LIGHT AM — Another Abstract Machine for Feature Structure Unification

Liviu Ciortuz*

Language Technology Lab, DFKI, Saarbrücken, Germany
Email: ciortuz@dfki.de

This chapter will present the core part of LIGHT AM, an abstract machine designed for OSF-theory unification which is the core of the LIGHT system that we designed for head-corner parsing with feature grammars. LIGHT stands for Logic, Inheritance, Grammars, Heads, and Types. Building on the design of the abstract machine for unification of OSF-terms presented in [2], henceforth called OSF AM, we show exactly what constructions have to be added in order to perform OSF-theory unification for the class of order- and type-consistent OSF-theories. OSF-theory unification [5][6] generalizes both OSF-term (ψ-term) unification [4] and well-formed typed feature structure unification [9]. The LIGHT system was applied to do parsing with LinGO [18], a large-scale HPSG [29] grammar for English.

After Section 1 will give the reader an overview on different approaches to compilation of feature structure unification, Section 2 will construct the logical background in which LIGHT performs (OSF-theory) unification. Section 3 reviews the technical knowledge that our abstract machine borrowed from OSF AM. Section 4 presents an algorithm for lazy OSF-theory unification; it is an 'interpreted' alternative to the (eager) compiled OSF-theory unification achieved by LIGHT AM. Section 5, the main one of this chapter, presents all the constructs needed to extend the competence domain of OSF AM from OSF-term unification to OSF-theory unification. Section 6 overviews some of the control-level components of the LIGHT system, which are concerned with non-unification issues — namely, compilation of the head-corner parsing and the quick-check pre-unification filter [25] — and provides some measurements on the LIGHT system, and comparisons with related systems.

1 Background

A number of abstract machines for well-typed FS unification were developed in the past ([11], $\mathcal{A}$MALIA [33] and LiLFeS [26]). All of them have been derived basically from the WAM [32], and did not elaborate on (the specific difference w.r.t.) the OSF AM [2].[2]

The relation between all these abstract machines, together with the domains on which they work, is given in Figure 1. The typed feature structure theory [9], and the (more general) OSF-logic constraint theory [4] were elaborated somehow in parallel in the early 90's; the first one was influential in computational linguistics, the other one in constraint logic programming.[3] One can view the work on LIGHT as a challenging link between the two domains.

Carpenter defined the unification of well-typed feature structures [9]. Aït-Kaci, Podelski and Goldstein coined in [5][6] the notion of OSF-theory unification on order-consistent OSF-

* This is a slightly revised and extended version of the chapter published in "Efficiency in Unification-Based Processing", S. Oepen, D. Flickinger, J. Tsujii, H. Uszkoreit (eds), CSLI Publishers, University of Stanford, CA, 2002, pages 167–194.

[2] We note that [11] and [2] did not elaborate control (deduction or parsing) issues above unification.

[3] Aït-Kaci et. al. designed a set of rewriting rules performing OSF-theory unification on (only) order-consistent OSF-theories, but one of them is potentially non-terminating [5][6].

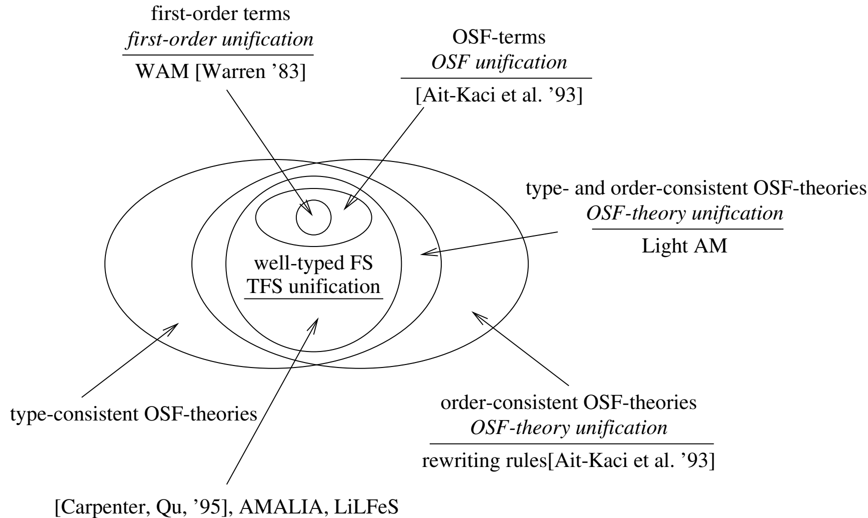

Fig. 1. The relation between LIGHT AM and other AMs for FS unification.

theories.[4] We introduced in [12] the notion of order- and type-consistent OSF-theory, which enables in a simple manner a more efficient expansion and unification with LinGO-like typed grammars [17]. The following *properties* hold:

- Every order- and type-consistent OSF-theory can be put under an equivalent well-typed form. Two OSF-terms/FSs are unifiable w.r.t. an order- and type-consistent OSF-theory if and only if they are unifiable in the corresponding well-typed theory, although the unification result is not identical (it is identical up to leaf nodes' type unfolding).
- In order- and type-consistent OSF-theories there is no need to use the notion of *appropriate function*, a central notion to the Carpenter's typed FS theory setup. For every order- and type-consistent OSF-theory, a minimal ("canonical") set of appropriateness constraints can be computed.[5]

When we enhanced the OSF AM in order to do parsing with LinGO, we in fact translated this grammar — or even more general: the conjunctive positive subset of $\mathcal{TDL}$ [23] — into a logic language called LIGHT, a CLP(OSF)-inspired language in which the set of predicate symbols is empty.[6] LinGO, which in fact can be put under (the equivalent) form of an order- and type-consistent OSF-theory, is thus seen as a LIGHT logic grammar.

We will show that upgrading the OSF AM so to be able to perform OSF-theory unification was not especially difficult: three main design improvements are required in order to make the OSF AM capable of unifying feature structures w.r.t. a given OSF-theory. These changes will be presented in the Section 5; one of them refers to the run-time unification function osf_unify, the remaining ones concern two of the OSF AM's abstract instructions.

[4] The OSF acronym stands for Order-Sorted Features.

[5] The design of the LIGHT abstract machine (AM) clearly shows that the appropriate constrains are used only for potential code optimizations, not in the central design of the machine.

[6] For the CLP schema see [20]. LIGHT can be seen as a successor of LIFE — Logic, Inheritance, Functions and Equalities — a well-known constraint logic language based on the OSF constraint system [4].

