LiLFeS – Towards a Practical HPSG Parser *

MAKINO Takaki†, YOSHIDA Minoru†, TORISAWA Kentaro†, TSUJII Jun'ichi†‡
†Tsujii Group, Department of Information Science, University of Tokyo
Hongo 7-3-1, Bunkyo-Ku, Tokyo 113-0033, Japan
{mak,mino,torisawa,tsujii}@is.s.u-tokyo.ac.jp
‡CCL, UMIST, U.K.

Abstract

This paper presents the LiLFeS system, an efficient feature-structure description language for HPSG. The core engine of LiLFeS is an Abstract Machine for Attribute-Value Logics, proposed by Carpenter and Qu. Basic design policies, the current status, and performance evaluation of the LiLFeS system are described. The paper discusses two implementations of the LiLFeS. The first one is based on an emulator of the abstract machine, while the second one uses a native-code compiler and therefore is much more efficient than the first one.

1 Motivation

Inefficiency is the major reason why the HPSG formalism (Pollard and Sag, 1993) has not been used for practical applications. However, one can claim that HPSG may not be so inefficient; it is just that an efficient implementation of HPSG has not been seriously pursued till now.

We set a goal for the performance of our HPSG parser: 100 milliseconds of average parsing time on a sentence in real-world corpora. If our HPSG parser accomplished this goal, it would be capable to parse about 1,000,000 sentences in a day, and could be used for applications such as knowledge acquisition from corpora.

1.1 Existing Systems for Typed Feature Structures (TFSs)

Since Typed Feature Structures (TFSs) (Carpenter, 1992) are the basic data structures in HPSG, the efficiency of handling TFSs has been considered as the key to improve the efficiency of an HPSG parser. There are two representative systems that handle TFSs[1]: ALE (Carpenter and Penn, 1994), a TFS interpreter written in Prolog, and ProFIT (Erbach, 1995), a TFS-to-Prolog-term compiler.

However, as the comparison of these systems with our system (Section 3.2) shows, neither of these two systems is able to achieve the efficiency we established as our goal. Moreover, these two systems have serious disadvantages as a framework for practical applications. The ProFIT approach, for example, tends to consume too much memory for execution. It is also difficult, if not impossible, to combine them with other techniques like parallel parsing, etc., because these two systems have been embedded in Prolog.

1.2 Our Approach

One of the promising directions of improving the efficiency of handling TFSs while retaining a necessary amount of flexibility is to take up the idea of AMAVL proposed in (Carpenter and Qu, 1995) to design a general programming system based on TFS.

LiLFeS is a logic programming system thus designed and developed by our group, based on AMAVL implementation. LiLFeS can be characterized as follows.

- Architecture based on an AMAVL implementation, which compiles a TFS into a sequence of abstract machine instructions, and performs unification of the TFS by emulating the execution of those instructions. Although the proposal of such an AMAVL was already made in 1995, no serious implementation has been reported. We believe that LiLFeS is the first serious treatment of the proposal.
- Rich language specification: We have adopted a language syntax similar to Prolog. LiLFeS as a programming language has almost the full capabilities of ordinary Prolog systems. Furthermore, we provide efficient built-in predicates that are often required in NLP applications, such as TFS copy, equivalence check, and associative arrays.
- Independent language system: In order to develop an efficient and portable language system, we chose not to develop the language depending on an existing high-level language such as Prolog. Instead, we programmed the LiLFeS system from scratch. The independence also allows us to provide various built-in predicates in efficient ways.

1.3 Structure of This Paper

Section 2 describes LiLFeS as a programming

* This research is partially funded by the project of Japan Society for the Promotion of Science (JSPS-RFTF96P00502).

[1] LIFE (Aït-kaci et al., 1994) is also famous, but we do not discuss it because it does not follow Carpenter's TFS definition. Moreover, our separate experiments show that LIFE is more than 10 times slower than emulator-based LiLFeS. As for AMALIA (Wintner, 1997), we cannot make experiments since it is not freely distributed. His experiments in his dissertation shows that AMALIA is 15 time faster than ALE at maximum; it is close to emulator-based LiLFeS, and is outperformed by native-code compiler of LiLFeS.

```
my_list <- [bot].
e_list <- [my_list].
ne_list <- [my_list]
         + [FIRST\ bot, REST\ my_list].

append(e_list, X, X).
append( (FIRST\ A & REST\ X),
        Y,
        (FIRST\ A & REST\ Z) ) :-
                  append( X, Y, Z ).
```

Figure 2 Sample LiLFeS Program

language. Section 3 gives a brief description of the AMAVL we implemented, the core inference engine of the LiLFeS system. In Section 4, we discuss the current status of the LiLFeS system and the results of experiments on the system performance. Section 5 describes a native-code compiler we are currently developing on the LiLFeS system, and discusses its performance.

