Natural Language Engineering **6** (1): 47–61. Printed in the United Kingdom

The LiLFeS Abstract Machine and its Evaluation with the LinGO Grammar

YUSUKE MIYAO
Department of Information Science, Graduate School of Science, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033 Japan
e-mail: yusuke@is.s.u-tokyo.ac.jp

TAKAKI MAKINO
Department of Information Science, Graduate School of Science, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033 Japan
e-mail: mak@is.s.u-tokyo.ac.jp

KENTARO TORISAWA
Department of Information Science, Graduate School of Science, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033 Japan
Information and Human Behavior, PRESTO, Japan Science and Technology Corporation, 4-1-8 Kawaguchi Hon-cho, Kawaguchi-shi, Saitama 332-0012 Japan
e-mail: torisawa@is.s.u-tokyo.ac.jp

JUN-ICHI TSUJII
Department of Information Science, Graduate School of Science, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033 Japan
CCL, UMIST, P.O.Box 88, Manchester, M60 1QD, England
e-mail: tsujii@is.s.u-tokyo.ac.jp

(*Received 15 October 1999; revised 18 February 2000*)

Abstract

This article evaluates the efficiency of the LiLFeS abstract machine by performing parsing tasks with the LinGO English resource grammar. The instruction set of the abstract machine is optimized for efficient processing of definite clause programs and typed feature structures. LiLFeS also supports various tools required for efficient parsing (e.g. efficient copying, a built-in CFG parser) and the constructions of standard Prolog (e.g. cut, assertions, negation as failure). Several parsers and large-scale grammars, including the LinGO grammar, have been implemented in or ported to LiLFeS. Precise empirical results with the LinGO grammar are provided to allow comparison with other systems. The experimental results demonstrate the efficiency of the LiLFeS abstract machine.

1 Introduction

Efficient processing of feature structures is essential for efficient parsing with HPSG-based grammars, and also for practical applications that process real-world linguistic resources by using feature structures. While optimization methods specific to

HPSG parsing have proven to drastically increase parsing speed (Kiefer, Krieger, Carroll, & Malouf, 1999; Oepen & Carroll, this volume; Torisawa, Nishida, Miyao, and Tsujii, this volume), efficient feature structure processing is still required not only for efficient parsing but also to process parse results in various applications.

The LiLFeS system (Makino, Torisawa, & Tsujii, 1997) is a solution for providing a programming environment with efficient processing of feature structures. The LiLFeS language is an extension of Prolog for expressing typed feature structures instead of first order terms. Large-scale systems, not limited to HPSG grammars, can be easily developed on LiLFeS because feature structure descriptions and definite clause programs are consistently integrated. Several large-scale applications have already been developed on this system. Examples include wide-coverage Japanese and English grammars (Mitsuishi, Torisawa, & Tsujii, 1998; Tateisi, Torisawa, Miyao, & Tsujii, 1998), and a statistical disambiguation module for the Japanese grammar (Kanayama, Torisawa, Mitsuishi, & Tsujii, 1999).

The system's core engine is an abstract machine that can process feature structures and execute definite clause programs. In other abstract machine approaches, feature structure processing is separated from execution of programs such as parsers. On the other hand, the LiLFeS abstract machine increases processing speed by seamlessly processing feature structures and executing definite clause programs; it directly executes instructions compiled from the LiLFeS language. This approach also enables efficient low-level manipulation of feature structures, such as block copying and block equivalence checking.

The goal of this article is to evaluate the performance of the LiLFeS abstract machine and to explain how the abstract machine provides efficient processing of feature structures. The performance is evaluated with precise empirical results with the LinGO English resource grammar (Flickinger, this volume) to allow the comparison with other systems in this volume. The LinGO grammar is successfully translated to LiLFeS with the help of the LKB system (Copestake, 1992). This article also describes this translation process of the LinGO grammar and discusses the requirements for running the LinGO grammar on the LiLFeS system. Note that this article does not aim at describing the LiLFeS implementation in detail, which is exhaustively reported in other literature (Makino et al., 1997; Makino, Yoshida, Torisawa, & Tsujii, 1998; Makino, 1999).

Section 2 describes the design of LiLFeS and discusses the translation of the LinGO grammar into the LiLFeS language. Section 3 describes the advantages of the LiLFeS abstract machine architecture. Section 4 reports empirical results on the parsing performance of the abstract machine with the translated LinGO grammar. Section 5 introduces ongoing work aimed at further improvement in feature structure processing.