The LIGHT abstract machine for OSF-theory unification was enhanced with a control level for head-corner deductive parsing [21] [30], thus generalizing the chart-based parsing-oriented control level designed for $\mathcal{A}$MALIA, and putting under compiled form the idea of hyper-active parsing [27].

As far as we know, LiLFeS [26] is the only other existing compiler that does parsing with LinGO. It is based on an AM [24] that extends the abstract machine devised previously by Carpenter and Qu for unification of typed feature structures[11] and adds a relational control oriented level as in Prolog and Login [3]. LIGHT's control level is strictly head-corner parsing-oriented, and this helps to explain its better performances compared to LiLFeS (see Section 6).

2 LIGHT Logical Framework

Let $\mathcal{S}$ be a set of symbols called *sorts*, $\mathcal{F}$ a set of *features*, and $\prec$ a computable partial order relation on $\mathcal{S}$. We assume that $<\mathcal{S}, \prec>$ is a lower semi-lattice, meaning that, for any $s, s' \in \mathcal{S}$ there is a unique greatest lower bound glb(s, s') in $\mathcal{S}$. This glb is denoted $s \wedge s'$.

In the sequel, the notions of sort constraint, feature constraint and equality constraint, OSF-term (or ψ-term, or feature structure/FS, or OSF normalized clause) over the sort signature $<\mathcal{S}, \prec>$ are like in the OSF constraint logic theory [6]. The same holds for unfolding an OSF-term, and also for subsumption (denoted $\sqsubseteq$), and unification of two ψ-terms.

Notations: *root*(ψ) and $\psi.f$ denote the sort of the root node in the term ψ, and respectively the value of the feature f at the root level in ψ. The reflexive and transitive closure of $\prec$ will be denoted $\preceq$. The *logical form* associated to an OSF-term $\psi \equiv s[f_1 \rightarrow \psi_1, \ \ldots \ , f_n \rightarrow \psi_n]$ is $Form(\psi, X) \equiv \exists X_1 ... \exists X_n ((X.f_1 \doteq Form(\psi_1, X_1) \wedge ... \wedge X.f_n \doteq Form(\psi_n, X_n)) \leftarrow X : s)$, where $\leftarrow$ denotes logical implication, and $X, X_1, ..., X_n$ belong to a countable infinite set $\mathcal{V}$.

An OSF-*theory* is a set of OSF-terms $\{\Psi(s)\}_{s \in \mathcal{S}}$ such that *root*$(\Psi(s)) = s$, and for any $s, t \in \mathcal{S}$, the OSF-terms $\Psi(s)$ and $\Psi(t)$ have no common variables. The term $\Psi(s)$ will be called the s-sorted type, or simply the s type of the given OSF-theory. A *model* of the theory $\{\Psi(s)\}_{s \in \mathcal{S}}$ is a logical interpretation in which every $Form(\Psi(s), X)$ is valid.

The notion of OSF-term unification is naturally generalized to *OSF-theory unification:* ψ_1 and ψ_2 unify w.r.t. the theory $\{\Psi(s)\}_{s \in \mathcal{S}}$ if there is ψ such that $\psi \sqsubseteq \psi_1, \psi \sqsubseteq \psi_2$, and $\{\Psi(s)\}_{s \in \mathcal{S}}$ entails ψ, i.e., $Form(\psi, X)$ is valid in any model of the given theory.

Following the definition given in [6], an OSF-theory $\{\Psi(s)\}_{s \in \mathcal{S}}$ is *order-consistent* if $\Psi(s) \sqsubseteq \Psi(t)$ for any $s \preceq t$. We say that 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 $\psi \sqsubseteq \Psi(s)$. A term is said to be non-atomic (or: framed) if it contains at least one feature.

Example 1. Let us consider two OSF-terms

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

and a sort signature in which $\mathsf{b} \wedge \mathsf{c} = \mathsf{d}$ and the symbol $+$ is a subsort of the sort *bool*. We consider the OSF-theory made (uniquely) of

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

The glb (i.e., OSF-term unification result) of ψ_1 and ψ_2 is

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

while the $\{\Psi(\mathrm{d})\}$ OSF-theory relative glb (i.e., unification result) for ψ_1 and ψ_2 is

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

A *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
$\exists u \in \mathcal{S}$, such that $s \preceq u, t \preceq u$ and $f \in Arity(u)$;
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 $\psi.f \sqsubseteq \Psi(root(s.f))$,

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* w.r.t. the OSF theory $\{\Psi(s)\}_{s\in\mathcal{S}}$.

Let us add a couple of *notes:*

- 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. [7] The appropriate domain and values for all features $f \in \mathcal{F}$ define the 'canonical' *appropriateness constraints* for a well-typed OSF-theory.
- As a well-typed OSF-theory is (by definition) order-consistent, it implies that *Arity(s)* $\supseteq$ *Arity(t)*, and *Approp(f, s)* $\preceq$ *Approp(f, t)* for every $s \preceq t$;
- A stronger version for the condition *ii.* would be: the feature f is defined (at the root level) for ψ if and only if $f \in Arity(s)$, and $\psi.f \sqsubseteq \Psi(s.f)$. In the latter case, the theory is said to be *totally well-typed*.
- For well-typed OSF theories $\{\Psi(s)\}_{s\in\mathcal{S}}$, the notion of OSF-unification extends naturally to *well-typed* OSF-*unification*. The well-typed glb of two feature structures ψ_1 and ψ_2 is the most general (w.r.t. $\sqsubseteq$) 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.
- Obviously, the well-typed OSF-theories are a particular class of order- and type-consistent OSF-theories. On this class, well-typed unification coincides with OSF-theory unification.[8]

To *summarize*, the *first difference* between the class of order- and type-consistent OSF-theories (on one side) and the class of well-typed OSF-theories (on the other side) concerns the subsumption condition, which for the latter class is limited to non-atomic substructures ψ: if $root(\psi) = s$, then $\psi \sqsubseteq \Psi(s)$. For instance, if $\mathsf{a}[\mathsf{F} \rightarrow \mathsf{cons}]$ is type-consistent, its well-typed correspondent would be $\mathsf{a}[\mathsf{F} \rightarrow \mathsf{cons}[\mathsf{FIRST} \rightarrow \mathsf{top},\ \mathsf{REST} \rightarrow \mathsf{list}]]$.

---

[7] Our current implementation of 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 $Approp Dom(f) = \text{luub}(s, t)$, and $Approp Val(f) = \text{luub}(root(s.f),\ root(t.f))$, where luub, the least unique upper bound of the two sorts, is always guaranteed to exist.

