_2, ..., i_k\}$ there is a glb OSF-unifier of ψ and ψ_j, with τ the matching OSF-substitution, $\tau\psi_j = \text{glb}(\psi, \psi_j)$. The result of the combination is the generalized instance
  $$(\tau\sigma\ ;\ \psi_0 :- \psi_1\ \psi_2\ ...\ \psi_n\ ;\ \{i_1, i_2, ..., i_k\}\backslash\{j\}).$$
  Note: For bottom-up chart-based parsing, [26] and [27] propose two derivation rules: *scan* and *complete*. Head-corner parsing [27] distinguishes between *left-* and *right-* scan and respectively complete, and adds a new derivation rule, *head-corner*. It is worth to note that in a (order-)sorted environment, the distinction between terminal and non-terminal symbols is deleted, and therefore scan and complete become identical. In conclusion, in the Light setup, head-corner bottom-up chart-based parsing is achieved via two deduction rules: head-corner and (left- and right-) complete.
- If $w =< w_1, w_2, ..., w_l >$ is a finite sequence of symbols (from $\mathcal{S}$), then a lexical term ψ *recognizes* the input word w_i if $\psi.\text{PHON} \doteq w_i$. By recursive definition, a generalized rule instance defined as above recognizes the union of input sequences (of words) recognized by the arguments $\psi_{i_1}, \psi_{i_2}, ..., \psi_{i_k}$.
  A generalized rule instance is *admissible* wrt $w =< w_1, w_2, ..., w_l >$ if the input sequences recognized by $\psi_j, j \in \{i_1, i_2, ..., i_k\}$ are disjunct. In the sequel we will consider only admissible generalized rule instances.
- An *parse sequence* (or derivation): a finite sequence of *parsing items* $i_1, i_2, ..., i_m$ so that for every r with $1 \leq r \leq m$, the item i_r is either an admissible instance of a lexical entry type, or it is obtained by combining an active instance i_p of a rule with one passive item i_q, where $p, q < r$.
- Let $w =< w_1, w_2, ..., w_n >$ be a finite sequence of symbols (from $\mathcal{S}$). The Light logic grammar $\mathcal{G}$ *recognizes* the input sentence w iff there is a parse sequence (derivation) in $\mathcal{G}$ ending in a passive generalized rule instance *start*-sorted $(\sigma\ ;\ \psi_0 :- \psi_1\ \psi_2\ ...\ \psi_n\ ;\ \emptyset)$ that recognizes w.

Semantically, the fact that the (well-typed) grammar $\mathcal{G}$ recognizes the input sentence w means that there is a (well-typed) OSF-substitution σ and the lexical terms $\psi_1, \psi_2, ..., \psi_m$ such that (ψ_1.PHON, ψ_2.PHON, ..., ψ_m.PHON) constitute a partition of $\{w_1, w_2, ..., w_l\}$ and

[25] These definitions apply to both well-typed and non-typed Light logic grammars.

[26] Understood as a set/conjunction of constraints. Composition of substitutions in the sequel is meant as constraint conjunction.

$$\sigma\psi_1, \sigma\psi_2, ..., \sigma\psi_m \models \sigma\psi$$

or, equivalently,

$$\mathcal{G} \models_{\mathcal{M}} \tilde{\exists}(\psi_1 \wedge \psi_2 \wedge ... \wedge \psi_m \rightarrow \psi).$$

where $\mathcal{M}$ is the OSF-graph algebra [4].

In head-corner parsing, unification with a certain argument, designated by the grammar writer, is tried before the treatment of the other arguments. This "head" argument is the most important one, because it is usually critical for the application of the whole rule, i.e., statistically speaking, it is most probable to produce unification failure. On LinGO, tests have revealed that active chart-based head-corner active parsing is in average about twice as fast as passive (non head-corner) parsing.

Note: The above definitions do not ask for contiguity of (the recognized input sequences by) one rule's arguments. Like all the other LinGO-processing systems, ABC Light works with the contiguity assumption.[27]

5.2 Light grammar expansion

Inference of appropriateness constraints

The following simple schema is used in ABC Light to infer the appropriateness constraints (abbreviated: AC), i.e., the appropriate domain and value for every feature, for an input Light logic grammar $\mathcal{G}$:

