LIGHT — a constraint language and compiler system for typed-unification grammars

Liviu Ciortuz
CS Department, University of York, UK. E-mail: ciortuz@cs.york.ac.uk

Abstract. This work presents LIGHT, a feature constraint language for deduction-based bottom-up parsing with typed-unification grammars. We overview both its formal definition, as a logic language operating bottom-up inferences over OSF-terms, and its implementation — an elegant combination of a virtual machine for head-corner parsing and an extended abstract machine for feature structure unification.[1]

1 Introduction

Interest in (typed) unification grammars for Natural Language Processing can be traced back to the seminal work on the PATR-II system [32]. Since then, the *logics* of feature constraints were studied and became well-understood [36] [7], and different types of unification-based *grammar formalisms* were developed by the Computational Linguistics community — most notably Lexical Functional Grammars (LFG, [19]), Head-driven Phrase Structure Grammars (HPSG, [31]) and Categorial (unification) Grammars [39]. More recently *large-scale implementations* for such grammars were devised.

So are for instance the HPSG for English [17] developed at Stanford (called LinGO), two HPSGs for Japanese developed at Tokyo University [24], and respectively at DFKI–Saarbrücken, Germany [34], and the HPSG for German, developed also at DFKI [26]. LFG large-scale grammars were developed by Xerox Corp., but they are not publicly available.

The last years witnessed important advances in the elaboration of techniques for efficient unification and parsing with such grammars [29] [30]. The work here presented was part of this international, concerted effort.

The LIGHT language is the only one apart TDL [22] — which proved too much expressive, and therefore less efficient — to be formally proposed for (the systems processing) large-scale LinGO-like grammars. TDL was implemented by the PAGE platform at DFKI — Saarbrücken and the LKB system [16] at CSLI, University of Stanford. They are both interpreters, written in Lisp. LIGHT may be seen as redefining implementing the subset of TDL used in the current versions of LinGO. The LIGHT compiler is one of the (only) two compilers running LinGO, the large-scale HPSG grammar for English. (The other compiler running LinGO is LiLFeS [25]. It basically extends the Prolog unification mechanism to typed feature structures.)

There are already a number of papers published about different unification or parsing issues involved in the LIGHT system: [13] [9] [11] [10] [12]. However, none

[1] This paper appeared in the Proceedings of the 25th German Conference on Artificial Intelligence (KI2002), Aachen, Germany, 16-20 September, 2002, published as LNAI vol. 2479, Springer-Verlag, pages 3–17.

of them gave until now a thorough overview of the LIGHT language and system. Besides, we consider that the possibilities for improving the LIGHT system's implementation are far from exhaustion. This is why we found appropriate to publish this overview work about LIGHT.

LIGHT has a fairly elegant logic design in the framework of OSF-logic [4]. The LIGHT name is an acronym for Logic, Inheritance, Grammars, Heads, and Types.[2] Distinctive from the other systems processing LinGO-like grammars, LIGHT uses OSF-theory unification [5] on order- and type-consistent theories. This class of OSF-logic theories extends well-typed systems of feature structures [7], the framework of the other systems processing LinGO. Deduction in LIGHT is the head-corner generalisation [20] [35] of bottom-up chart-based parsing-oriented deduction [33].[3]

LIGHT is implemented in C and compiles the input grammar into C code. The implementation is based on a unique combination of an abstract machine for OSF-theory unification — which extends the AM for OSF-term unification [3][13] — and a virtual machine for active bottom-up chart-based parsing [15] [10]. The interface between the two machines is structure sharing-oriented. LIGHT uses compiled Quick-Ckeck to speed-up unification [11] [23], and a learning module to reduce the size (i.e., the number of constraints) in the rule feature structures [12].

Concerning the organisation of this paper, Section 2 presents the logical background of order- and type-consistent OSF-theories, Section 3 introduces the formal specifications of the LIGHT language, exemplifying the main notions with excerpts from an HPSG-like typed-unification grammar, and Section 4 overviews the implementation of the LIGHT compiler and parsing system.

2 LIGHT logical background: order- and type-consistent OSF-theories

Aït-Kaci, Podelski and Goldstein have introduced in [5] the notion of OSF-theory unification which generalise both OSF-term (ψ-term) unification and well-formed typed feature structure unification [7]. The class of order- and type-consistent OSF-theories introduced in this section extends that of systems of well-formed typed feature structures, henceforth called *well-typed feature structures*. Systems of well-typed feature structures will correspond in fact to order- and type-consistent OSF-theories satisfying a set of appropriateness constraints concerning every feature's value and domain of definition.

[2] The analogy with the name of LIFE — Logic, Inheritance, Functions and Equalities — a well-known constraint logic language based on the OSF constraint system [4] is evident.