[8] The reader will note that the role of the check_osf_unify_result function further defined in the Section 4 is ensuring that the glb computed in LIGHT AM by the osf_unify function when applied to two well-typed feature structures is 'refined' to the glb well-typed feature structure (by enforcing condition *ii.* from above).

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

```
push_cell 0
set_sort 0, vp
push_cell 1
set_feature 0, ARGS, 1
set_sort 1, cons
push_cell 2
set_sort 2, verb
push_cell 3
set_feature 2, HEAD, 3
push_cell 4
set_feature 2, OBJECT, 4
set_sort 4, np
push_cell 5
set_feature 2, SUBJECT, 5
set_sort 5, sign
push_cell 6
set_list 1, 2, 6
set_sort 6, cons
set_list 6, 4, nil
set_feature 0, HEAD, 3
set_feature 0, SUBJECT, 5
```

Fig. 2. A sample OSF-term and the corresponding 'query' abstract code.

This (more relaxed) condition has been proved to be truly beneficial for LinGO-like grammars [7], since it lead to a significant reduction of both the expanded size of the grammar and the parsing time (due to reduction of copying or other structure manipulation operations), without needing a stronger notion of unification.

The *second* main *difference* between order- and type-consistent OSF-theories on one side, and well-typed OSF-theories on the other side is related to appropriate features: well-typed theories do not allow a subterm ψ of root sort s to use features not defined at the root level in the corresponding type $\Psi(s)$.

Example 2. If $\psi_5 = \mathsf{a}[\ \mathsf{FEAT1} \rightarrow \mathsf{d}[\ \mathsf{FEAT2} \rightarrow +, \mathsf{FEAT3} \rightarrow \mathit{bool}\ ]\ ]$, then the OSF-theory glb of ψ_5 and ψ_2 from Example 1 will be defined (and equal to ψ_5), while their well-typed glb relative to the same theory does not exist, simply because ψ_5 is not well-typed w.r.t. $\Psi(d)$, due to the non-appropriate feature FEAT3.

Therefore, LIGHT will allow the grammar writer more freedom. The source of this freedom resides in the *openness* of OSF-terms. Also, at the implementation level, LIGHT AM works with free-order registration of features inside feature frames. (The $\mathcal{A}$MALIA and LiLFeS abstract machines for unification of well-typed FS work with closed records and fixed feature order for a given type.) It is interesting to note that on one side, the free order of features is well suited for incremental parsing with shared feature structures, and on the other side, work by Callmeier [8] has shown that fix-order feature storing does not lead to improvement of the parse performance on LinGO. We will show at the end of the Section 5 an optimization done for LIGHT AM when taking into account appropriate constraints.

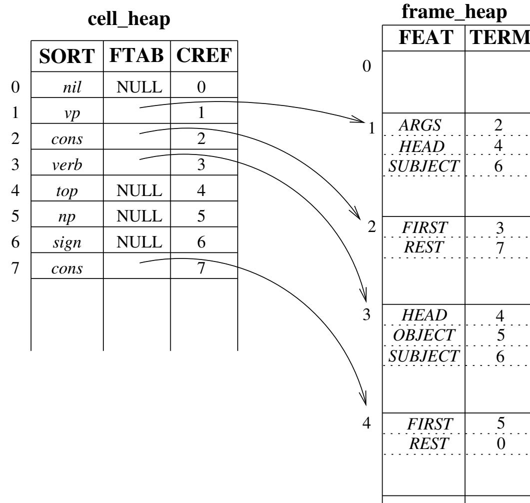

Fig. 3. The internal representation of the `vp` OSF-term given in Figure 2.

3 OSF AM: Data Structures and Abstract Instructions

In the section we will familiarize the reader with the AM main constructs, the data structures and abstract instructions as presented in [2]. Figure 3 illustrates what a feature structure as shown on the left hand side in Figure 2 looks like when represented on the basic data structures of the OSF and LIGHT AMs. This low-level representation of the given OSF-term is obtained by calling the ('query') function whose abstract code is provided on the right hand side in Figure 2.

OSF AM has two stacks — one of *cells*, the other of *feature frames*. The first is the most important one, and will usually be called simply the AM's heap. Each cell stores information about a node in a feature structure:

– SORT, the sort (index in the symbol table);
– FTAB, the feature frame address (in the frame heap);
– CREF, the coreference, containing the index of a heap cell.

Implicitly, if a heap cell c has the index value i in the heap, then the CREF field of c is set to i. If, during unification, the feature structure, whose root node is c gets instantiated/coreferenced to another, more specific feature structure whose root node/cell has the index value j, then the CREF field of c changes its value to j. While not shown explicitly in Figure 3, a frame in the frame_heap has in fact two subcomponents: nf, the number of features actually stored in that frame, and features, the address of a chunk of feature-value pairs in a global array reserved for this purpose.

The osf_unify function as provided in [2] performs unification of two OSF-terms represented on the heap of the OSF AM (and equally LIGHT AM) AM. It is given — adapted to the pseudo-code we use, close to the C programming language — in Figure 4.

The osf_unify function tests whether two OSF-terms found on the heap at the addresses a1 and respectively a2 are unifiable. If so, unification will combine destructively the two terms so to provide the unification result. A stack — which is called PDL, as abreviation from 'push down list' —, will store (pairs of) corresponding feature path values in the two FSs.

```
bind_refine( int d1, int d2, sort s )
{
    heap[ d1 ].CREF = d2;
    heap[ d2 ].SORT = s;
}

carry_features( int d1, int d2 )
{
    FEAT_frame *frame1 = heap[ d1 ].FTAB, *frame2 = heap[ d2 ].FTAB;
    FHEAP_cell *feats1 = frame1 -> feats, *feats2 = frame2 -> feats;
    int feat, nf = frame1 -> nf;

    for (feat = 0; feat < nf; ++feat) {
        int f, f1 = feats1[ feat ].FEAT, v1 = feats1[ feat ].TERM, v2;
        if ((f = get_feature( d2, f1 )) ≠ FAIL) {
            v2 = feats2[ f ].TERM;
            push_PDL( &PDL, v2 );
            push_PDL( &PDL, v1 ); }
        else  add_feature( d2, f1, v1 ); }
}

boolean osf_unify( int a1, int a2 )
{
    boolean fail = FALSE;

    push_PDL( &PDL, a1 ); push_PDL( &PDL, a2 );
    while non_empty( &PDL ) ∧¬fail {
        d1 = deref( pop( &PDL ) ), d2 = deref( pop( &PDL ) );
        if d1 ≠ d2 {
            new_sort = heap[d1].SORT ∧ heap[d2].SORT;
            if new_sort = BOT
                fail = TRUE;
            else {
                bind_refine( d1, d2, new_sort )
                if deref( d1 ) = d2
                    carry_features( d1, d2 );
                else  carry_features( d2, d1 ); } } }
    return ¬fail;
}
```

Fig. 4. The osf_unify function.