For every feature f, the first[28] sort s that has the feature f defined at the root level in the s-sorted clause $\Psi(s)$ in the grammar $\mathcal{G}$ is the *appropriate domain* of f, and is denoted $Intro(f)$.[29] The propagation of appropriate domain constraints is done by replacing for every subterm ψ in the grammar $\mathcal{G}$ its root sort $root(\psi)$ with

$$\bigwedge_{f \text{ defined for } \psi} Intro(f)$$

Let $s_1, s_2, ..., s_n$ be the sorts in $\mathcal{S}$, topologically sorted according to $\prec$, i.e., if $s_i \prec s_{j_1}, ..., s_{j_k}$, then $i > j_1, ..., i > j_k$. Note that, without restricting the generality, s_1 can be assumed the most general sort (*top*) in $\mathcal{G}$.[30] Then, for $i = 1, 2, ..., n$, if $s_i \prec s_{j_1}, ..., s_{j_k}$, and $\Psi(s_1), ..., \Psi(s_n)$ are the clauses/types associated respectively to $s_1, ..., s_n$ in $\mathcal{G}$, then $\Psi(s_i)$ is replaced by $\Psi(s_i) \wedge \Psi(s_{j_1}) \wedge \Psi(s_{j_2}) \wedge ... \Psi(s_{j_k})$.[31]

$$\Psi(s_i) := \Psi(s_i) \wedge \Psi(s_{j_1}) \wedge \Psi(s_{j_2}) \wedge ... \Psi(s_{j_k}) \qquad (*)$$

[27] Other large systems like Babel [21], that implements a large HPSG grammar for German, are specialized in discontinuous phrase treatment.

[28] Here "first" is taken in the sense of top-down (and for instance depth-first) traversal of the sort hierarchy $(\mathcal{S}, \prec)$.

[29] ABC Light implementation does more: if several sorts $s_1, s_2, ..., s_n$ ($n \geq 2$) introduce the same feature f (i.e., f is defined for $\Psi(s_1), \Psi(s_2), ..., \Psi(s_n)$, but for any ancestor of $s_i, i = 1, 2, ..., n$, the feature f is not defined), then the system computes the minimal common ancestor for $s_1, s_2, ..., s_n$, and defines f for it, choosing as its value the minimal supersort for $\Psi(s_1.f), \Psi(s_2.f), ..., \Psi(s_n.f)$.

[30] In ABC Light we assume that there are no features associated to the *top* sort.

[31] From the procedural point of view, in the $(*)$ relation, the conjunction $\wedge$ can be replaced by OSF-unification, as presented in [5] and [2], i.e. without incorporating the on-line type expansion mechanism described in this paper. The ABC Light system achieves expansion phase with the OnLineExpansion register simply turned off.

If the is-a relation $\prec$ is not cyclic (and this fact is checked out by the ABC Light system), then the grammar unfolding terminates in finite time for any finite input.

Note that: 1. Conjunction/unification in $(*)$ may not always succeed. In such cases, either the grammarian corrects the grammar, or he/she let the (ABC Light) system itself soften the appropriate value for the feature that caused the sort unification clash (in this case, at the end, unfolding must be restarted). 2. If unification succeeds always in $(*)$, then the newly resulting system of types is order-consistent and verifies the appropriateness condition *i*. The condition *ii* will be accomplished by (off-line) type unfolding.

Finally, for $f \in \mathcal{F}$ and $s \in \mathcal{S}$, the *appropriate value* of f for s is (by definition) the root sort of the sub-term $\Psi(s).f$. The *maximal appropriate value* for f is *root*(*Intro*(f)).

Light type unfolding

Light type unfolding (or: off-line type expansion) proceeds in a manner which is to a certain extent similar to the AC inference schema presented above:

Let $s_1, s_2, ..., s_n$ be the sorts in the grammar's signature Σ, and $\Psi(s_1), \Psi(s_2), ..., \Psi(s_n)$ the corresponding clauses/types in the grammar. We define the relation $\trianglelefteq$ in the following, simple way: if ψ is a non-atomic[32] subterm of $\Psi(s_i)$, then $root(\psi) \trianglelefteq s_i$. If the relation $\trianglelefteq$ is acyclic, let $t_1, t_2, ..., t_n$ be the sorts $s_1, s_2, ..., s_n$ topologically sorted wrt $\trianglelefteq$. Then, type expansion for $\Psi(t_i), i = 1, 2, ..., n$ is achieved by replacing — in in-depth traversing order — every non-atomic subterm ψ of $\Psi(t_i)$ with $\psi \wedge \Psi(root(\psi)$.[33]

$$\psi := \psi \wedge \Psi(root(\psi) \qquad (**)$$

Again, some remarks: 1. unification in $(**)$ may fail; in this case the grammar writer has to revise the grammar; 2. if a subterm's root sort becomes more specific $(s \prec t)$ through partial unfolding $(**)$, then its consistency has to be checked, by further applying $(**)$ to that subterm, provided that the expansion of the type corresponding to its new sort (s) was already done;[34] [35] 3. if the $\trianglelefteq$ relation is not cyclic, then the type expansion for the input grammar $\mathcal{G}$ proceeds in finite time; 4. type unfolding as described above ensures for the expanded, newly obtained form of the grammar $\mathcal{G}$ the satisfaction of the type-consistency, implicit in the well-typedness condition *ii* (in its strong form). If the input form of $\mathcal{G}$ is so that for any subterm ψ in the grammar, if $root(\psi) = s$, then $Arity(\psi) \subseteq Arity(\Psi(s))$, then the well-typedness condition *ii* is fully satisfied, therefore the output is (totally) well-typed.