2 LiLFeS as a programming language

This section describes the LiLFeS language and the translation of the LinGO English resource grammar written in the $\mathcal{TDL}$ syntax (Uszkoreit et al., 1994). Since the formal definition of the LiLFeS language appears in another paper (Makino, 1999),

```
head <- [bot].
valence <- [bot] + [SUBJ\list, COMPS\list, SPR\list].
category <- [bot] + [HEAD\head, VAL\valence].
local <- [bot] + [CAT\category, CONT\bot].
synsem <- [bot] + [LOCAL\local, NONLOCAL\bot].
sign <- [bot] + [PHON\list, SYNSEM\synsem].
word <- [sign].
phrase <- [sign] + [HEAD_DTR\sign, NONHEAD_DTR\sign].
id_schema <- [pred].
head_feature_principle <- [pred].
lexical_entry <- [pred].
parse <- [pred].
parse_ <- [pred].
```
Type definitions

```
id_schema("head subject schema", $LEFT, $RIGHT, $HEAD, $NONHEAD, $MOTHER) :-
    $LEFT = $NONHEAD,
    $RIGHT = $HEAD,
    $MOTHER = (HEAD_DTR\($HEAD & SYNSEM\LOCAL\CAT\VAL\SUBJ\[$SYNSEM]) &
               NONHEAD_DTR\($NONHEAD & SYNSEM\$SYNSEM)).
head_feature_principle($HEAD_DTR, $MOTHER) :-
    $HEAD_DTR = SYNSEM\LOCAL\CAT\HEAD\$HEAD,
    $MOTHER = SYNSEM\LOCAL\CAT\HEAD\$HEAD.
parse_([$WORD|$TAIL], $TAIL, _, $LEXICON) :-
    lexical_entry($WORD, $LEXICON).
parse_($SENTENCE, $TAIL, [_|$LENGTH], $MOTHER) :-
    parse_($SENTENCE, $MID, $LENGTH, $LEFT),
    parse_($MID, $TAIL, $LENGTH, $RIGHT),
    id_schema($NAME, $LEFT, $RIGHT, $HEAD, $NONHEAD, $MOTHER),
    head_feature_principle($HEAD, $MOTHER).
parse($SENTENCE, $SIGN) :- parse_($SENTENCE, [], $SENTENCE, $SIGN).
```
Definite clause programs

Fig. 1. A sample program written in the LiLFeS language

here we mainly discuss the differences between the $\mathcal{TDL}$ and the LiLFeS languages. The problems and their solutions in the translation process are also discussed in this section.

2.1 The LiLFeS language

The LiLFeS language is a programming language to write definite clause programs with typed feature structures. It is similar to Prolog and has various expressions for both processing feature structures and describing procedures. Since typed feature structures can be used like first order terms in Prolog, the LiLFeS language can describe various kinds of application programs based on feature structures. Examples include HPSG parsers, HPSG-based grammars, and compilers from HPSG to CFG. Furthermore, other natural language processing systems can be easily developed because feature structure processing can be directly written in the LiLFeS language.

Figure 1 shows a sample LiLFeS program, which is a very simple parser and grammar. The LiLFeS language makes a clear distinction between type definitions (the upper section in Figure 1) and definite clause programs with feature structures (the lower section in Figure 1). A type (e.g. `phrase`) is defined by specifying supertypes (e.g. `sign`) and features (e.g. `HEAD_DTR\`, `NONHEAD_DTR\`) with their appropriate types (e.g. `sign`). After defining the types, we can use an instance of a feature structure as a first order term in the Prolog syntax. For example, in the predicate `head_feature_principle`, the `$HEAD_DTR` and `$MOTHER` are variables and supposed to be the structure of a sign defined in the type definition section. This predicate is called in the predicate `parse_` in order to apply the *Head Feature Principle* (Pollard & Sag, 1994). Having the same name variable indicates they are structure-shared.

```
head_only := unary_phrase &
             headed_phrase &
  [ HEAD-DTR #head &
      [ SYNSEM.LOCAL.CONJ cnil ],
    ARGS < #head > ].
```

$\Longrightarrow$

```
head_only <- [headed_phrase, unary_phrase]
  ./ constr\('HEAD-DTR'\($0 &
                 SYNSEM\LOCAL\CONJ\cnil) &
               ARGS\[$0]).
```

```
a_half := degree_spec_mle2 &
  [ STEM < "a", "half" >,
    SYNSEM.LOCAL.KEYS.KEY _a_half_rel ].
```

$\Longrightarrow$

```
lexical_entry("a half",
  degree_spec_mle2 &
  STEM\["a", "half"] &
  SYNSEM\LOCAL\KEYS\KEY\_a_half_rel).
```

Fig. 2. The translation of type definitions (above) and lexical entries (below) from the $\mathcal{TDL}$ syntax to the LiLFeS syntax

In `head_feature_principle`, the `HEAD` features of the `$HEAD_DTR` and `$MOTHER` are shared because they are indicated with the same variable `$HEAD`.

The differences between the LiLFeS language and the $\mathcal{TDL}$ are summarized in the following.