[3] For non-lexicalized typed-unification grammars, top-down inferences can be conveniently defined, and in this case LIGHT would be seen as a particular CLP(OSF) language. (For the CLP schema see [18].) It is in fact in this way that LIGHT was first implemented, before it was converted to parsing with large-scale lexicalized unification grammars.

The advantage of generalising (and subsequently simplifying) the logical framework of typed-unification grammars (from well-typed FSs) to order- and type-consistent OSF-theories is reflected on LIGHT's implementation side by the reduction of the expanded form of LinGO with 42%, and the improving of the average parsing time for the sentences in the *csli* test-suite with 20%.[4]

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 normalised clause) over the sort signature Σ are like in the OSF constraint logic theory [5]. The same is true for unfolding an OSF-term, and also for subsumption (denoted as $\sqsubseteq$), and unification of two ψ-terms.

Some *notations* to be used in the sequel: $root(\psi)$ 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 the relation $\prec$ will be denoted as $\preceq$.

$Form(\psi, X)$, the *logical form* associated to an OSF-term ψ of the form $s[f_1 \rightarrow \psi_1, ..., f_n \rightarrow \psi_n]$ is $\exists X_1 ... \exists X_n((X.f_1 \doteq Form(\psi_1, X_1) \wedge ... \wedge X.f_n \doteq Form(\psi_n, X_n)) \leftarrow X : s)$, 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 $s, t \in \mathcal{S}$, if $s \neq t$ then $\Psi(s)$ 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 generalised 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.

Example 1. Let us consider a sort signature in which $b \wedge c = d$, and the symbol + is a subsort of *bool*, and the OSF-terms $\psi_1 =$ a[FEAT1 $\rightarrow$ b], $\psi_2 =$ a[FEAT1 $\rightarrow$ c[FEAT2 $\rightarrow$ *bool*]]. The glb of ψ_1 and ψ_2 — i.e., their *OSF-term unification* result — will be $\psi_3 =$ a[FEAT1 $\rightarrow$ d[FEAT2 $\rightarrow$ *bool*]]. Taking $\Psi(d) =$ d[FEAT2 $\rightarrow$ +], the glb of ψ_1 and ψ_2 relative to the $\{\Psi(d)\}$ OSF-theory — i.e., their *OSF-theory unification* result — will be $\psi_4 =$ a[FEAT1 $\rightarrow$ d[FEAT2 $\rightarrow$ +]].

Now we can formalise the link towards well-typed feature structures:

As defined in [5], 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.

[4] These numbers were (computed after data) obtained and published by U. Callmeier, a former contributor to the LIGHT project [6].

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 $\{\Psi(s)\}_{s \in \mathcal{S}}$.

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.[5] The appropriate domain and values for all features $f \in \mathcal{F}$ define the "canonical" *appropriateness constraints* for a well-typed OSF-theory.

2. 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 $s \preceq t$;

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 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.

To summarise:

The first difference between the class of order- and type-consistent OSF-theories and the class of well-typed OSF-theories concerns the subsumption condition — limited to non-atomic substructures ψ: if $root(\psi) = s$, then $\psi \sqsubseteq \Psi(s)$. For instance, if a[F → cons] is type-consistent, its well-typed correspondent is a[F → cons[FIRST → top, REST → list]].

This (more relaxed) condition has been proved to be beneficial for LinGO-like grammars [6], since it lead to a both significant reduction of the expanded size of the grammar and the parsing time (due to reduction of copying and other structure manipulation operations), without needing a stronger notion of unification.

[5] If the lub (least upper bound) of any two sorts exists and is unique, our current implementation of LIGHT uses a weaker version for the condition *i*: if $f \in$ *Arity(s)* $\wedge$ $f \in$ *Arity(t)*, and $\neg\exists u \in \mathcal{S}$ such that $s \preceq u$, $t \preceq u$ and $f \in$ *Arity(u)* then $ApropDom(f) = \text{lub}(s, t)$, and $ApropVal(f) = \text{lub}(root(s.f), root(t.f))$.

Attribute subtyping: $\left.\begin{array}{l} Approp(a,s) = t \\ s' \prec s \end{array}\right\}$ then $Approp(a,s') = t'$, and $t' \preceq t$.

Attribute unique introduction: $\forall f \in \mathcal{F}, \exists^{\star} s \in \mathcal{S}$ and $t.f \uparrow$ for any $t \in \mathcal{S}, s \prec t$.

Type (strict-)consistency: $\left.\begin{array}{l} \psi \text{ is a subtype in } \Psi, \\ root(\psi) = s \end{array}\right\}$ then $\psi \sqsubseteq \Psi(s)$.