First, a1 and a2, the roots of the two input FSs are pushed onto PDL. Then, for each pair of cell indices (d1, d2) popped from PDL, if (after dereferenciation, see below) they are proven different one from another, then the compatibility of their respective SORTs is checked. In case they are compatible, using the CREF field of one cell the bind_refine procedure links it to the other cell; the same procedure stores the computed (*glb*) sort in the SORT field of the second cell. Finally,

– for all features which are retrieved in the frames of both cells, the carry_feature procedure will push into PDL the corresponding feature values, and
– features in the first cell's frame but not in the second will be registered also, together with the respective values, in the second cell's frame.

The *derefenciation* operation refers to (traversing) the chain of CREF fields starting from a designated cell and ending with the cell whose CREF field points to itself. It is exactly the address/index of this last cell that the function deref, invoked by osf_unify and certain abstract instructions, returns. Note that the CREF fields are affected by the function bind_refine, called by osf_unify, which in turn may be called by the unify_feature abstract istruction, presented in the sequel.

The OSF AM performs compiled OSF-unification. The pseudo-code of the OSF AM's abstract instructions is given in a form slightly adapted from [2] in Figure 5.

The abstract instructions of OSF AM are devided into two categories: *READ* instructions and *WRITE* instructions. This categorization is related to the so-called two-stream optimisation which has been imported into the OSF AM desgin from the Warren Abstract Machine [1]. The *READ* instructions in OSF AM are: push_cell, set_sort, and set_feature. *WRITE* instructions are: intersect_sort, test_feature, unify_feature, and write_test.

In LIGHT, sequences of abstract instructions build up functions which compile OSF-terms (FSs). The code of the so-called 'query' functions will be made exclussively of *WRITE* abstract instructions. 'Program' functions will employ both *READ* and *WRITE* instructions. In LIGHT grammars, 'query' terms are those destined to be built on the heap (by the execution of the corresponding 'query' functions/code. 'Program' terms are those which (under compiled form) will unify with FSs already found on the heap.

We give below a short description for each one of the abstract instructions in OSF AM.

- The push_cell abstract instruction reserves the next cell available on the AM's heap. (That cell is pointed to by the H register; after the execution of push_cell, the value of the register H is incremented by 1.) The address/index of this newly allocated cell is stored in X[i+Q]. Q is the register that points to the root of the feature structure which is currently under construction, or is subject to unification with another, compiled FS. The three fields of the alocated cell are (re)set to implicit values, making it a *top*-sorted, self-referenced, atomic FS.
- For the cell whose address is stored in X[i+Q], the set_sort abstract instruction sets the SORT field to the indicated sort s.
- The set_feature abstract instruction adds the attribute-value pair (f, X[j+Q]) to the frame of features associated to the cell whose address is given by X[i+Q] via dereferenciation.
- The intersect_sort abstract instruction takes the SORT field of the cell pointed by X[i+Q] via dereferenciation, and makes its intersection (*glb*) with the given sort s. If the intersection operation is succesful, then its result replaces the old value of the referred SORT field. Otherwise, the flag fail is set TRUE.
- The test_feature abstract instruction tests whether the feature f is among the attribute-value pairs of the cell pointed to by X[i+Q], via dereferenciation. If so, it sets X[j+Q] to the actual value of that feature. Otherwise it sets the D ('depth') register to the indicated level, and jumps to the label W_l in the *WRITE* stream of instructions.
- The unify_feature abstract istruction checks whether the feature f is among the attributes of the FS addressed (through dereferenciation) by X[i+Q]. If so, it unifies the actual value of that feature with the FS rooted by X[j+Q]. Otherwise it adds the pair (f, X[j+Q]) to the feature frame of cell designated by X[i+Q].
- The write_test abstract instruction compares the value of the register D with a designated level. If D has a greater value, then the execution control is passed to the label R_l in the *READ* stream of instructions.

Note that we inserted push_TRAIL operations into the set_feature, intersect_sort and unify_feature abstract instructions, in order to be able to further undo the changes that these

```
push_cell i:int ≡
    if i+Q ≥ MAX_HEAP ∨ H ≥ MAX_HEAP
        error( "heap allocated size exceeded\n" );
    else {
        heap[ H ].SORT = TOP;
        heap[ H ].FTAB = FTAB_DEF_VALUE;
        heap[ H ].CREF = H;
        setX( i+Q, H++ ); }

set_sort i:int, s:sort ≡
    heap[ X[ i+Q ] ].SORT = s;

set_feature i:int, f:feat, j:int ≡
    int addr = deref( X[ i+Q ] );
    FEAT_frame *frame = heap[ addr ].FTAB

    push_TRAIL( &TRAIL, addr, FEAT,
                    (frame ≠ FTAB_DEF_VALUE ? frame−>nf : 0) );
    add_feature( addr, f, X[ j+Q ] );

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

    if new_sort = ⊥
        fail = TRUE;
    else {
        if s ≠ = new_sort
            push_TRAIL( &TRAIL, addr, SORT, heap[ addr ].SORT );
        heap[ addr ].SORT = new_sort; }

test_feature i:int, f:feat, j:int, level:int, l:label ≡
    int addr = deref( X[ i+Q ] ), p;
    int k = get_feature( addr, f );

    if k ≠ FAIL
        X[ j+Q ] = heap[ addr ].FTAB.features[ k ].VAL;
    else
        { D = level; goto W_l; }

unify_feature i:int, f:feat, j:int ≡
    int addr = deref( X[ i+Q ] ), k;
    FEAT_frame *frame = heap[ addr ].FTAB;

    if (k = (get_feature( addr, f )) ≠ FAIL)
        fail = osf_unify( heap[ addr ].FTAB.feats[ k ].TERM, X[ j+Q ] );
    else {
        push_TRAIL( &TRAIL, addr, FEAT,
                    (frame ≠ FTAB_DEF_VALUE ? frame−>nf : 0) );
        add_feature( addr, f, X[ j+Q ] ); }

write_test level:int, l:label ≡
    if D ≥ level
        goto R_l;
```

Fig. 5. Abstract instructions in OSF AM.

push_cell 0 set_sort 0, a push_cell 1 set_feature 0, FEAT1, 1 set_sort 1, b	push_cell 0 set_sort 0, a push_cell 1 set_feature 0, FEAT1, 1 set_sort 1, c push_cell 2 set_feature 1, FEAT2, 2 set_sort 2, bool

Fig. 6. The 'query' OSF abstract codes for ψ_1 (left) and ψ_2 (right) from Example 1.