Then the ABC Light system softens the condition *ii* through *unfilling* [13]. The aim of AC-reduction is to get a smaller Light logic grammar that generates a given totally well-typed grammar $\mathcal{G}$. The idea of unfilling is very simple: for every type $\Psi(s)$ in the grammar, if it contains a feature constraint $\psi.f \doteq \psi'$, (ψ is a subterm of $\Psi(s)$), then this constraint can be eliminated from $\Psi(s)$ if

- ψ' is the atomic features structure $Approp(f, root(\psi))$, and
- if ψ is the type $\Psi(s)$ itself, then $s \neq Intro(f)$.

Unfilling for the LinGO grammar reduced the size of the binary image with a factor of 2.8; the resources consumed during parsing also lowered, and the gain in speed up was sensitive (30%).

On-line type expansion will complete type unfolding "by need", namely when a leaf node of a term gets framed (i.e. it is associated at least one feature) during the unification process.

[32] I.e., containing at least one feature.
[33] Here, $\wedge$ must be understood as OSF-theory unification.
[34] Note that at this point the relation $\trianglelefteq$ may be re(de)fined.
[35] As an example, consider again $\psi_1, \psi_2, \psi_3, \psi_4, \Psi(d)$ like in Examples 1, 2, 3, and let ψ_1, ψ_2 be represented on the heap, as subterms of two types, one already expanded (ψ_1), the other being under expansion now (ψ_2). OSF-unification (in its version osf_into_unification to be presented further) obtains ψ_3 from ψ_2. As the root sort of ψ_2.FEAT1 was refined to d, a new unification is launched: $\Psi(d)$ is unified (*into*) ψ_2.FEAT1, causing ψ_3 to be refined into the form of ψ_4.

For efficient grammar expansion, we proposed a new algorithm for OSF-unification, the *directed OSF-unifier* shown in Figure 13. This unifier doesn't affect its first operand. It affects only the second operand, which is assumed stored in an upper area of the heap, and finally represents the unification result.

The function *into_carry_features* (see Figure 13), called by *osf_into_unify* is a slightly enhanced version of the function *carry_features* (designed by Aït-Kaci and Di Cosmo), so to support directed unification.

The h border is a cell address/index, initially the root of the second operand. It has the property that it separates the two feature structures to be unified:

$$\mathsf{c1} < \mathsf{h} \leq \mathsf{c2}$$

for any heap cell c1 in the first operand, and any heap cell c2 in the second. This is an invariant property, since it is true all along the unification process.

Note that copying is called by directed unification only by need, and several optimizations are possible, like delaying adding feature structures (unnecessary, if unification fails), and indexing virtual copies (to be effectively created only when/if they will be really required by subsequent computations).

The OSF directed unification algorithm presented in Figure 13 is correct, i.e., it computes the glb of the input feature structures when

– performing OSF-term unification, i.e., no typing is assumed (the onLineExpansion register is set off), or
– doing OSF-theory unification wrt order- and type-consistent OSF-theories, i.e., if typing is assumed (the onLineExpansion register is set on), then the terms representing types are totally expanded (possibly up to unfilling), at compilation time.

Now, when doing Light grammar expansion, getting the order-consistent form of the input (the first phase of expansion) is obviously not taking typing into account, but then, while getting the type-consistent form of the grammar (during the second phase of expansion) types are expanded only one after another, at run time. Otherwise said, in this second expansion phase, we have to ensure that the output of osf_into_unify is fully consistent with all expanded type descriptions. In fact it is enough to check consistency only with those types that correspond to the root sorts of the osf_into_unify output's non-atomic substructures.

In order to achieve this aim, the unification procedure in Figure 13 has to be enhanced in the following way: the non-atomic subterms of the osf_into_unify output feature structure whose root sorts have been affected so to satisfy the condition onFlyExpansionCondition in Figure 14 will be "filled" by pushing the corresponding type constraints through application of the check_osf_into_unify_result presented in Figure 14. ("Filling" in this new function is done simply by osf_into_unification.)