2 LiLFeS as a Programming Language

LiLFeS has basically the same syntax as Prolog, except that it uses TFSs instead of terms. Types and features must be defined before being used in TFS terms.

Figure 2 show the definition of the predicate `append` in LiLFeS. The first paragraph contains the type definitions of `my_list`, `e_list`, and `ne_list`. The type `ne_list`, for example, is a subtype of the type `my_list`, and has two appropriate features, `FIRST` and `REST`. The value of the feature `REST` is restricted to the type `my_list` or one of its subtypes. The type `bot` is the universal type that subsumes all types.

The rest of the program is definite clauses. As one can see, the predicate `append` is represented by TFSs instead of Prolog first-order terms[2].

3 Abstract Machine for Attribute-Value Logics

The Abstract Machine for Attribute-Value Logics (AMAVL) is the unification engine of the LiLFeS system. AMAVL provides (1) efficient representation of TFSs on the memory, and (2) compilation of TFSs into abstract machine codes.

3.1 Representation of a TFS on the Memory

AMAVL, as does LiLFeS, requires all TFSs to be totally well-typed[3]. In other words, (1) the types and features should be explicitly declared, (2) appropriateness of specifications between types and features should be properly declared, and (3) all TFSs should follow these appropriateness specifications. Provided these requirements are satisfied, AMAVL efficiently represents a TFS in memory.

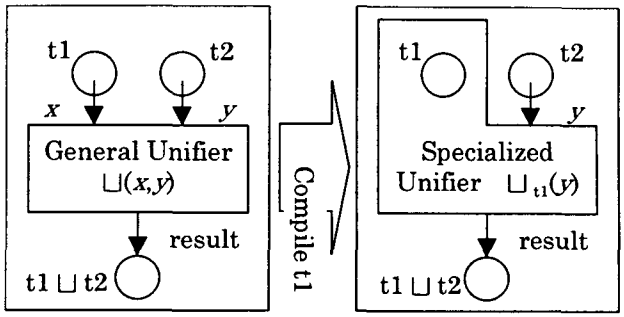

Figure 1 Compiling TFS

The representation of a TFS on memory resembles the graph notation of a TFS; A node is represented by $n+1$ continuous data cells, where n is the number of features outgoing from the node. The first data cell contains the type of the node, and the rest of the data cells contain pointers to the values of the corresponding features.

The merit of this representation is that feature names need not be represented in the TFS representation on memory. The requirements on a TFS guarantee that the kinds and number of features are statically defined and constant for a given type, therefore we can determine the offset of the pointer to a given feature only by referring to the type of the given node.

3.2 TFS as an Instruction Sequence

Unification is an operation defined between two TFSs. However, in most cases, one of the two TFSs is known in advance at compile-time. We can therefore compile the TFS into a sequence of specialized codes for unification, as illustrated in Figure 1, rather than using a general unification routine. The compiled unification codes are specialized for given specific TFSs and therefore much more efficient than a general unifier. This is because any general unifier has to traverse both TFSs each time unification occurs at run-time.

Many studies have been reported for compiling unification of Prolog terms (for example, WAM (Aït-Kaci, 1991)). However, the TFS unification is much more complex than Prolog-term unification, because (1) unification between different types may succeed due to the existence of a type hierarchy, and (2) features must be merged in the fixed-offset TFS representation on memory.

AMAVL compiles a type hierarchy and prepares for the complex situations described above. A TFS itself is compiled into a sequence of four kinds of instructions: `ADDNEW` (the unification of TFS types), `UNIFYVAR` (creation of structure-sharing), `PUSH` (feature traversing) and `POP` (end of `PUSH` block). These instructions refer to the compiled type hierarchy, if necessary.

These operations are implemented following the

[2] Note that LiLFeS has built-in list types as Prolog does. This program is just an example to illustrate how a TFS is represented in LiLFeS

[3] For a more formal definition, see (Carpenter, 1992).