(and the other) instructions produce on both heaps of the OSF AM. To make this move complete, further extensions to the bind_refine function will show how it also incorporated a call to push_TRAIL.

Example 3. When compiling as 'query' terms the ψ_1 and ψ_2 OSF-terms given in Example 1, the abstract code produced by the LIGHT system will be as shown in Figure 6. Executing this code will build on the heap the representations (that will be further affected by unification) in Figure 8 (see Example 4). 'Program' versions of the two terms will be shown later in Figures 12 and 14 (see Examples 6 and 7).

4 A Lazy OSF-theory Unifier

The present work is concerned mainly with getting the compiled form of OSF-theory unification w.r.t. order- and type-consistent theories. However, for sake of exposure completeness we will give also an interpreted-like OSF-theory unification algorithm for the same class of theories. Called consistent_osf_unify and presented in Figure 7, it is a natural extension of osf_unify.

The key insight in the design of consistent_osf_unify is that certain type constraints must be locally checked/propagated (by the function check_osf_unify_result) after the execution of osf_unify. The heap cells which identify exactly the places (inside the to-be-unified terms on the heap) where those type constraints must be checked are identified via the extended version of bind_refine, presented in the same figure.[9] The list toBeChecked stores the indices of these cells, and its address is stored in the register ToBeChecked we added to the abstract machine.

Example 4. Let us consider the OSF-terms given in Example 1. When applied to the ψ_1 and ψ_2 representations on the heap — obtained by runnning the 'query' functions given in Figure 6 — osf_unify will produce ψ_3, while consistent_osf_unify will produce ψ_4, provided that $\Psi(\mathsf{d})$ is also represented on the heap at the address representation[d]. During the application of consistent_osf_unify, the ∗ToBeChecked list will be constructed out of the root cell corresponding to the subterm d[FEAT2 → *bool*] in ψ_3. The function check_osf_unify_result will subsequently unify exactly this subterm with the representation of $\Psi(\mathsf{d})$. A heap representation corresponding to this example can be seen in Figure 8. The strike-lined data

[9] The expansionCondition function called by this improved version of bind_refine appears in Figure 9 and will be explained in detail in the next section.

```
bind_refine( int d1, int d2, sort s )
{
    heap[ d1 ].CREF = d2;
    if ToBeChecked ≠ NULL ∧ expansionCondition( d1, d2, s )
        ToBeChecked = cons( d2, ToBeChecked ); }
    heap[ d2 ].SORT = s;
}

int check_osf_unify_result( int_list *toBeChecked )
{
    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 s = heap[ k ].SORT;
        int_list new_list; = NIL

        if  representation[ s ] = 0
            { ToBeChecked = l, result = -TRUE; }
        else {
            ToBeChecked = &new_list;
            if osf_unify( representation[ s ], k ) = FALSE
                result = FALSE;
            else append( l, ToBeChecked ); } }
    return result;
}

boolean consistent_osf_unify( int i, int j )
{
    int_list toBeChecked = NIL;
    ToBeChecked = &toBeChecked;
    return
        osf_unify( i, j ) ∧
        (*ToBeChecked = NIL ∨ check_osf_unify_result( ToBeChecked ));
}
```

Fig. 7. The (type-)consistent version of the osf_unify function.

in this figure correspond to modifications done on the heap during the execution of consistent_osf_unify(ψ_1, ψ_2).

Note that the way consistent_osf_unify extends osf_unify to perform OSF-theory relative unification is a lazy one. An eager version will be obtained in the sequel (section 5.1), when we will (further) extend the function bind_refine. That new version of (an interpreted-like) OSF-theory unifier will simply consist in calling the function osf_unify with the register OnlineExpansion turned on.

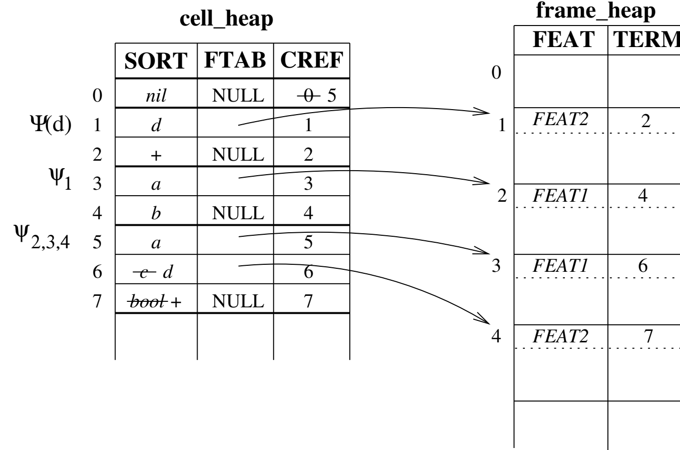

Fig. 8. The effect of OSF-theory unification on ψ_1 and ψ_2 from Example 1.

5 Compiling OSF-theory Unification

The input (LinGO-like) grammar for our system is translated into the form of an OSF-theory,[10] and then it is *expanded*. Expansion, detailed in [12], is a type inference technique (see [9], chapter 6) that puts the input grammar into an equivalent order- and type-consistent form, which is also well-typed w.r.t. its canonical appropriateness constraints.

The operations described in this section are related to the dynamic checking for compatibility with the involved types, achieved through a form of 'filling' — according to [9] terminology — the appropriate type constraints when needed. This form of filling, or *on-line type expansion*, as we used to call it, ensures the soundness of OSF-theory unification.

During feature structure unification,

- leaf nodes of a feature structure may become framed, i.e, they can get associated features, carried from another feature structure, whose root sort is less specific than the target leaf node's sort; or
- non-atomic 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.

These are reasons 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] must be enhanced as described in the sequel, in order to achieve OSF-theory unification.