- The LiLFeS language expresses definite clause programs with typed feature structures, while the $\mathcal{TDL}$ is a typed feature structure description language.
- The LiLFeS language makes a clear distinction between type definitions and instances of typed feature structures, while the $\mathcal{TDL}$ does not distinguish them.

The former says that the description of typed feature structures in the LiLFeS language corresponds to the $\mathcal{TDL}$. Actually, the feature structure description of the LiLFeS language is a syntactic variant of the $\mathcal{TDL}$. Hence, the translation of typed feature structures from the $\mathcal{TDL}$ to the LiLFeS language is simply a syntactic conversion.

On the other hand, the latter indicates that type definitions require a more complex translation process. The type definition in the LiLFeS language follows *totally well-typed feature structures* (Carpenter, 1992). That is, LiLFeS presupposes that type hierarchies are well formed and all types must be defined with their appropriate features. In contrast, the $\mathcal{TDL}$ can describe type definitions which do not follow totally well-typedness, and in addition the LinGO grammar written in the $\mathcal{TDL}$ is based on a slightly different type system, as specified in the Appendix (Copestake, this volume). There are two essential differences which can be summed up as follows.

- Type hierarchies in LiLFeS must be a bounded complete partial order (Carpenter, 1992).
- Each type in LiLFeS is associated with its appropriate features, each of which is associated with an appropriate type.

The type system assumed in the LinGO grammar violates these conditions and therefore we cannot translate type definitions in the LinGO grammar straightforwardly. These problems are discussed next.

2.2 Translation of the LinGO grammar

An HPSG-based grammar is described by using *lexical entries* and *grammar rules*, both of which are denoted by *typed feature structures*. As a result, we have to translate type definitions, lexical entries, and grammar rules. Figure 2 is an example of translating type definitions and lexical entries from the $\mathcal{TDL}$ syntax into the LiLFeS syntax. Note that the LiLFeS language has clear distinction between the type definitions and the feature structure descriptions, while both are described with the same syntax in the $\mathcal{TDL}$.

Translation of lexical entries (and grammar rules) is simple because they are instances of feature structures and the typed feature structure description of the LiLFeS language is directly converted to the $\mathcal{TDL}$ as discussed above. In the lower section of Figure 2, a definition of a lexical entry is converted to the second argument of the predicate `lexical_entry`. This is simply because our parsers on LiLFeS presuppose that lexical entries are provided by the arguments of the predicate `lexical_entry`. Grammar rules are translated in the same way, and they are supposed to be provided by the predicate `id_schema`.

Problems come out when we turn to the translation of type definitions. As described in Section 2.1, the LiLFeS language follows *a totally well-typed feature structure* as defined in Carpenter (1992). Reinterpreting the differences between LiLFeS and the LinGO grammar, the type system in LiLFeS must satisfy the following conditions.

1. Only one least upper bound is allowed for each pair of consistent types
2. An appropriate value for a feature must be a type, not a feature structure

Note that *a greatest lower bound* in Copestake (this volume) corresponds to *a least upper bound* here and in Carpenter (1992). This article follows the definitions in Carpenter (1992).

Condition 1 comes from the condition that the proper type hierarchy is a finite bounded complete partial order, while the type hierarchy in the LinGO grammar does not satisfy this requirement as described in Copestake (this volume). To comply Condition 1, required least-upper-bound types are automatically computed by the LKB system and output for the LiLFeS version of the LinGO grammar (Copestake, this volume). The translated grammar includes 1118 new types for this condition.

Condition 2 is violated by some of the types in the LinGO grammar. For example, `head_only` in Figure 2 is associated with a feature structure. This feature structure is *a type constraint* (Copestake, this volume), which must follow the condition that if a feature structure F is associated with a type τ, a feature structure F' whose root is of the type τ must be unified with F. Apparently, the type constraints cannot be translated to feature appropriateness in the definition of totally well-typed feature structures.

This problem was solved by an extended feature of LiLFeS, which we call *complex constraints*. The extended type definition syntax for complex constraints is shown in Figure 3. The part following `./` defines complex constraints associated with the type *Newtype* as follows.

$Newtype$ `<- [` *supertypes...* `] + [` *features...* `]`
`./ constr\` F `&` `pred\` Q
F: A feature structure constraint
Q: A definite clause constraint

Fig. 3. A type definition syntax for defining complex constraints

Feature structure constraints F is unified with a feature structure with the associated type.
Definite clause constraints Q is executed when the associated type is generated.

The feature structure constraints correspond to the type constraints in Copestake (this volume).

The complex constraint solver is triggered when a type of a node in a feature structure is *changed* into the constrained type or its subtype. It is thus guaranteed that the same constraint will not be triggered again on the same node. The triggered constraint works as if a clause P(F):–Q. is called with F assigned to the changed node. If two or more constraints are triggered on a node at the same time, the order of execution is determined by the types associated to the constraints; a constraint of a certain type is executed earlier than constraints of its subtypes. As for constraints of two types without subsumption relation and constraints on different nodes, their execution order is arbitrary.