Fig. 1. The type discipline in well-typed unification grammars.

Order consistency: $s' \prec s$ then $\Psi(s') \sqsubseteq \Psi(s)$.

Type consistency: $\left\{\begin{array}{l} \psi \text{ is a } \textit{non-atomic} \text{ subtype in } \Psi, \\ root(\psi) = s \end{array}\right\}$ then $\psi \sqsubseteq \Psi(s)$.

Fig. 2. The type/FS discipline in order- and typed-consistent grammars.

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 $\Psi(s)$.

For instance, if ψ_5 = a[FEAT1 $\rightarrow$ d[FEAT2 $\rightarrow$ +, FEAT3 $\rightarrow$ *bool*]], then the OSF-theory glb of ψ_2 and ψ_5 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 works with free-order registration of features inside feature frames.[6]

Just to get a comparative overview on 1. well-typed unification grammars [7] versus 2. order-and type-consistent OSF-theories/grammars, we synthesise the definitions for 1. the appropriateness constraints in Figure 1, and respectively 2. the order- and type-consistency notions in Figure 2. It is obvious that our approach is simpler than introduced in [7].

A procedure — called *expansion* — for automate transformation of an OSF-theory into an order- and type-consistent one was presented in [9]. This procedure also computes the "canonical" set of appropriate constraints associated to the expanded theory. OSF-theory unification for an order- and type-consistent theory is well-formed with respect to this canonical set of appropriate constraints.

[6] 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. Recent work by Callmeier has shown that fix-order feature storing does not lead to improvement of the parse performance on LinGO.

With respect to the canonical appropriateness constraints induced for an order- and type-consistent OSF-theory (in particular, for LinGO), well-typed feature structure unification coincides with OSF-theory unification.

3 Formal introduction to LIGHT

3.1 Constraint logic definitions

A LIGHT logic grammar will be defined as a particular order- and type-consistent OSF-theory. (For an introduction to logic grammars see [1].)

Let $< \mathcal{S}, \mathcal{F}, \prec >$ be a sort signature, with $\mathcal{S}$ containing several "reserved" sorts, namely *top, sign, rule-sign, lexical-sign* and *start*, and $\mathcal{F}$ the "reserved" features STEM and ARGS. The sorts *rule-sign* and *lexical-sign* — both being descendants of *sign* — are disjunct, while *start* is a descendant of *rule-sign*. $\mathcal{S}$ is assumed a lower semi-lattice w.r.t. $\prec$.

- A LIGHT Horn clause, denoted $\psi_0 :- \psi_1 \ \psi_2 \ ... \ \psi_n$ $(n \geq 0)$, corresponds to the logical formula $\tilde{\forall}(\psi_0 \leftarrow \psi_1 \wedge \psi_2 \wedge ... \wedge \psi_n)$, where $\psi_i, i = 0, 1, 2, ..., n$ are ψ-terms, $root(\psi_0) \preceq$ *rule-sign*, and $root(\psi_i) \preceq$ *rule-sign* or $root(\psi_i) \preceq$ *lexical-sign* for $i = 1, 2, ..., n$. (Remark the absence of predicate symbols.)
- A LIGHT *rule* is a LIGHT Horn clause with $root(\psi_0)$ a leaf node in the sort hierarchy $(\mathcal{S}, \prec)$.[7] (The STEM feature in rules' definition is related to the consumed input sentence.)

Remark: 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.

- A LIGHT *logic grammar* is an order- and type-consistent OSF-theory containing a non-empty set of LIGHT rules. (In practice we require in fact that all leaf *rule-sign*-descendants types be LIGHT rules.)

3.2 Parsing-oriented derivation specific definitions

For bottom-up chart-based parsing, [33] and [35] propose two derivation rules: *scan* and *complete*. Head-corner parsing [35] distinguishes between *left*- and *right*-scan and respectively complete, and adds a new derivation rule, *head-corner*. At one rule's application, unification with the head-corner argument, designated by the grammar writer, is tried before the other arguments are treated. 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.

[7] For the moment, the LIGHT language is not designed to deal with so-called ϵ-rules, i.e., rules whose right hand side (RHS) is empty.

It is worth to note that in a (order-)sorted framework, the distinction between terminal and non-terminal symbols is erased, since either a terminal or a non-terminal may occupy a same slot in the ARGS list of a rule. 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. [8]

Lexical analysis:

Let $< w_1 w_2 \ ... \ w_n >$ be an input (word) sequence. If ψ is a leaf *lexical-sign*-descendant in the grammar $\mathcal{G}$ such that ψ.STEM $= < w_i w_{i+1} \ ... \ w_j >$, with $1 \leq i \leq j \leq n$, then the quadruple $(\epsilon, \psi', i-1, j)$, where ψ' is obtained from ψ by renaming all its variables with new variables, is a *lexical item*. Here, ϵ is the empty set (of atomic OSF-constraints). Any lexical item is a *passive* item.