5.1 Extending osf_unify

The bind_refine function which corresponds primarily to equation constraint propagation simply by affecting one heap cell's CREF field, has been enhanced to the version shown in Figure 9 so perform two more tasks:

[10] For LinGO, which comes in $\mathcal{TDL}$ format, this work was done by the precompiler implemented by U. Callmeier, starting from our specifications.

```
boolean bind_refine( int d1, int d2, sort s )
{
    push_TRAIL( &TRAIL, d1, LINK, heap[ d1 ].CREF );
    heap[ d1 ].CREF = d2;
    if heap[ d2 ].SORT ≠ s
        push_TRAIL( &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; }
}

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. 9. The new (LIGHT) version of the bind_refine function.

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

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

    return result;
}
```

Fig. 10. The on_line_expansion function.

- 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 cells 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 encoded into the expansionCondition function also shown in Figure 9.

The on_line_expansion function is presented in Figure 10. This function has to propagate onto the feature structure recorded at the address addr on the heap the constraints associated with the type $\Psi(s)$, in order to check for consistency. In the LIGHT AM's current implementation, the type $\Psi(s)$ 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 (meta-)function program with the argument p. The effect is exactly the unification of the feature structure rooted at addr (previously stored in the register Q) with the type $\Psi(s)$.

What remained to be explained in the behavior of on_line_expansion are 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 our LIGHT AM implementation, only those different from the implicit values) are saved/restored;
- the index p is pushed onto the programPDL stack before on-line expansion, and respectively popped out afterward; the top element in this stack will be 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 5. Let us consider the ψ-terms $\psi_1, \psi_2, \Psi(\mathsf{d})$ and the sort signature from Example 1. It is obvious that, if ψ_1 and ψ_2 are represented on the heap,[11] 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.FEAT1 term, namely c[FEAT2 → *bool*]. As soon as the root sort of this term was refined to d, it is transformed into d[FEAT2 → +] by unification with $\Psi(\mathsf{d})$.

Note: If the following conditions hold:

i. every type s in the OSF theory (w.r.t. which unification is performed) is present on the heap at the address representation[s], where s is the symbol index corresponding to s,
ii. the line result = program(p); in the code of on_line_expansion is replaced with result = program(Q, representation[s]); and
iii. the OnLineExpansion register is turned on,

then the effect of applying osf_unify is the same with the effect of consistent_osf_unify presented at the end of the previous section, achieving — again, like consistent_osf_unify — interpreted-like OSF theory unification, but now in an eager manner.

5.2 Extending intersect_sort and test_feature

Fully compiled OSF theory unification will require modification of two abstract instructions in OSF AM, namely intersect_sort and test_feature (which are actually implied in the application of the function program(p) in the code of on_line_expansion), to be presented in the sequel.

The understanding of (the new form of) the intersect_sort abstract instruction, presented in Figure 11, is quite easy now, due to the explanations given above, which apply also here. The only thing to be added is that prior to 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.

[11] In order to obtain those representations, one could run the 'query' functions whose abstract code has already been provided in Figure 6.

```
intersect_sort i:int , s:sort ≡

  int addr = deref( X[ i+Q ] ), 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( &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. 11. The enhanced intersect_sort abstract instruction.

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

Fig. 12. Abstract 'program' code for the term ψ_1 in the Example 6.

Example 6. Let us consider again $\psi_1, \psi_2, \Psi(\mathsf{d})$ 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 (see Figure 12), when the instruction intersect_sort 1, b will be executed, the computed new_sort will be d, which is different from b. Therefore on_line_expansion will be applied, and it will add the feature FEAT2 with the value + at ψ_2.FEAT1.

The test_feature abstract instruction basically has to test whether the feature structure rooted by the cell whose index is X_i has the feature feat present at root level. If so, it will instantiate X_j to (the root address of) the feat's value term.

Concerning the new form of the test_feature abstract instruction, presented in Figure 13, 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 root sort of the current feature path's value in the type whose compiled code contains this test_feature instruction. If the addr cell, the current path's value for the target term on the heap, has the sort new_sort, more specific than s, then on-line expansion must be applied, since an s-specific feature constraint will be added at addr, and therefore we have to check

```
test_feature i:int , feat:int , j:int , level:int , l:label, s:sort ≡

    int addr = deref( X[ i+Q ] ), p;
    int f = get_feature( addr, feat );