By using this extension, type constraints are successfully translated to the LiLFeS language. In Figure 2, a feature structure associated with `head_only` written in the $\mathcal{TDL}$ is converted to a feature structure in the LiLFeS language and specified following `constr\`. The translated LinGO grammar includes 721 types which are converted by using complex constraints.

The LinGO grammar was successfully ported to LiLFeS in a short period with the help of the LKB system and the extended feature of the LiLFeS language. The next section describes the mechanism for providing an efficient programming environment for HPSG-based systems including the translated LinGO grammar.

3 The LiLFeS abstract machine

The LiLFeS abstract machine is the core portion of the LiLFeS system, as shown in Figure 4. Since the LiLFeS language is a base system for developing natural language processing systems, the abstract machine should have i) efficient feature structure processing and ii) the ability to execute application programs. To provide these requirements, our research has focused on:

- Efficient implementation of feature structure operations frequently used in parsing tasks:
  - Unification of typed feature structures
  - Copying and equivalence checking
- Instructions and data structures required for application programs

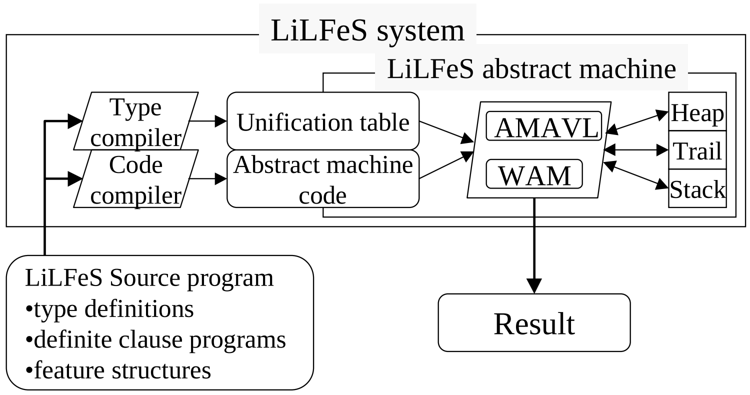

Fig. 4. The architecture of the LiLFeS system

— Execution of definite clause programs such as cut, findall, and negation as failure
— Standard data types such as strings, integers, floating point values, and arrays

For the efficient processing of typed feature structures, we used the *Abstract Machine for Attribute-Value Logics* (AMAVL), the architecture proposed by Carpenter & Qu (1995). By using AMAVL, the LiLFeS abstract machine benefits from i) compact memory representation of typed feature structures and ii) efficient unification by pre-compiling feature structures. In addition, the architecture has been extended for standard data types and arrays. To execute definite clause programs, we adopted *Warren's Abstract Machine* (WAM) (Aït-Kaci, 1991). WAM supports efficient backtracking, as well as cut, negation as failure, and predicate indexing. These two abstract machines are integrated into the LiLFeS abstract machine in an efficient way (Makino et al., 1997), as the data structures and abstract machine instructions of the two abstract machines are unified and re-designed.

The LiLFeS abstract machine has been extended further with efficient built-in subroutines such as efficient copying and equivalence checking of typed feature structures with the concept of *normalization*. Since these subroutines are frequently used in parsing, the increase in speed in these subroutines significantly improves parsing efficiency. In addition, a constraint solving mechanism associated with a type is implemented to handle the complex constraints introduced in Section 2.2.

Consequently, the LiLFeS abstract machine is an integration of the feature structure processing capabilities of AMAVL and the definite clause program execution function of WAM, with extensions required for natural language processing such as the efficient built-in subroutines and the complex constraints. Since the architecture of the abstract machines has already been published (Carpenter & Qu, 1995; Aït-Kaci, 1991; Makino et al., 1997), this section describes the extensions of the LiLFeS abstract machine.