In LinGO, the lexical analysis is slightly more elaborated: a *lexical-rule*, which is a leaf descendant of the *lexical-rule* sort (disjunct from *rule-sign*) may be applied during the morphological analysis to a leaf *lexical-sign*-descendant. The resulting feature structure takes the role of the above ψ'.

Syntactic analysis:

Head-corner: If (σ, ψ, i, j) is a passive item, $\psi_0 \ :- \ \psi_1 \ ... \ \psi_r$ is a newly renamed instance of a rule in $\mathcal{G}$, and ψ_k is the head/key argument of this rule $(1 \leq k \leq r)$, then if there is a glb φ of ψ_k and ψ, $(1 \leq k \leq r)$ with τ the subsumption substitution of ψ_k into $\varphi = \text{glb}(\psi_k, \psi)$, i.e., $\tau\psi_k = \text{glb}(\psi_k, \psi)$, then $(\tau\sigma, \psi_0 :- \psi_1 \ ... \ .\psi_k. \ ... \ \psi_r, i, j)$ is an item. It is an *active* item if $r > 1$, and a passive item if $r = 1$. The compositions $\tau\sigma$ and $\sigma\psi$ must be interpreted simply as a conjunction/union of the formulae/sets of atomic OSF-constraints τ and σ, respectively σ and ψ.

Right complete: If $(\sigma, \psi_0 :- \psi_1 \ ... \ .\psi_p \ ... \ \psi_q. \ ... \ \psi_r, i, j)$ is an active item, with $q < r$, and there is a passive item, either a lexical (τ, ψ, j, k) or a non-lexical one $(\tau, \psi :- .\psi'_1 \ ... \ \psi'_m., j, k)$, assuming that $\text{glb}(\tau\psi, \sigma\psi_{q+1})$ exists, and υ is the corresponding subsumption substitution, then $(\upsilon\sigma, \psi_0 :- \psi_1 \ ... \ .\psi_p \ ... \ \psi_{q+1}. \ ... \ \psi_r, i, k)$ is an item, namely a passive one if $p = 1$ and $q + 1 = r$, respectively an active one, otherwise.

Left complete: Similar to the above definition, except that the active item is leftward "open". If $(\sigma, \psi_0 :- \psi_1 \ ... \ .\psi_p \ ... \ \psi_q. \ ... \ \psi_r, i, j)$ is an active item, with $1 < p$, and there is a passive item, either a lexical (τ, ψ, k, i) or a non-lexical one $(\tau, \psi :- .\psi'_1 \ ... \ \psi'_m., k, i)$, assuming that $\text{glb}(\tau\psi, \sigma\psi_{q+1})$ exists, and υ is the corresponding subsumption substitution, then $(\upsilon\sigma, \psi_0 :- \psi_1 \ ... \ .\psi_{p-1} \ ... \ \psi_q. \ ... \ \psi_r, k, j)$ is an item, namely a passive one if $p - 1 = 1$ and $q = r$, respectively an active one, otherwise.

A LIGHT *parse sequence* (or *derivation*) is a finite sequence of items $i_1, ..., i_m$ such that for every r with $1 \leq r \leq m$, the item i_r is either a lexical item, or a head-corner production based on a passive item i_q with $q < r$, or it is obtained

[8] LinGO uses only binary rules, therefore it became convenient to use the name *complete* as a "rewriting" one for both the classical scan and complete parsing operations, when extended to order-sorted unification based-parsing.

```
satisfy_HPSG_principles
[ STEM   diff_list,
  CAT    #1:categ,
  SUBCAT #2:categ_list,
  HEAD   #4:phrase_or_word
            [ STEM   diff_list,
              CAT #1,
              SUBCAT #3|#2 ],
  COMP   #5:phrase_or_word
            [ STEM   diff_list,
              CAT #3,
              SUBCAT nil ],
  ARGS   <#4, #5> ]
```

```
girl
[ STEM   <!"girl"!>,
  CAT    noun,
  SUBCAT <det> ].
```

Fig. 3. A sample rule (parent) type (satisfy_HPSG_principles) and a lexical entry (girl).

by left- or right- completion of an active item i_p with a 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 $\mathcal{G}$ *recognises* the input sentence w iff there is a parse sequence (derivation) in $\mathcal{G}$ ending with a passive item $(\sigma, \psi, 0, n)$, where ψ is *start*-sorted.