Components	Lines
WAM/AMAVL Emulator	5,434
LiLFeS-to-WAM/AMAVL Compiler	6,091
Built-in Functions	9,530
TFS Display Routine	2,320
Others (Class Library etc.)	2,374
Total	25,749

Table 1 Source Code Lines of the LiLFeS System

original proposal of AMAVL. We also added several other instructions in our AMAVL implementation, such as initialization of instructions for successive unifications and combined instructions for reducing overhead.

4 LiLFeS System

The LiLFeS system is designed based on two abstract machines: AMAVL for TFS representation and unification procedures, and Warren's Abstract Machine (WAM) (Aït-Kaci, 1991) for control of execution of definite clause programs. In this section we describe the current status of the LiLFeS system and applications running on it. Thereafter, we discuss the performance of LiLFeS in our experiments.

4.1 Current Status of the LiLFeS System

The LiLFeS system is developed as a combination of AMAVL/WAM emulator, TFS compiler, and built-in support functions. They are all written in C++ with the source code of more than 25,000 lines (See Table 1). The source code can be compiled by GNU C++, and we have confirmed operation on Sun SunOS4/Solaris, DEC Digital UNIX, and Microsoft Windows.

We have several practical applications on the LiLFeS system. We currently have several different parsers for HPSG and HPSG grammars of Japanese and English, as follows:

- A underspecified Japanese grammar developed by our group (Mitsuisi, 1998). Lexicon consists of TFSs each of which has more than 100 nodes. The grammar can produce parse trees for 88% of the corpus of the real world texts (EDR Japanese corpus), 60% of which are given correct parse trees[4]. This grammar is used for the experiments in the next section.
- XHPSG, An HPSG English grammar (Tateisi, 1997). The grammar is converted from the XTAG grammar (XTAG group, 1995), which has more than 300,000 lexical entries.
- A naïve parser using a CYK-like algorithm. Although using a simple algorithm, the parser utilizes the full capabilities provided by LiLFeS, such as built-in predicates (TFS copy, array operation, etc.).
- A parser based on the Torisawa's parsing algorithm (Torisawa and Tsujii, 1996). This algorithm compiles an HPSG grammar into 2 parts: its CFG skeletons and a remaining part, and parses a sentence in two phases. Although the parser is not a complete implementation of the algorithm, its efficiency benefits from its 2-phase parsing, which reduces the amount of unification.

These parsers and grammars are used for the performance evaluations in the next section.

[4] The grammar does not contain semantic analysis such as coreference resolution.

(Parsing time per sentence, Unit: seconds)

Our Goal	0.100
CYK-style naïve parser	1.050
Parser based on Torisawa's algorithm	0.350

Condition: 600 sentences from EDR Japanese corpus (average length 21 words), Average in the parsing of successfully parsed 539 sentences
Environment: DEC Alpha 500/400MHz with 256MB memory

Table 2 Parsing Performance Evaluation with a Practical Grammar

4.2 Performance Evaluation

We evaluated the performance of the LiLFeS system over three aspects: Parsing performance of LiLFeS, comparison to other TFS systems, and comparison to different Prolog systems.

Table 2 shows the performance of HPSG parsers on a real-world corpus. However, even with the sophisticated algorithm, the parsing speed is 3.5 times slower than intended. To achieve our goal, we need a drastic improvement of a performance. We therefore performed the following experiments to find out the problem.

Table 3 shows the performance comparison to other TFS systems, ALE and ProFIT. Two grammars are used in the experiments: "Simple" is a small HPSG-like grammar written by our group, while "HPSG" is the small-lexicon HPSG grammar distributed with the ALE package. In the "Simple" experiments, the LiLFeS system is far more efficient than ALE, but is outperformed by ProFIT. However, in the "HPSG" experiment, which has to handle much more complex TFSs than "Simple" experiments, LiLFeS is clearly better than ProFIT.

On the contrary, with simple data LiLFeS is relatively inefficient. Experiments in Table 4, which show comparisons to Prolog systems, show that the performance of LiLFeS is significantly worse than that of those Prolog systems.