    if f ≠ FAIL
        X[ j+Q ] = 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+Q ] = heap[ addr ].FTAB.features[ f ].TERM; }
            else { D = level; goto W_label; } }
        else D = level; }
```

Fig. 13. The enhanced test_feature abstract instruction.

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

Fig. 14. Abstract 'program' code for the term ψ_2 in the Example 7.

its "appropriateness" for new_sort.

Example 7. Again, consider $\psi_1, \psi_2, \Psi(\mathsf{d})$ and the sort signature like in Examples 1, 5 and 6. Now let ψ_1 be represented on the heap and ψ_2 be compiled as a 'program' term (see Figure 14). The instruction test_feature 1, FEAT2, 2, 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 1, c, that did not apply on_line_expansion since the ψ_1.FEAT1 substructure on the heap was not framed (yet)). Now, as $\mathsf{d} \neq \mathsf{c}$, the on_line_expansion function is invoked by the instruction test_feature, adding the feature FEAT2 with the value + to the subterm ψ_1.FEAT1.

5.3 A Technical Comparison with $\mathcal{A}$malia

At a first sight, our on_line_expansion function seems to play a similar role to that of Wintner's build_most_general_fs function in $\mathcal{A}$MALIA, since they both achieve type checking

through associated constraints filling.[12] However, there are important differences between build_most_general_fs and on_line_expansion:

Firstly, 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 and output. On the contrary, the function on_line_expansion is a *basic*, crucial construct for LIGHT AM, understood as one of the *extensions* to the OSF AM that together make it capable of performing OSF-theory (and therefore type) unification. It may not be eliminated in any way from the LIGHT's operational semantics.

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

Thirdly, 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 was (then restrictive) ALE-like [10], allowing neither coreference/equality constraints on different feature paths, nor refining of s-type constraints in a substructure of root sort s. The OSF/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 filled/unfolded, could affect a certain node is impractical, since among those nodes are not only members of the compiled term that performs the current unification, but also other nodes, in the enveloping super-structure, whose filling was priorly delayed.

Finally, 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 tries simple filling, i.e., full type unfolding. On these nodes, filling is complete if type constraints' checking succeeds.

To summarize, $\mathcal{A}$MALIA's build_most_general_fs strategy is: do lazy breadth-first type construction in view of further feature constraint checking during unification, while LIGHT AM's on_line_expansion strategy is: during unification do eager depth-first type checking for sort-refined, framed or to-be-framed sub-structures.[13]

5.4 Some Optimizations

LIGHT is designed for order- and type-consistent theories. When taking into account also the (canonical) appropriateness constraints, what we saw is that:

1. Relative to the *feature introduction* property, the LIGHT AM supports unfilling [19] of the expanded form of the input grammar. The effect of unfilling is very important on reducing the size of the grammar, but it does not provide a dramatic speed up for unification.

Also related to this property is a simple optimization added to the (compiler that generates code for) LIGHT AM when dealing with well-typed grammars: 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 the feature f defined, then its value is t or a subsort of t. For example, in LinGO, the type

[12] Unfortunately, neither the definition of this function, nor the conditions under which it is applied were formalized in [33], section 3.4.1 and [34], section 5.1.

[13] Note that the eager-ness of our *on_line_expansion* mechanism might be decreased if the calls to on_line_expansion are stacked, and their execution is done only if unification succeeds.

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

Fig. 15. Example of relaxed code w.r.t. maximal appropriate values.

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

is coded as shown in Figure 15. The underlined intersect_sort instructions are eliminated, since maximal appropriate values for PN and GEN are pernum and gender respectively.

2. The arity constraints imposed by well-typedness have no direct consequence on the LIGHT system design. For the other existing AMs that support typed unification, the issue of fixed arity is a basic assumption in the design of the data structures; it leads to the representation of features frames directly on the (main) heap, and also provides direct feature retrieval inside feature frames, based on fixed-order storage. Instead, the openness of (the representation of) OSF-terms we adopted for LIGHT AM lead to a very convenient way to share feature structure representations during parsing; it is presented in detail in the technical report [13].

6 Beyond Compiled OSF-theory Unification — LIGHT on LinGO

The LIGHT AM as it was presented in the precedent section was extended with a control level so to achieve head-corner parsing with FS unification grammars. Two important issues have to be mentioned in this respect:

Firstly, we introduced in [14] the *specialized compiled form* of feature structures representing rules. In this approach, every rule is associated two different execution modes: a *key mode* and a *complete mode*. This dissociation allows for

— saving time and space by partitioning the creation of feature structures for rule instances: basically, the FS corresponding to an argument instance is built only if unification with that argument succeeds.

This idea is also behind the hyper-active parsing, which is based on the "create once, delay copying sub-structures as long as possible" strategy. The strategy incorporated into the specialized form of rule compilation is "delay FS (partition) creation as long as

Overall:	memory use	process size full	process size resident	average parse time
regular compilation	59.5MB	73MB	80MB	128 msec
specialized compilation	3.9MB	44MB	13MB	35 msec

Detailed:	heap cells	feature frames	environ-ments	trail cells	corefe-rences
regular	1,915,608	1,050,777	2669	128,747	0
specialized	77,060	57,663	2669	77,454	22,523

Fig. 16. Regular vs. specialized compilation: a comparison between the respective effects on parsing.

possible". Note that in the compilation approach, FS creation is cheaper than copying. The partitioning into copy/creation delay-able parts is finer grained in hyper-active parsing. It uses a (dynamic) indexation scheme for substructures whose copying can be delayed.

— dealing with quasi-destructive FS sharing, using environments. The environment creation is easily done staring from backtrack information stored in the abstract machine's trail during unification.

Efficient parsing with FS sharing requires the minimization of (time and space) for environment manipulation. We achieve this aim in the LIGHT system by *i.* delaying as much as possible to save environments, and *ii.* minimizing the number of times an environment is restored.

The specialized compilation of rules provided for parsing with the LIGHT system a speed up of 72% (on the *CSLI* test suite, without running the quick-check) compared to 27% on the same test suite for the LKB system [16], as reported in [27].[14] Our specialized compilation strategy also reduced dramatically the memory space used during parsing, as one can see in Figure 6. Again, the numbers refer to parsing the *CSLI* test suite, and hey have been obtained on a Pentium III PC at 933MHz running Red Hat Linux 7.1.

Secondly, we designed in [15] a compiled version of the "quick-check" pre-unification filter [25]. We argued that the specialized compilation forms of rules makes the computation of quick-check vectors for non-key arguments impossible at run-time if the pre-computation (i.e., compilation) of those vectors is ruled out. Compiling the quick-check makes possible several optimisations that cannot be designed in interpreted mode.

In order to get an evaluation of the (unification) power of LIGHT AM, we compared its performance with PET — known as the fastest system running LinGO [28]. The following measurements were done when the development of the LIGHT system started to be frozen, due to the departure of its author from DFKI Saarbruecken. Both systems have been run on a SUN Sparc server at 400MHz.

If the two systems run without quick-check filtering, when parsing a same test suite the two systems perform the same unifications. On the *CSLI* test suite, LIGHT AM scored 0.06 seconds/sentence while PET reported 0.11 seconds/sentence. With quick-check turned on, the LIGHT system registered the same performance than PET on the *CSLI* test suite: 0.04

[14] A speed up factor of 40% was obtained by hyper-active parsing with LKB on the *fuse* test suite.

seconds/sentence.[15] The LiLFeS system with CFG filtering scored for the same test suite 0.06 seconds/sentence on a SUN Sparc station at 336MHz (that would correspond to 0.05 on a 400MHz station).

