LiLFeS —
Practical Unification-based Programming System for Typed Feature Structures

MAKINO Takaki,
TORISAWA Kentaro,
TSUJII Jun'ichi

Department of Information Science,
Tokyo University

1

Good morning.

My name is Makino Takaki, Tokyo University.

I will give a presentation about "LiLFeS, Practical Unification-based Programming System for Typed Feature Structure."

Outline

- Goal
- Background
- Design Policy
- Overview of LiLFeS Language
- Architecture of the LiLFeS Interpreter
- Current Status
- Performance Evaluation
- Future of LiLFeS System

2

This is the outline of my presentation.

At first, I will describe our goal, the background, and our design policy.

Our Goal

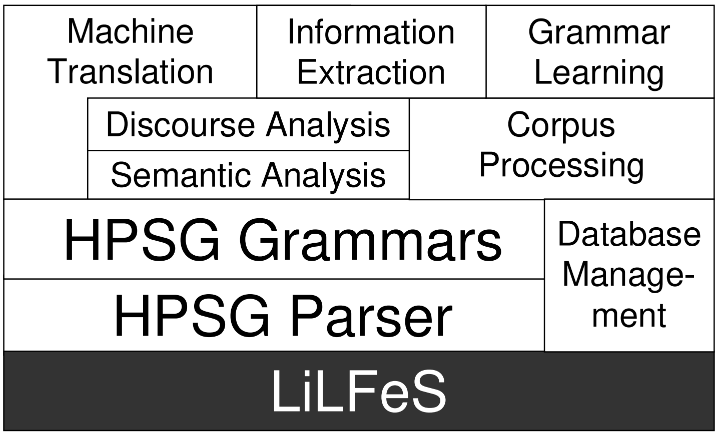
- Programming language for developing flexible and efficient NLP systems, especially on HPSG (Pollard & Sag, 94)
  - HPSG uses Typed Feature Structure (TFS)
- For this purpose, we develop LiLFeS

3

The goal of this study is to provide a programming language for developing flexible and efficient NLP systems. As the primary formalism for our NLP applications, we chose HPSG, a sophisticated grammar formalism proposed by Pollard and Sag. Since HPSG uses typed feature structure, the language must be capable of handling typed feature structure. For this purpose, we developed the language LiLFeS.

Goal: Platform for NLP systems

- Language for TFS is required for high-level NLP systems



4

As I described in the previous slide, what we want to do is these NLP works, such as machine translation and so on. In order to realize them, we need, in addition to these modules, a practical syntactic analyzer. Since we chose HPSG as the primary formalism, we need a platform programming language for typed feature structures. This is LiLFeS, what I describe in this presentation.

Background(1/2)

- Existing systems for TFS
  - ALE (Carpenter et al, 94) : Parser that interprets feature structures, written in Prolog
  - ProFIT (Erbach, 94): Compiler from TFS to Prolog terms

5

Currently there are some systems for typed feature structures already developed. One is ALE, developed by Carpenter et al. This is a parser that interprets feature structures on Prolog.

Another system is ProFIT, developed by DFKI in Germany. This is a compiler from Typed feature structures to Prolog terms. The result program is a Prolog program and can be executed efficiently, because recent Prolog systems are efficient.

Background(2/2)

- Problems:
  - ALE is not efficient
  - ProFIT represents TFS indirectly
    → Tuning for practical NLP systems is hard
    (hard to make built-ins for TFS)

6

However, these systems have problems.

As for ALE, the problem is inefficiency. ALE is interpreting TFS on Prolog, and cannot provide efficiency enough to meet our goal, such as grammar learning from a large corpus.

On the other hand, ProFIT is efficient. However, since ProFIT represents TFS indirectly, it is hard to tune for practical NLP systems. For example, we want to tune up a routine that is used heavily in NLP applications. One solution is to rewrite the routine in low-level language, such as C, and build into the system. However, the data of ProFIT is a compiled term from TFS to Prolog term, so it is complex and hard to understand, let alone manipulating from C routine.

Design Policy

- Efficient system that can handle TFSs
  - Efficient TFS unification
  - Direct representation of TFS
  - Powerful built-in predicates that provide flexibility and efficiency for NLP
- Practical programming capability
  - Full Prolog features
  - Integration with existing NLP systems

7

To solve these problems, we designed LiLFeS according to these policies.

We require LiLFeS to be efficient system that can handle typed feature structure.

Concretely, LiLFeS should be able to perform typed feature structure unification efficiently.

We also need to solve the problem of ProFIT described before, that is, hard to tune up with built-ins, and we require that the typed feature structure is represented in a direct way.

It should be also easy to integrate with efficient programs written in low-level language such as C++.

On the other hand, we require practical programming capability for LiLFeS.

Since pure-prolog level programming language is not sufficient for practical applications, we need full prolog features, such as cut, negation as failure, and assertions.

The system should be also able to integrate with existing NLP system, for example, stand-alone morphological analyzer.

These requirements just turn out to the design policy for LiLFeS.