To summarize, the performance of LiLFeS is far more impressive when it has to handle complex TFSs. This fact indicates that the TFS engine in LiLFeS is efficient but that the other parts, i.e. the

System \ Grammar	Simple (1 answer)	Simple (64 answers)	HPSG
LiLFeS system (emulator-based)	5.70	322.4	2.56†
ALE on SICStus WAM emulation	225.60	10560	37.71*
ALE on SICStus native-code	67.05	3046	26.69*
ProFIT on SICStus WAM emulation	2.94	127.51	8.08
ProFIT on SICStus native-code	1.48	64.08	9.78

(Unit: seconds)

Simple: a simple HPSG-like grammar, parsed 1000 times by a bottom-up parser, 9-word sentence results in 1 parse-tree
HPSG: a toy HPSG grammar distributed with ALE, parsed by a parser distributed with ProFIT, 14-word sentence results in 134 parse-trees
*: ALE built-in parser is used instead of parser written in definite clauses
†: The parser program is translated to avoid the "call" built-in, which contains some problems in the LiLFeS implementation
(Environment: Sun UltraSparc 1/167MHz with 128MB memory)

Table 3 Performance Comparison to Other TFS Systems

parts concerning LiLFeS as a general logic programming system, are not yet efficient enough. This means that, in order to improve the LiLFeS system as a whole, we have to include various optimization techniques already encoded in recent Prolog implementations.

Thus we decided to redesign and optimize the whole system. The next section describes this optimized LiLFeS.

5 LiLFeS Native-Code Compiler

We are currently developing a native-code compiler of LiLFeS in order to attain maximum performance. This section at first describes the design policies of the compiler, and then, describes the current status of implementation. The results of the performance evaluations on the native-code compiler are also presented.

5.1 Design Policies of the LiLFeS Native-Code Compiler

The design policies for the LiLFeS native-code compiler are:

- Native code output. We chose native-code compiling for optimal efficiency. Although this costs high for development, the resulting efficiency will compensate the cost.
- Execution model close to a real machine. We designed the execution model by referring to the implementation of Aquarius Prolog (Van Roy, 1990), an optimizing native-code compiler for Prolog. Aquarius Prolog adopts an execution model with an instruction set that is fine-grained and close to an instruction set of a real machine. As a result, the output code can be optimized up to the real-machine instruction level. In particular, we fully redesigned the AMAVL instructions as fine-grained instructions, which allow extensive optimizations on compiled TFS code.
- Static code analysis. The types of variables can be determined by analyzing the flow of data within a program. The result of this dataflow analysis will help to further optimize in the compilation process.

These techniques are the basis of latest Prolog systems. It is therefore expected that LiLFeS augmented with these techniques becomes as efficient as commercially available Prolog systems.

(Unit: seconds)

System \ Application	fib(30)	rev(1000) (10 times)
LiLFeS (emulator-based)	106.4	48.7
SICStus WAM emulation	5.21	4.84
SICStus native-code	1.12	2.02
Aquarius Prolog	1.27	0.953

fib(30): Naïve calculation of Fibonacci(30) = 1346269
rev(1000): Naïve reverse of 1000-element list
(Environment: Sun UltraSparc 1/167MHz with 128MB memory)

Table 4 Performance Comparison to Prolog Systems

5.2 Current Status of the LiLFeS Native-code Compiler

We are developing the LiLFeS native-code compiler in LiLFeS itself. This is because the best language that manipulates TFSs is LiLFeS; low-level languages, such as C, are not appropriate for TFS manipulation.

Currently all of the basic components have been implemented. We are now working on further code optimizations and implementation of built-in functions on the native-code compiler.

5.3 Performance Evaluation of the LiLFeS Native-Code Compiler

We evaluated the performance of the LiLFeS native-code compiler with the same experiments as used in Section 4.2. The results of the experiments are shown in Table 5 and Table 6.

The results of the native-code compiler are significantly better than those of the emulator-based LiLFeS system. In particular, comparison to Prolog (Table 6) shows that the LiLFeS native-code compiler achieves a speedup of 20 to 30 times compared to emulator-based LiLFeS, and

System \ Grammar	Simple (1 answer)	Simple (64 answers)	HPSG
LiLFeS native-code compiler	1.46	77.51	0.92†
LiLFeS system (WAM based)	5.70	322.4	2.56†
ALE on SICStus WAM emulation	225.60	10560	37.71*
ALE on SICStus native-code	67.05	3046	26.69*
ProFIT on SICStus WAM emulation	2.94	127.51	8.08
ProFIT on SICStus native-code	1.48	64.08	9.78

(Unit: seconds)

(See notes in Table 3 for environment and other notes)

Table 5 Performance Comparison of LiLFeS Native-Code Compiler to Other TFS Systems
(Some of the data is overlapped to Table 3)

approaches to the native-code compiler versions of commercial Prolog systems. We can say that the bottleneck of the emulator-based LiLFeS system is effectively eliminated.