Those (totally expanded) type constraints are assumed present on the heap at the representation[t] address for each type t corresponding to the root sorts in the set/list toBeChecked. If this condition is not (yet) satisfied, the application of consistent_osf_into_unify, the new form of osf_into_unify, is retried later. When that condition is satisfied, the result of consistent_osf_into_unify, that combines naturally osf_into_unify and check_osf_into_unify_result (see Figure 14) gets an equivalent result with osf_into_unify and osf_into_unify_result on the expansion result, the totally expanded form of the input grammar/theory.

After adding a pointer to the list toBeChecked as a new, additional argument for the function osf_into_unify, all these "ingredients" presented in Figure 14 are added to osf_into_unify by inserting the line

```
if onFlyExpansionCondition( new_sort, d1, d2, toBeChecked )
    *toBeChecked = cons( d2, *toBeChecked );
```

after the line

```
new_sort = glb( heap[ d1 ].SORT, heap[ d2 ].SORT);
```

```
int osf_into_unify( int i, int j, int h )
{
    PushDownList PDL;
    boolean fail = FALSE;

    push_PDL( &PDL, P, i );
    push_PDL( &PDL, P, j );

    while (PDL.top > 1 ∧ ¬fail) {
        t = deref( pop_PDL( &PDL ) );
        d1 = deref( X[ t ] );
        d2 = deref( pop_PDL( &PDL ) );
        if (d1 ≠ d2) {
            if (d1 < h)
                X[ d1 ] = d2;
            if (d1 ≥ h)
                fail = ¬osf_unify( d1, d2 );
            else {
                new_sort = glb( heap[ d1 ].SORT, heap[ d2 ].SORT);
                if (new_sort == BOT)
                    fail = TRUE;
                else
                if (¬bind_refine( d1, d2, new_sort, FALSE ))
                    fail = TRUE;
                else
                if (¬isAtomicFS( d1 ))
                    into_carry_features( d1, d2, h, &PDL ); } } }
    resetX();

    if (fail) return -1;
    else return d2;
}

into_carry_features( int d1, int d2, int h, PushDownList *pPDL )
{
    FEAT_frame *frame1 = heap[ d1 ].FTAB, *frame2 = heap[ d2 ].FTAB;
    FHEAP_cell *feats1 = frame1->feats, *feats2;

    if (isAtomicFS( d2 )) {
        int nf = frame1->nf;
        frame2 = new_frame( nf );
        push_TRAIL( d2, FEAT, 0 );
        frame2->nf = nf;
        feats2 = frame2->feats;
        for (k = 0; k < nf; ++k) {
            feats2[ k ].FEAT = feats1[ k ].FEAT;
            t = deref( feats1[ k ].TERM );
            if (X[ t ] < h)
                X[ t ] = copy_fs( t );
            feats2[ k ].TERM = X[ t ]; } }
    else carry_features( d1, d2, h, pPDL );
}
```

Fig. 13. The OSF directed unifier.

```
boolean onFlyExpansionCondition( int new_sort, int d1, int d2, int_list *toBeChecked )
{
   return
      (*toBeChecked ≠ NIL ∧ new_sort ≠ BOT ∧ new_sort ≠ heap[ d2 ].SORT ∧
        (new_sort ≠ heap[ d1 ].SORT ∧ (¬ isAtomicFS( d1 ) ∨ ¬ isAtomicFS( d2 )) ∨
         new_sort == heap[ d1 ].SORT ∧ isAtomicFS( d1 ) ∧ ¬ isAtomicFS( d2 )))
}

boolean check_osf_into_unify( int r, int_list *toBeChecked, int *representation )
{
   boolean result = TRUE;
   int_list *l;
   for (l = toBeChecked; result ∧ *l ≠ NIL; l = l -> next) {
      int k = l -> value; // take the 1st elem from *l
      int_list new_list = NIL;
      if osf_into_unify( representation[ s ], k, r, &new_list ) == -1
         result = FALSE;
      else
         append( toBeChecked, new_list ); }
   return result;
}

boolean consistent_osf_into_unify( int i, int j, int h, int *representation )
{
   int_list toBeChecked = NIL;
   return
      osf_into_unify( i, j, h, &toBeChecked ) ≠ -1 ∧
      (toBeChecked ≠ NIL ∨
        check_osf_into_unify_result( h, &toBeChecked, representation ));
}
```

Fig. 14. Building blocks of the type-consistent OSF directed unifier.

Note that the function check_osf_into_unify_result is somehow a hard-wired, lazy correspondent of on_line_expansion, to be used "on fly" in case the OSF-theory against/wrt which unification is performed is only partially type-consistent.