3.3 Exemplification:

The satisfy_HPSG_principles feature structure adapted from [37] and presented in Figure 3 encodes the Head Feature Principle (CAT $\doteq$ HEAD.CAT), The Saturation Principle (COMP.SUBCAT:nil), and the Subcategorization Principle (to be later detailed). These three principles are among the basic ones in the HPSG linguistic theory [31]. The syntax used in Figure 3 is that of LIGHT:

The sorts start, list and the list subsorts cons and nil are special (reserved LIGHT) types, and so is diff_list, the difference list sort. The notation <! !> is a syntax sugar for difference lists, just as < > is used for lists.[9] The symbol | is used as a constructor for non-nil (i.e., cons) lists.

The linguistic significance of the Head Feature Principle: the CATegory of a phrase is that of (or: given by) its head. For the Saturation Principle: the complement of a phrase must be saturated — procedurally: it must have been fully parsed — at the moment of its integration into a (larger) phrase. The Subcategorization Principle correlates the SUBCAT feature value — namely a list of categories — for the HEAD argument with the SUBCAT feature value of the phrase itself, and the CAT feature value for the complement (COMP) daughter of the same phrase.

[9] Formally, $<!a_1, a_2, ..., a_n!>$ stands for

diff_list[FIRST_LIST $a_1|a_2|...|a_n|\#1$,
REST_LIST $\#1$]

The Subcategorization Principle is directly linked to the nature of the parsing itself in such lexicalized typed-unification grammars: no specific parsing rules are provided; only very general principles/constraints are given about how phrases or words may be combined, while specific information about the combinatorial valances of words is provided in the lexical descriptions.

Basically, the Subcategorization Principle says that, when applying a rule — in order to build a *mother* feature structure out of a *head daughter* and a *complement daughter* —, the complement daughter "consumes" the first element of the head daughter's SUBCAT list, and "transmits" the rest of that list to the mother feature structure.

The satisfy_HPSG_principles type will transmit via inheritance the constraints it incorporates to the rules used in the [37] grammar, namely two binary rules: lh_phrase and rh_phrase. The constraints specific to these rules will impose only that the head is on the left, respectively the right position inside a phrase.

4 LIGHT — implementation

4.1 Related systems

Systems processing typed-unification grammars opted for different approaches. ALE [8] opted for the translation of typed FSs into Prolog terms. $\mathcal{A}$MALIA [40] [41] offered the first view on compiling ALE grammars — in which typed FSs had a less general form than in the LinGO-like HPSG grammars —, based on an abstract machine which adapts WAM [2] to FS unification and replaces SLD-resolution with simple bottom-up chart-based parsing. LiLFeS [25], the other compiler (besides LIGHT) running LinGO followed in the beginning the same direction, but then processed the grammar rules so to apply well-known CFG parsing algorithms. (Later developments for TDL also explored CFG approximations of LinGO-like grammars [21].) The LKB[16], TDL [22] systems' basic versions, and PET [6] implemented simple head-corner parsers, opting for different versions of FS unifiers [38] [42] [23]. Later, hyper-active parsing [28] and ambiguity packing [27] were incorporated into LKB and PET.

For LIGHT, we chose to extend the AM for unification of OSF-terms [3] to OSF-theory unification [5], making it implicitly able to unify typed FSs [7]. For parsing, while originally the LKB simple head-corner active bottom-up chart-based parser was imported in our system, later on we replaced it with a VM for incremental head-corner bottom-up parsing with FS sharing and backtracking [15]. The interesting point about this VM is that on one hand it can be used by the LKB-like systems (if they would like to opt for FS sharing), and on the other hand it can work as an interpreter of the abstract code (for rules) generated by the LIGHT compiler.

4.2 LIGHT system's architecture

An overview of our LIGHT parser's architecture is shown in Figure 4. The LIGHT compiler translates typed-unification grammars (written or converted into a

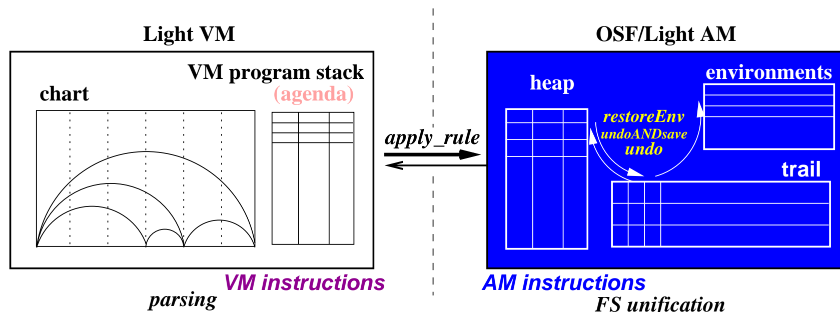

Fig. 4. An overview of the VM for HC bottom-up chart-based parsing.

format inspired by the OSF notation) into abstract code, and then into C. Each typed FS in the input grammar gets an associated (compiled) C function. The FSs representing rules undergo a further, automate transformation of the abstract code so to make them suitable for efficient head-corner bottom-up chart-based parsing with FS sharing. We refer to this transformation scheme as Specialised Rule Compilation (SRC) of rules [10].[10]

Each of the basic operations for parsing in LIGHT — a rule application, a lexical FS construction, or a type-checking (we call it: on-line expansion) — are achieved by calling a compiled function. The abstract machine instructions which build up these functions are shown in the right side of the table in Figure 5.