Outline

- Goal
- Background
- Design Policy
- Overview of LiLFeS Language
- Architecture of the LiLFeS Interpreter
- Current Status
- Performance Evaluation
- Future of LiLFeS System

9

Next, I will describe overview of LiLFeS language, the architecture of the LiLFeS Interpreter, and the current status of LiLFeS system.

Overview of LiLFeS Language

- Syntax is similar to Prolog
  with additional features of handling
  Typed Feature Structures (TFSs)

```
alist <- [bot].
elist <- [alist].
nelist <- [alist] + [FIRST:bot, REST:alist].
append <- [pred].

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

Sample LiLFeS program

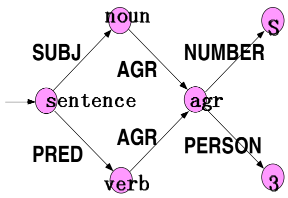
10

Let me talk about the overview of LiLFeS language.

The syntax of LiLFeS is similar to Prolog. The difference is that LiLFeS uses typed feature structures for terms in addition to Prolog first-order terms. This is the sample LiLFeS program. These are type definitions, and these are clauses. As you can see, LiLFeS can handle definite clauses. The terms in definite clauses are typed feature structures.

Typed Feature Structure (1/3)

- Formally defined in (Carpenter, 92)
- A rooted directed graph
  - Nodes/edges are labeled by types/features
- LiLFeS code resembles AVM notation



$$\begin{bmatrix} \mathtt{sentence} & \\ \text{SUBJ} & \begin{bmatrix} \mathtt{noun} & \\ \text{AGR} & \boxed{1}\begin{bmatrix} \mathtt{agr} & \\ \text{NUMBER} & \text{S} \\ \text{PERSON} & 3 \end{bmatrix} \end{bmatrix} \\ \text{PRED} & \begin{bmatrix} \mathtt{verb} & \\ \text{AGR} & \boxed{1} \end{bmatrix} \end{bmatrix}$$

```
(sentence &
 SUBJ:(noun &
       AGR:(X & agr &
            NUMBER:S &
            PERSON:3))&
 PRED:(verb &
       AGR:X)).
```

Graph Notation AVM Notation LiLFeS Notation

Let us take a closer look on typed feature structures.

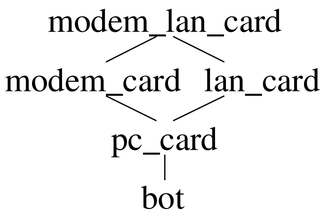
Bob Carpenter defined typed feature structure formally. A typed feature structure is a rooted directed graph, where nodes and edges are labeled. Node labels are called "types", and edge labels are called "features." The graph may have merging points, like this. It is called "structure-sharings."

We usually use attribute-value-matrix notation to represent a typed feature structure. This is the type, and this is the feature, and this is the substructure rooted from this node. Structure-sharing is represented by squared numbers, called tags.

LiLFeS program code resembles this AVM notation. Structure-sharing is handled by variables, like this.

Typed Feature Structure (2/3)

- Inheritances forms a type hierarchy
- Unification result is the least upper bound of two types



Sample type hierarchy