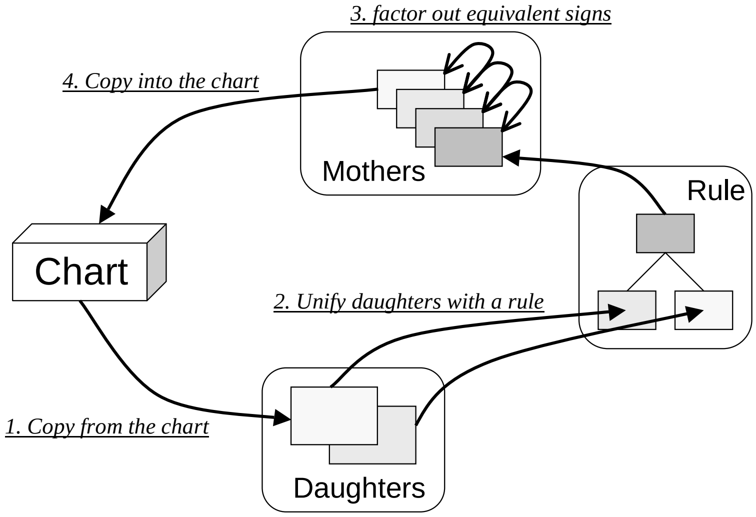

Fig. 5. Operations in a chart parser

3.1 Efficient built-in subroutines

Efficient copying and equivalence checking are especially useful in HPSG parsing because these operations consume a significant proportion of the total parsing time. Figure 5 illustrates operations in a chart parser. First an edge is retrieved from the chart and copied into a working area (1. copy from the chart). A rule is then applied to the copied feature structures (2. Unify daughters with a rule). If it succeeds, the result is compared to previously stored results for factoring out equivalent edges (3. factor out equivalence signs). Finally, the result is stored in the chart (4. copy into the chart). This chart parser is slightly different from those in other systems such as the LKB system. In our system, daughters are copied from the chart and the result is again copied into the chart. By copying the daughters to the working area in each rule application, the rule is applied with purely destructive unification without saving any trails for backtracking. This is faster than unification with backtracking.

However, this parsing algorithm requires extensive copying. The cost cannot be ignored if a naïve copying algorithm is used by traversing a feature structure. The equivalence checking for the edge factoring is also expensive because it requires simultaneous traversing on two typed feature structures and occurrence checking to avoid infinite loops on cyclic feature structures. The factoring is therefore not widely used, although it can reduce the number of rule applications and edges stored in the chart. These problems are solved when optimized built-in subroutines for efficient parsing are provided on the compact memory representation inherited from AMAVL.

To avoid inefficiency in copying, the LiLFeS abstract machine supports *block copying*, which does not need to traverse a feature structure (Makino et al., 1997; Brown & Manandhar, 1998, 2000). As shown in Figure 6, when the LiLFeS abstract machine copies a typed feature structure, it is first *normalized*. That is, the copied

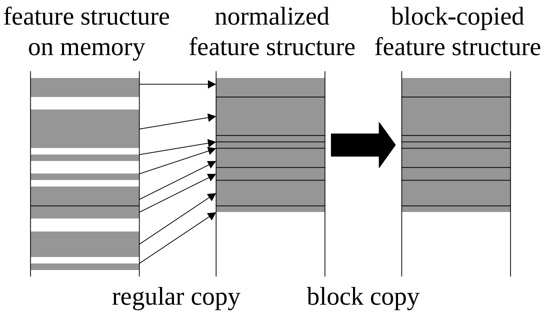

Fig. 6. Block copying

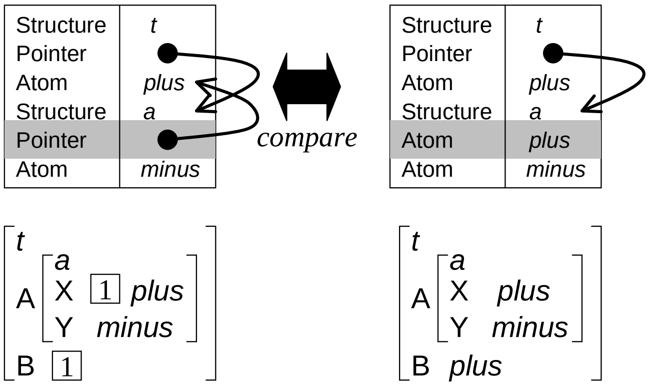

Fig. 7. Block equivalence checking

structure is put on a contiguous block in memory and sorted into a normal form. When copying this kind of normalized feature structure again, we can use block copying; copying the memory block in a linear way.

The factoring also benefits from normalization. *Block equivalence checking*, equivalence checking between two normalized feature structures, is more efficient than regular equivalence checking (see Figure 7). The block equivalence checking regards two normalized typed feature structures as two memory blocks to compare them in the same way as the block copying. Note that the two memory blocks are exactly the same only when they are equivalent feature structures. By using this routine within the factoring operation, recursively traversing a feature structure is not necessary, so factoring costs are drastically reduced.

It should be noted that the LiLFeS abstract machine requires application programmers to call a memory freeing predicate explicitly. This is because the automatic garbage collection involves processing overhead, and also because we can take full advantages of the built-in subroutines. It is hard to implement those subroutines with low overhead on systems that depend on other high-level programming language systems, such as Lisp and Prolog, since their own 'advanced' memory management interferes such subroutines. The parsers we have implemented on LiLFeS free memory whenever they finish parsing a sentence, so that the abstract machine is able to free all memory blocks that were used during the parsing.

3.2 Complex constraint solver

The complex constraint introduced in Section 2.2 is a constraint associated with a type τ and applied to every feature structure whose root node is of type τ or any of τ's subtypes. Any feature structures and definite clause programs can be used to express the constraint, which may include backtracking, cut, negation-as-failure, assertion, and any other operations.