As the AM in [3] was concerned only with OSF-term unification — this AM will be referred to in the sequel as OSF AM —, we extended the definitions of the two of its AM instructions, namely intersect_sort and test_feature, and also the *bind-refine* procedure invoked by the on-line unifier function *osf-unify* in [3], in order to support on-line expansion/type-checking needed for OSF-theory unification. Besides these transformations, we incorporated into OSF AM some other facilities: tracing, backtracking, and FS sharing. We use the name OSF/LIGHT AM to denote the new, more general version of OSF AM. For full details our work extending OSF AM to OSF/LIGHT AM, the interested reader must refer to [13]. OSF AM and consequently OSF/LIGHT AM imported the two-stream (*READ* and *WRITE*) optimisation from the Warren Abstract Machine [2]. Thus the set_sort and intersect_sort abstract instructions correspond to sort constraints, set_feature and test_feature correspond to feature constraints, and unify_feature corresponds to equation constraints.

[10] The execution of the abstract code for rules in LIGHT has an effect similar to that produced by $\mathcal{A}$MALIA on ALE grammars, namely it achieves bottom-up chart-based parsing. Unlike $\mathcal{A}$MALIA, we produce this ("specialised") code through automate transformation of the (non "specialised") abstract code produced for the FS representing the rule [10]. (Note that rules in LinGO are represented as FSs; their arguments constitute the value of the ARGS reserved feature/attribute.) Additionally, the SRC-optimised code in LIGHT incorporates specialised sequences for dealing with environments, for feature structure sharing.

VM Instructions		*AM Instructions*	
parsing	*interface*	*READ*-stream	*WRITE*-stream
keyCorner	undo	push_cell	intersect_sort
directComplete	saveEnvironment	set_sort	test_feature
reverseComplete	restoreEnvironment	set_feature	unify_feature
apply_rule		write_test	

Fig. 5. Instructions in LIGHT VM and OSF/LIGHT AM.

The main data structures in the OSF/LIGHT AM are a heap for building up FS representation, a trail which registers the modifications on the heap during unification-achieving operations, and an array of environments used for FS sharing.

The parsing (i.e., the control above rules' application) is implemented in the LIGHT system as a virtual machine hereby referred to as LIGHT VM. The main data structures in LIGHT VM are a chart and the VM's program stack, which replaces in a quite elaborated way the well-known *agenda* in chart-based parsing.

Other data structures in LIGHT VM:
− the rule (syntactic and lexical) *filters*,
− the *dictionary*, which associates to every word which is a *root* in at least one lexical entry the list of all lexical entries for which that word is the root,
− the *lexicon*, which associates every lexical entries the index of a *query* (compiled) function and eventually the index of a lexical rule to be applied to the FS constructed by that query function,
− the pre-compiled *QC-arrays* associated to rules [11].

The LIGHT VM's instructions are listed in the first column of the table in Figure 5. (The interested reader will find their detailed description in [15].)

The apply_rule function appearing on the bottom of the first column in Figure 5 is not a VM instruction (this is why we delimited it from above by a horizontal line). It is called by each of the three VM parsing instructions — keyCorner, directComplete, and reverseComplete. Conceptually, it acts in fact like a higher level function, which applies the compiled function corresponding to a (specified) rule FS to a certain argument.

Instead, apply_rule is part of the *interface* between the LIGHT VM and the OSF/LIGHT AM. This interface contains also three procedures implemented within the unifier (in our case: OSF/LIGHT AM), which will be applied/called from within the VM program: undo, saveEnvironment and restoreEnvironment. The undo procedure performs backtracking to a certain state of the AM (namely as it was before an unification attempt), restoring the information corresponding to a successful unification (according to FS sharing scheme). The saveEnvironment procedure performs the same task as undo, but also moves a certain amount of data from the trail to a new environment, while restoreEnvironment performs the opposite operation.