The result of the comparison to other TFS systems (Table 5) shows a speedup of 3–5 times from the emulator-based LiLFeS. It is still slower than ProFIT + SICStus native-code compiler in some experiments, though the difference is very small. We think the reason is the different traversing order between ProFIT + SICStus (breadth-first) and LiLFeS native-code compiler (depth-first)[5].

What is notable in those experiments is that the LiLFeS native code compiler shows a far better performance in the "HPSG" experiment than all other systems. Since the "HPSG" experiment focuses on the efficiency of TFS handling, this means that the native code compiler improves the TFS handling capability.

We cannot yet perform the experiments on real-world text parsing, because the implementation of the native-code compiler is not completed. However, we can estimate the result from the experiment result on emulator-based LiLFeS (350 milliseconds with sophisticated algorithm) and speed ratio between emulator-based LiLFeS and native-code compiler (3 to 5 times speed-up). The estimated parsing time is 120ms – 70ms per sentence; so we can say that we will be able to achieve our goal of 100ms in the near future.

6 Conclusion

We developed LiLFeS, a logic programming language for TFSs. Using AMAVL emulator as a core of the inference engine, the LiLFeS system achieves high efficiency on complex TFSs. We are now developing a native-code compiler version of LiLFeS; the prototype showed a significant speedup from the emulator-based version.

[5] We confirmed in the separate experiments that execution time of the "Simple" test varies up to 15% by changing the traversing order.

System \ Application	fib(30)	rev(1000) (10 times)
LiLFeS native-code compiler	2.45	2.29
LiLFeS (emulator-based)	106.4	48.7
SICStus WAM emulation	5.21	4.84
SICStus native-code	1.12	2.02
Aquarius Prolog	1.27	0.953

(Unit: seconds)

(See notes in Table 4 for environment and other notes)

Table 6 Performance Comparison of LiLFeS Native-Code Compiler to Prolog Systems
(Some of the data is overlapped to Table 4)

References

Makino, Takaki et al. (1998) LiLFeS Home Page. Available at http://www.is.s.u-tokyo.ac.jp/~mak/lilfes/.

Carl Pollard and Ivan A. Sag (1994) Head-Driven Phrase Structure Grammar. University of Chicago Press and CSLI Publications.

Bob Carpenter (1992) The Logic of Typed Feature Structures. Cambridge University Press.

Bob Carpenter and Yan Qu (1995) An abstract machine for attribute-value logics. In IWPT '95, pp. 59–70.

Gregor Erbach (1995) ProFIT—Prolog with Features, Inheritance and Templates. In EACL '95.

Bob Carpenter and Gerald Penn (1994) ALE 2.0 User's Guide. Carnegie Mellon University Laboratory for Computational Linguistics Technical Report.

Hassan Aït-Kaci (1991) Warren's Abstract Machine, A Tutorial Reconstruction. The MIT Press.

Peter Lodewijk Van Roy (1990) Can logic programming execute as fast as imperative programming? Technical Report CSD-90-600, University of California, Berkeley.

Kentaro Torisawa and Jun'ichi Tsujii (1996) Computing phrasal-signs in HPSG prior to parsing. In COLING '96, pages 949–955.

Tateisi, Yuka et al. (1997) Conversion of LTAG English Grammar to HPSG. IPSJ Report NL-122. (In Japanese).

The XTAG Research Group (1995) A Lexicalized Tree Adjoining Grammar for English. IRCS Research Report 95-03, IRCS, University of Pennsylvania.

Mitsuishi Yutaka et al. (1998) HPSG-Style Underspecified Japanese Grammar with Wide Coverage. To appear in COLING-ACL '98.

Hassan Aït-Kaci et al. (1994) Wild LIFE Handbook (prepublication edition)., a User Manual. PRL Technical Note #2.

Shalom Wintner (1997) An Abstract Machine for Unification Grammars — with Application to an HPSG Grammar for Hebrew. Ph.D. thesis, the Technion – Israel Institute of Technology.