The implementation of complex constraints is tricky; a naïve implementation, that calls up definite clause programs at the moment that a constraint type is generated, cannot be used since this may be unsound as it may call up programs on a partially unified structure. As a result, we chose the following method for implementation:

1. If a constrained type is generated during unification, a continuation frame, that points to the constraint solving routine, is pushed onto the stack.
2. When the unification is completed, each continuation frame is popped from the stack and its corresponding constraint is solved.
3. When all the constraints have been solved, the control is returned to the original routine.

The advantage of this method is that it does not require special handle routines to backtracking. Even when backtracking requires previous continuation frames to be restored on the stack, this is straightforward in this implementation because LiLFeS, as well as WAM, are able to preserve the content of popped portions of the stack for later recovery.

4 Performance evaluation

This section evaluates the efficiency of the LiLFeS abstract machine by performing parsing tasks with the LinGO grammar. The following six parsers were compared:

LKB (vanilla) The LKB system (Copestake, 1992, 1999) with all filtering methods disabled, simulating the LiLFeS naïve parsing strategy.
LKB (release) The parser in the October 1999 release version of the LKB system with filtering methods (Malouf, Carroll, and Copestake, this volume).
LKB (current) The parser in the current LKB development version with filtering and parsing strategy optimizations (Oepen & Carroll, this volume).
Naïve The CKY-style parser on LiLFeS with no filtering methods.
TNT The TNT parser on LiLFeS with CFG filtering (Torisawa et al., this volume).
TNT* The TNT parser with a threshold number of edges in the first (CFG) parsing phase (see Torisawa et al., this volume for details).

The following experiments are performed on three test suites described in Introduction in this volume, namely, '*csli*', '*aged*', and '*blend*'. Sentences that required

test set	parser	etasks	stasks	tgc (s)	total (s)	etasks/s
'csli'	**LKB (vanilla)**	36953	4308	0·18	3·98	9691
	LKB (release)	658	556	0·02	0·39	1760
	LKB (current)	359	183	0·00	0·23	1517
	Naïve	28002	3788	–	0·68	41094
	TNT	276	145	–	0·12	2300
'aged'	**LKB (vanilla)**	107219	13164	0·12	11·87	9060
	LKB (release)	1842	1604	0·07	1·13	1725
	LKB (current)	866	452	0·00	0·61	1409
	Naïve	68188	10002	–	1·72	39672
	TNT	668	379	–	0·31	2155
'blend'	**LKB (vanilla)**	703251	84755	2·97	80·99	9009
	LKB (release)	10493	9124	0·25	5·94	1839
	LKB (current)	3738	1685	0·12	3·10	1252
	Naïve	346772	70535	–	14·71	23574
	TNT	11261	4938	–	3·67	3068
	TNT*	9753	4308	–	1·90	5133

Table 1. *Parsing performance with the LinGO grammar*

more than 20,000 edges to parse were removed. The profiling data were collected by using [incr tsdb()] (Oepen & Flickinger, 1998; Oepen & Carroll, this volume) except for the TNT parser. The use of [incr tsdb()] also guarantees that the derivations obtained on LiLFeS are the same as the results of the LKB system. All the experiments were conducted on a 336 megahertz Sun UltraSparc with six gigabytes of memory.

We should note that *LKB (vanilla)* and *Naïve* are based on almost the same parsing algorithm, that is, they require almost the same number of rule applications (i.e. *etasks*). However, we must also mention that the number of tasks should be larger with *LKB (vanilla)* than with *Naïve*, because the grammars they use are slightly different. All lexical entries were precompiled (i.e. all lexical rules are already applied) and no lexical rules are included in the LiLFeS version of the LinGO grammar. The LiLFeS version has thirty seven grammar rules, while the original LinGO grammar has thirty seven grammar rules plus twenty seven lexical rules. *LKB* can filter out lexical rule applications to phrasal signs and does not suffer from the overhead of the application of the lexical rules.

4.1 Empirical results with the LinGO grammar

Table 1 shows the experimental results of parsing with the LinGO grammar and compares five parsers. The meanings of the values in each column are described in Oepen & Carroll (this volume). The last column shows the number of *etasks* per second, which provides an approximate measure of feature structure processing (mostly unification speed).

parser	preprocessing	filtering	parsing	total
Naïve	56	0	662	718
TNT	23	43	38	108

Table 2. *Time for preprocessing, filtering, and parsing (ms)*

Comparing the last column of *LKB (vanilla)* and *Naïve*, we can clearly see that the LiLFeS abstract machine processes feature structures very efficiently. The LiLFeS abstract machine can process (unify) typed feature structures around four times faster than the LKB system. While the number of *etasks* and *stasks* is significantly smaller in *LKB* due to the filtering methods, *Naïve* still offers a competitive parsing speed without using any filtering methods.