Overall:	memory consumption	process size full/resident	average parsing time
regular compilation	59.5MB	73MB/80MB	128 msec
specialised compilation	3.9MB	44MB/13MB	35 msec

Detailed:	heap cells	feature frames	environ-ments	trail cells	corefe-rences
regular compilation	1,915,608	1,050,777	2669	128,747	0
specialised compilation	77,060	57,663	2669	77,454	22,523
GR-optimisation	37,915	29,908	2669	44,424	12,137

Fig. 6. Regular vs. specialised rule compilation: a comparison between the respective effects on parsing the *csli* test-suite. (Further memory reduction due to GR is added.)

4.3 Final notes on LIGHT implementation

Both the OSF/LIGHT AM for unification and the LIGHT VM for head-corner parsing can be used separately, namely just for unifying FSs, or respectively to achieve parsing using another unifier (than the OSF/LIGHT AM). Two parsers co-exist in the LIGHT system, one corresponding to the Specialised Rule Compilation (SRC) optimisation, the other using as unification means only the *osf-unify* procedure [3] upgraded with type-checking/on-line expansion.[11] The two parsers are implemented in (or as instances of) the same VM, just by changing the definition of the higher-level function rule.

The speed-up factor provided by implementing the SRC optimisation is 3.66 when running the LinGO grammar on the *csli* test-suite. A subsequent 43% speed-up was provided the compilation of the QC pre-unification filter [11], while recently, the Generalised Reduction (GR) learning technique working in connection with two-step unification further sped-up parsing up to 23% [12]. LIGHT's current best parsing time was an average of 18.4 msec. per sentence on the *csli* test-suite, registered on a Pentium III PC at 933MHz running Red Hat Linux 7.1. The compilation of the LinGO grammar (the *CLE* version) on that computer takes 79 seconds, including the filter computation (which is the most consuming task in grammar processing). [12]

Our SRC-optimisation also reduced dramatically the memory space used during parsing, as one can see in Figure 6.

[11] Therefore the second parser does not appeal the real power of the compiler; it serves for basic tests, and — most important — as a support for developing new extensions and improved versions of the LIGHT compiler.

[12] By the time LIGHT system's development was frozen due to the author's departure from DFKI – Saabrbrücken (at the end of the year 2000), the LIGHT parsing speed has been proved slightly higher than that of the fastest (and since then, commercially supported) interpreter for LinGO-like grammars — namely the PET system [6].

Conclusion and further work

This paper presented LIGHT, a feature constraint language in the framework of OSF-logic, which underlines large-scale typed-unification grammars. Until now, LIGHT was used for parsing with LinGO [17], the HPSG grammar for English developed at CSLI, University of Stanford, and for automate learning of typed-unification grammars [14].

The implementation of LIGHT is based on an interesting combination of a virtual machine for head-corner bottom-up chart-based parsing and an abstract machine for (typed) FS unification. Other differences with respect to the other systems implementing LinGO-like typed-unification grammars are environment-based FS sharing, incremental parsing and compiled Quick-Check [23].

We expect that further major improvements to the LIGHT compiler will come following the finding (on the LinGO grammar) that there are usually a reduced number of paths (the so-called *GR-paths*) that contribute to unification failure [12]. This fact may be further exploited so to *i.* fully integrate QC into compiled unification; *ii.* eliminate the need for the — much time-consuming — FS restoring operation, currently part of the FS sharing mechanism; and *iii.* replace searching through feature frames with direct access into vectors of GR-path values associated to each rule.

Independent of the developments suggested above, further fine engineering the LIGHT system — for instance making the computations be done as locally as possible — is supposed to significantly improve the current performances. Finally, we expect that the system can be adapted to run other kind of applications, like deductive frame-oriented databases and ontologies.

Acknowledgements: The LIGHT system was developed at the Language Technology Laboratory of The German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany. U. Callmeier contributed to the implementation of the head-corner bottom-up (non VM-based) parser for CHIC, which was the development prototype of LIGHT. The present paper was written while the author was supported by an EPSRC ROPA grant at the University of York.

References

1. H. Abramson and V. Dahl. *Logic Grammars.* Symbolic Computation AI Series. Springer-Verlag, 1989.
2. H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, Cambridge, MA, 1991.
3. H. Aït-Kaci and R. Di Cosmo. Compiling order-sorted feature term unification. Technical report, Digital Paris Research Laboratory, 1993.
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. *Journal of Logic, Language and Information*, 30:99–124, 1997.