PET's overall development status was then (and continues to be) more advanced than that of the LIGHT system simply because it is an interpreter and normally the development of compilation techniques require considerably more time than an interpreter implementation. It should also be noted that PET's good performance compared to the LIGHT system is due to the incorporation of the Tomabechi's efficent unification algorithm [31], while LIGHT uses a compiled version of a simple unification alrgorithm. In PET computations have been already made as local as possible — leading to a speed up of about 40% — but this optimization was not yet done for LIGHT AM. Therefore we can estimate that compiled unification (in LIGHT AM) may easily become more than twice faster than interpreted unification (in PET).

The above comparative measurements — which attempt to judge among systems currently found on different ways or stages of development — may justify the following thoughts:

Firstly, compiling FS unification did not lead to such an impressive speed up factor as compilation of Prolog provided, since

- LinGO-like grammars deal with very large feature structures (having hundreds or even thousands of nodes), which is not at all the normal case for Prolog programs, and
- apart from syntactic and lexical filters, little can be foreseen at compilation time about the parsing itself for such grammars. (The heavily lexicalized form of these grammars, and implicitly the reduced number of rules, seems to turn against them in practice, regarding the efficiency of parsing.) Instead, in Prolog, compiling its relational reasoning level (based on top-down SLD-resolution) is much valuable;
- while compiled FS unification is significantly faster than interpreted unification, the quick-check pre-unification filter nearly eliminates the difference in parsing performances.

Secondly LinGO-like grammars have a highly dis-uniform distribution of failure paths inside the rule (argument) feature structures. About a quaker of feature paths — the so-called quick-check paths — are responsible for most of unification failures (or, equivalently: rule selection at the run-time). This fact implies that:

- in grammars with a much more uniform distribution of failure paths, the quick-check pre-unification filter is not proficient, and it will not anymore "save" interpreted systems in their competition against compiler systems;
- efforts to automatically transform LinGO-like (sub-)grammars in which failure paths are more uniformly distributed will lead probably to the creation of certain post-unification filters. (Such a filter would be responsible for ruling out — by filling/unfolding information associated to certain nodes in — presumable parses which were obtained by running a core of the grammar.) We claim that this kind of grammar transformations will clearly favor compilation against interpretation.

Thirdly, Supplementing the declarative nature of unification grammars with procedural ingredients will help very much. This was the case in Prolog with its top-down depth-first rule selection strategy, the cut construct ('!'), and some pre-defined predicates, although they sacrificed (for Prolog) the completion aspect of first-order logic reasoning.

In fact, the quick-check filter can be already seen as (or, better: generalized to) such a procedural ingredient. Unification in the current version of the LIGHT system proceeds by traversing simultaneously the two features structures in depth-first manner. The QC

[15] Our system proved to be sensibly faster than PET.

test, when seen as integrated into unification, affects this traversal order by first performing the *glb*-sort operations on corresponding QC-path values. (Note that the computation of QC-path values can be fully integrated into compiled unification.)

Conclusion

We extended the design of the AM for unification of (non-typed) OSF-terms so that it can be used to perform OSF-theory unification (if the theory is order-and type-consistent) and, as a consequence, well-typed feature structure unification.

Unification in LIGHT AM is "controlled" by a parsing-oriented level (that corresponds to the SLD-resolution level in the classical WAM [1]). Our strategy — incremental head-corner bottom-up chart-based parsing with quasi-destructive FS sharing — is substantially more general than the simple bottom-up chart-based parsing in $\mathcal{A}$MALIA [34]. (We also incorporated into the LIGHT AM's parsing control level all the main optimisations presented in [22]: the feature structure restrictor, the syntactic and lexical rule filtering.) In this view, our machine can be seen as inheriting from and generalizing both OSF AM [2] and $\mathcal{A}$MALIA [34].

A specialized compiled form of rules [14] is obtained in the LIGHT system via transformation of the abstract code generated (with the 'two-streams' optimization) by the OSF AM for rules represented as feature structures. Moreover, we integrated in LIGHT AM a compiled form of the pre-unification quick-check filter [15].

References

1. H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. 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.
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. MIT Press.
6. H. Aït-Kaci, A. Podelski, and S.C. Goldstein. Order-sorted feature theory unification. *Journal of Logic, Language and Information*, 30:99–124, 1997.
7. 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.
8. U. Callmeier. Pre-processing and encoding techniques in PET. In D. Flickinger, S. Oepen, J. Tsujii, and H. Uszkoreit, editors, *Collaborative Language Engineering*. Center for the Study of Language and Information, University of Stanford, CA, 2002.
9. B. Carpenter. *The Logic of Typed Feature Structures – with applications to unification grammars, logic programs and constraint resolution*. Cambridge University Press, 1992.
10. 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.
11. 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.
12. L. Ciortuz. Expanding feature-based constraint grammars: Experience on a large-scale HPSG grammar for English. In *Proceedings of the IJCAI 2001 co-located Workshop on Modelling and solving problems with constraints*, Seattle, USA, 2001. Downloadable from http://www.lirmm.fr/~bessiere/proc_wsijcai01.html.

13. 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.
14. L. Ciortuz. On compilation of head-corner bottom-up chart-based parsing with unification grammars. In *Proceedings of the IWPT 2001 International Workshop on Parsing Technologies*, pages 209–212, Beijing, China, 2001.
15. L. Ciortuz. On compilation of the Quick-Check filter for feature structure unification. In *Proceedings of the IWPT 2001 International Workshop on Parsing Technologies*, pages 90–100, Beijing, China, 2001.
16. A. Copestake. *The (new) LKB system*. CSLI, Stanford University, 1999.
17. A. Copestake. *Implementing typed feature structure grammars*. CSLI Publications, Stanford, CA, 2002.
18. 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.
19. D. Gerdemann. Term encoding of typed feature structures. In *Proceedings of the 4th International Workshop on Parsing Technologies*, pages 89–97, Prague, Czech Republik, 1995.
20. J. Jaffar and M. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19(20):503–582, May-July 1994.
21. M. Kay. Head driven parsing. In *Proceedings of the 1st Workshop on Parsing Technologies*, pages 52–62, Pittsburg, 1989.
22. B. Kiefer, H-U. Krieger, J. Carroll, and R. Malouf. A bag of useful techniques for efficient and robust parsing. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics*, pages 473–480, 1999.
23. 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.
24. 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.
25. 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.
26. Y. Miyao, T. Makino, K. Torisawa, and J. Tsujii. The LiLFeS abstract machine and its evaluation with the LinGO grammar. *Journal of Natural Language Engineering*, 6 (1) (Special Issue on Efficient Processing with HPSG):47–61, 2000.
27. S. Oepen and J. Carroll. Performance profiling for parser engineering. *Journal of Natural Language Engineering*, 6 (1) (Special Issue on Efficient Processing with HPSG: Methods, Systems, Evaluation):81–97, 2000.
28. 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.
29. C. Pollard and I. Sag. *Head-driven Phrase Structure Grammar*. CSLI Publications, Stanford, 1994.
30. N. Sikkel. *Parsing Schemata*. Springer Verlag, 1997.
31. H. Tomabechi. Quasi-destructive graph unification. In *Proceedings of the 29th meeting of the Association for Computational Linguistics*, pages 315–322, Berkeley, California, 1991. Association for Computational Linguistics.
32. D.H.D. Warren. An abstract Prolog instruction set. Technical report, SRI International, Menlo Park, CA, 1983. Technical Note 309.
33. S. Wintner. *An Abstract Machine for Unification Grammars*. PhD thesis, Technion – Israel Institute of Technology, Haifa, Israel, 1997.
34. S. Wintner and N. Francez. Efficient implementation of unification-based grammars. *Journal of Language and Computation*, 1(1):53–92, 1999.