Comparing *LKB* and *TNT*, we notice that efficient feature structure processing is still essential with parsers based on advanced algorithms. When they performed almost the same number of *etasks* and *stasks*, *TNT* achieved faster parsing speed thanks to the efficiency of the LiLFeS abstract machine.

Naïve vs. *TNT* (and *LKB (vanilla)* vs. *LKB*) shows the effectiveness of the filtering method. The number of *etasks* was significantly reduced in *TNT* compared with the others, and the parsing speed was drastically increased. The number of unifications per second (*etasks/s*) is smaller than that in *Naïve*, because some percentage of the parsing time is consumed in filtering out unnecessary tasks and a successful unification tends to take more time than a failed unification. Note that the *stasks* value is also significantly reduced in *TNT*. This is because CFG filtering can approximate not only local constraints but also global constraints (Torisawa et al., this volume).

Table 2 shows the time consumed for preprocessing, filtering, and parsing in *Naïve* and *TNT* with the '*csli*' test suite. Since sentences in the test corpus are short, the preprocessing requires significant time in the overall parsing time. We found that most of the preprocessing time is consumed at fetching lexical entries. The cost for looking up lexical entries should be optimized in future research, although it might be less significant with longer sentences.

4.2 Efficiency of unification and built-in subroutines

Table 3 shows the time for the unification and other operations (copying and equivalence checking) in parsing the '*csli*' test suite. The total parsing time is slightly larger than the results in Table 1 because of the overhead for measuring the time. Unification accounts for most of the parsing time. When the efficient built-in subroutines are used (i.e. block copying and block equivalence checking), their computational cost is negligible as can be seen on the first row. The second row provides processing times without factoring, (i.e. without any equivalence checking). In this case, the efficiency of the block equivalence checking becomes evident as the processing time for the first two rows is virtually the same. This is reinforced by the

	parser	unification	copying & equiv. checking	total
Naïve	(with subroutines)	610	46	775
	(no factoring)	615	45	770
	(without subroutines)	625	196	877
TNT	(with subroutines)	30	17	123

Table 3. *Time for unification, copying, and equivalence checking (ms)*

poor performance without the efficient built-in subroutines (the third row). The fourth row shows the time consumption for the unification and built-in subroutines in the TNT parser. It shows that the efficiency of the built-in subroutines is more important in parsers based on an advanced parsing algorithm.

5 Ongoing work

We are researching several more methods to further increase the speed of feature structure processing. The following methods have already been evaluated and shown their effectiveness at an experimental level. They will be integrated within the current parsers and should contribute to their improvement.

The LiLFeS native code compiler (Makino, 1999) generates native machine code for a Pentium CPU from a LiLFeS source code. An abstract machine instruction is decomposed and directly compiled to several CPU-level instructions. The abstract machine code is optimized in compile-time by referring to the result of dataflow analysis with abstract interpretation. In the current state the compiled code is more than two times faster in parsing sentences than the abstract machine emulator. Unfortunately the parsing speed with the LinGO grammar cannot be measured because complex constraints have not been implemented yet.

Feature structure packing (Miyao, 1999) aims at processing feature structures with disjunctions efficiently. A set of non-disjunctive feature structures is automatically converted into a more compact data structure, *a packed feature structure*, by collapsing equivalent parts in the input feature structures. Since any unification operation is performed on the collapsed parts only once, unification speed is significantly improved for feature structures with many disjunctions. Even when the factoring operation is not effective, such as in parsing with the LinGO grammar, the packing method should improve the performance of the system significantly. Preliminary experiments show that the unification speed improved by a factor of 6.4 to 8.4.

6 Conclusion and future work

The LiLFeS system processes typed feature structures efficiently by taking an abstract machine approach. The efficiency of feature structure processing in the LiL-

FeS abstract machine is apparent in the experimental results done with the LinGO grammar ported to LiLFeS. Along with the efficient parsing systems, we are investigating several applications with HPSG-based grammars. Some research, such as statistical parsing (Kanayama et al., 1999) and robust parsing (Steiner & Tsujii, 1999a, 1999b) to support these applications is being conducted. Furthermore, the LiLFeS system is used not only for HPSG-based systems but also for other formalisms, such as an efficient TAG parser (Yoshida, Ninomiya, Torisawa, Makino, & Tsujii, 1999).

Acknowledgments

This article could not have been completed without the help of Dr. Ann Copestake. She helped with the translation of the LinGO grammar by using the LKB system. We are also indebted to Mr. Stephan Oepen for letting us use his profiling system [incr tsdb()] and his detailed discussion on the framework. We would like to thank Mr. Takashi Ninomiya for making a number of helpful suggestions and integrating [incr tsdb()] with the LiLFeS system. Finally, we thank the anonymous reviewers for their valuable comments on this manuscript.