6. 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.
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. 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.
10. 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.
11. 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.
12. L. Ciortuz. Learning attribute values in typed-unification grammars: On generalised rule reduction. In *Proceedings of the 6th Conference on Natural Language Learning (CoNLL–2002)*, Taipei, Taiwan, 2002. Morgan Kaufmann Publishers and ACL.
13. L. Ciortuz. LIGHT – another abstract machine for feature structure unification. In D. Flickinger, S. Oepen, J. Tsujii, and H. Uszkoreit, editors, *Collaborative Language Engineering*. CSLI Publications, The Center for studies of Language, Logic and Information, Stanford University, 2002.
14. L. Ciortuz. Towards inductive learning of typed-unification grammars. In *Proceedings of the 17th Workshop on Logic Programming*, 2002. Dresden Technical University.
15. L. Ciortuz. A virtual machine design for head-corner parsing with feature structure sharing. Submitted for publication, 2002.
16. A. Copestake. *The (new) LKB system*. CSLI, Stanford University, 1999.
17. 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.
18. J. Jaffar and M. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19(20):503–582, May-July 1994.
19. R. M. Kaplan and J. Bresnan. Lexical-functional grammar: A formal system for grammatical representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages 173–381. MIT Press, 1983.
20. M. Kay. Head driven parsing. In *Proceedings of the 1st Workshop on Parsing Technologies*, pages 52–62, Pittsburg, 1989.
21. B. Kiefer and H-U. Krieger. A context-free approximation of Head-driven Phrase Structure Grammar. In *Proceedings of the 6th International Workshop on Parsing Technologies*, pages 135–146, Trento, Italy, 2000.
22. H.-U. Krieger and U. Schäfer. TDL – A Type Description Language for HPSG. Research Report RR-94-37, The German Research Center for Artificial Intelligence (DFKI), 1994.

23. 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.
24. Y. Mitsuishi, K. Torisawa, and J. Tsujii. HPSG-Style Underspecified Japanese Grammar with Wide Coverage. In *Proceedings of the 17th International Conference on Computational Linguistics: COLING-98*, pages 867–880, 1998.
25. 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.
26. Stefan Müller. *Deutsche Syntax deklarativ. Head-Driven Phrase Structure Grammar für das Deutsche.* Number 394 in Linguistische Arbeiten. Max Niemeyer Verlag, Tübingen, 1999.
27. S. Oepen and J. Carroll. Ambiguity packing in HPSG — practical results. In *Proceedings of the 1st Conference of the North American Chapter of the ACL*, pages 162–169, Seattle, WA, 2000.
28. 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.
29. S. Oepen, D. Flickinger, H. Uszkoreit, and J. Tsujii, editors. *Special Issue on Efficient Processing with HPSG: Methods, Systems, Evaluation.* Cambridge University Press, 2000. Journal of Natural Language Engineering, 6 (1).
30. S. Oepen, D. Flickinger, H. Uszkoreit, and J. Tsujii, editors. *Collaborative Language Engineering.* CSLI Publications, University of Stanford, CA, 2002.
31. C. Pollard and I. Sag. *Head-driven Phrase Structure Grammar.* CSLI Publications, Stanford, 1994.
32. S. M. Shieber, H. Uszkoreit, F. C. Pereira, J. Robinson, and M. Tyson. The formalism and implementation of PATR-II. In J. Bresnan, editor, *Research on Interactive Acquisition and Use of Knowledge.* SRI International, Menlo Park, Calif., 1983.
33. S.M. Shieber, Y. Schabes, and F. Pereira. Principles and implementation of deductive parsing. *Jornal of Logic Programming*, pages 3–36, 1995.
34. M. Siegel. HPSG analysis of Japanese. In *Verbmobil: Foundations of Speech-to-Speech Translation*, pages 264–279. Springer Verlag, 2000.
35. N. Sikkel. *Parsing Schemata.* Springer Verlag, 1997.
36. G. Smolka. Feature-constraint logics for unification grammars. *Journal of Logic Programming*, 12:51–87, 1992.
37. G. Smolka and R. Treinen, editors. *The DFKI Oz documentation series.* German Research Center for Artificail Intelligence (DFKI), Stuhlsatzenhausweg 3, Sarrbrücken, Germany, 1996.
38. H. Tomabechi. Quasi-destructive graph unification with structure-sharing. In *Proceedings of COLING-92*, pages 440–446, Nantes, France, 1992.
39. H. Uszkoreit. Categorial Unification Grammar. In *International Conference on Computational Linguistics (COLING'92)*, pages 498–504, Nancy, France, 1986.
40. S. Wintner. *An Abstract Machine for Unification Grammars.* PhD thesis, Technion – Israel Institute of Technology, Haifa, Israel, 1997.
41. S. Wintner and N. Francez. Efficient implementation of unification-based grammars. *Journal of Language and Computation*, 1(1):53–92, 1999.
42. D. A. Wroblewski. Non-destructive graph unification. In Dalle Miller, editor, *Proceedings of the 6th national conference on artificial intelligence (AAI'87)*, pages 582–587, Seattle, 1987.