References

Aït-Kaci, H. (1991). *Warren's Abstract Machine: A tutorial reconstruction.* Cambridge, MA: MIT Press.

Brown, J. C., & Manandhar, S. (1998). *An abstract machine for fast parsing of typed feature structure grammars* (Tech. Rep.). Computer Science Department of University of the Saarlandes.

Brown, J. C., & Manandhar, S. (2000). An abstract machine for fast parsing of typed feature structure grammars: including experimental results. In *Proceedings of Future Generation Computer Systems.*

Carpenter, B. (1992). *The logic of typed feature structures.* Cambridge, UK: Cambridge University Press.

Carpenter, B., & Qu, Y. (1995). An abstract machine for attribute-value logics. In *Proceedings of the 4th International Workshop on Parsing Technologies.* Prague, Czech Republik.

Copestake, A. (1992). The ACQUILEX LKB. Representation issues in semi-automatic acquisition of large lexicons. In *Proceedings of the 3rd ACL Conference on Applied Natural Language Processing* (pp. 88–96). Trento, Italy.

Copestake, A. (1999). *The (new) LKB system.* (CSLI, Stanford University: `http://www-csli.stanford.edu/~aac/doc5-2.pdf`)

Kanayama, H., Torisawa, K., Mitsuishi, Y., & Tsujii, J.-I. (1999). Statistical dependency analysis with an HPSG-based Japanese grammar. In *Proceedings of the Natural Language Processing Pacific Rim Symposium* (pp. 138–143). Beijing, China.

Kiefer, B., Krieger, H.-U., Carroll, J., & Malouf, R. (1999). A bag of useful techniques for efficient and robust parsing. In *Proceedings of the 37th Meeting of the Association for Computational Linguistics* (pp. 473–480). College Park, MD.

Makino, T. (1999). *A native-code compiler for a unification-based programming language with typed feature structures.* Unpublished master's thesis, University of Tokyo.

Makino, T., Torisawa, K., & Tsujii, J.-I. (1997). LiLFeS — practical programming language for typed feature structures. In *Proceedings of the Natural Language Processing Pacific Rim Symposium.*

Makino, T., Yoshida, M., Torisawa, K., & Tsujii, J. ichi. (1998). LiLFeS — towards a practical HPSG parser. In *Proceedings of the 17th International Conference on Computational Linguistics and the 36th Annual Meeting of the Association for Computational Linguistics* (pp. 807 – 11). Montreal, Canada.

Mitsuishi, Y., Torisawa, K., & Tsujii, J.-I. (1998). HPSG-style underspecified Japanese grammar with wide coverage. In *Proceedings of the 17th International Conference on Computational Linguistics and the 36th Annual Meeting of the Association for Computational Linguistics* (pp. 876 – 80). Montreal, Canada.

Miyao, Y. (1999). Packing of feature structures for efficient unification of disjunctive feature structures. In *Proceedings of the 37th Meeting of the Association for Computational Linguistics* (pp. 579 – 84). College Park, MD.

Oepen, S., & Flickinger, D. P. (1998). Towards systematic grammar profiling. Test suite technology ten years after. *Journal of Computer Speech and Language*, *12 (4) (Special Issue on Evaluation)*, 411 – 436.

Pollard, C., & Sag, I. A. (1994). *Head-Driven Phrase Structure Grammar.* Chicago, IL and Stanford, CA: The Univeristy of Chicago Press and CSLI Publications.

Steiner, R., & Tsujii, J.-I. (1999a). M-rules: Adaptable rules for robustness in unification-based systems. In *Proceedings of the Natural Language Processing Pacific Rim Symposium* (pp. 126 – 131). Beijing, China.

Steiner, R., & Tsujii, J.-I. (1999b). Robustness in HPSG via extended ID schemata. In *Proceedings of the Natural Language Processing Pacific Rim Symposium* (pp. 108 – 113). Beijing, China.

Tateisi, Y., Torisawa, K., Miyao, Y., & Tsujii, J.-I. (1998). Translating the XTAG English grammar to HPSG. In *Proceedings of the TAG+4 workshop* (pp. 172 – 175). Philadelphia, PA.

Uszkoreit, H., Backofen, R., Busemann, S., Diagne, A. K., Hinkelman, E. A., Kasper, W., Kiefer, B., Krieger, H.-U., Netter, K., Neumann, G., Oepen, S., & Spackman, S. P. (1994). DISCO — an HPSG-based NLP system and its application for appointment scheduling. In *Proceedings of the 15th International Conference on Computational Linguistics.* Kyoto, Japan.

Yoshida, M., Ninomiya, T., Torisawa, K., Makino, T., & Tsujii, J.-I. (1999). Efficient FB-LTAG parser and its parallelization. In *Proceedings of the conference of the Pacific Association for Computaional Linguistics.* Waterloo, Canada.