```
pc_card    <-[bot] + [maker:].
modem_card <-[pc_card] + [bps:].
lan_card   <-[pc_card] + [cable:].
modem_lan_card <-
              [modem_card,lan_card].
```

LiLFeS type definition

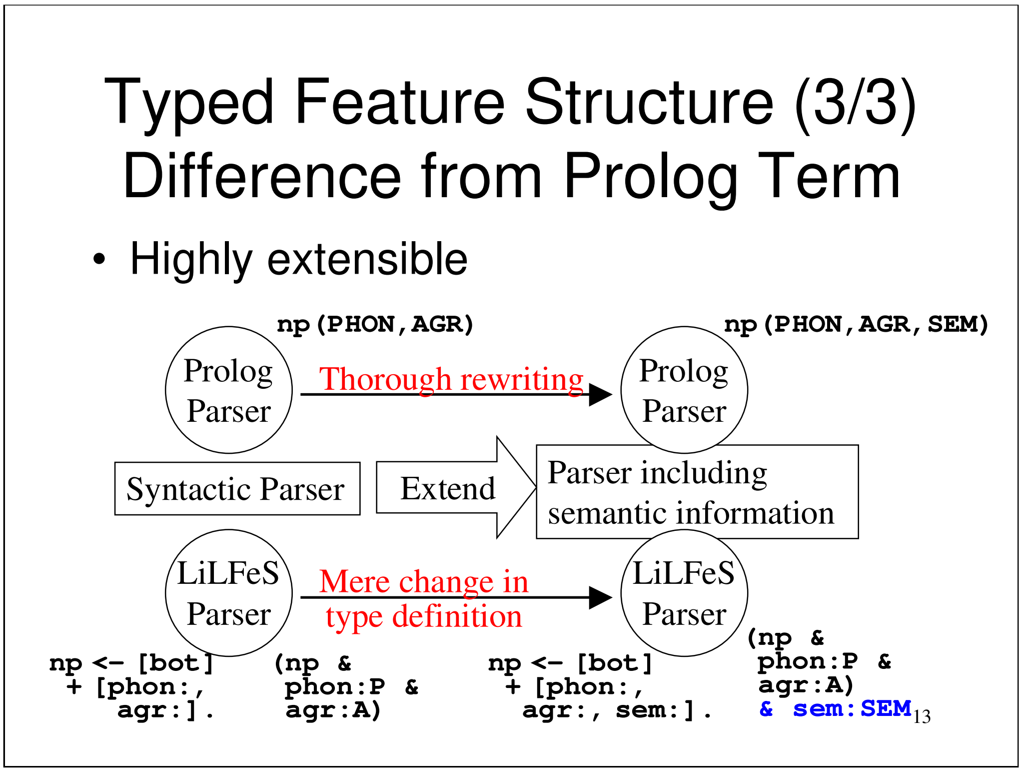
12

Types must be defined in order to be used in typed feature structures. We define inheritance relations between types. By the inheritance, the types forms a hierarchy.

This is the sample type hierarchy. type pc-card is a subtype of the type bot, the universal bottom type, and has a feature ‘maker.’ Type modem-card is a subtype of pc-card, so modem-card has the inherited feature ‘maker’ in addition to the introduced feature ‘bps.’

When two types are unified, the result is the least upper bound of two types. For example, unified result of type modem-card and type lan-card is type modem-lan-card.

These are type definitions in LiLFeS. As you can see, features are also defined in the type definitions.



The advantage of typed feature structure over Prolog terms is its high extensibility.

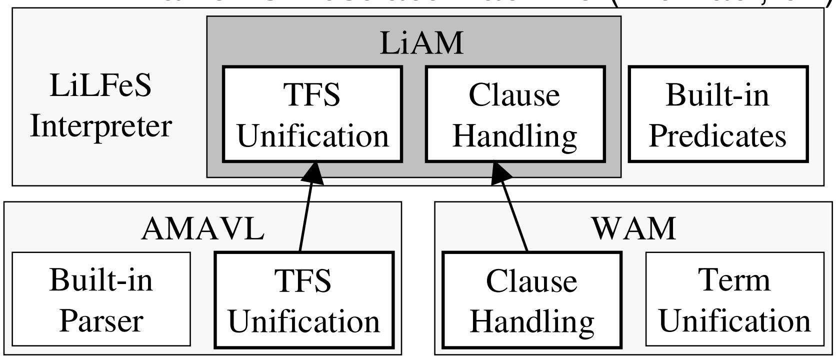
Suppose that we just wrote syntactic parser on Prolog. In the program, a term that represents noun phrase may be in this form. If we want to add semantic information to the parser, we will add the semantic information into the term noun phrase by increasing its arity. In this case, we must rewrite the parser thoroughly, because parser must be adapted to the new form of the term.

On the other hand, in the parser written in LiLFeS, we give a type definition for the term of NP as this sample, and represent NP term in this form. In order to add semantic information to the parser, all we have to do is a mere change in type definition; if feature SEM is defined in the definition of type NP, the semantic information is held in every NP term and handled properly.

This is only an example but clearly shows the capacity of Typed Feature Structures.

Architecture of the LiLFeS Interpreter

- LiAM — Inference engine
  - Abstract Machine for Attribute-Value Logics (Carpenter et al., 95)
  - Warren's Abstract Machine (Aït-Kaci, 91)



14

The next is the architecture of the LiLFeS interpreter.

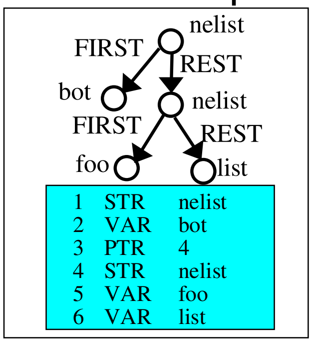
We developed an interpreter for the language LiLFeS.

The core of the interpreter is a LiLFeS abstract machine, LiAM. LiAM executes LiLFeS efficiently by interpreting abstract machine instructions, which is like intermediate code.

Actually, LiAM is the combination of two abstract machines. One is an Abstract Machine for Attribute-Value Logics, proposed by Carpenter et al. This controls unification of typed feature structures. The other part is Warren's Abstract Machine, one of the best-known framework of efficient Prolog implementation. This controls definite clauses and other full-Prolog features. I will describe this unification part first, and after that, this clause part.

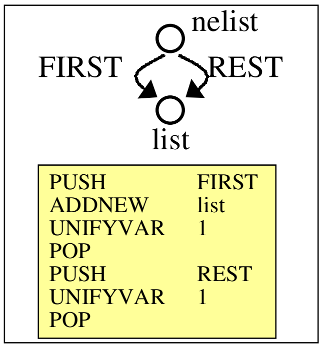
LiAM: Unification Part (AMAVL)

- Two types of representations for a TFS
  - Heap representation / Code representation
- Performs unification by the execution of code-represented TFS



← Heap representation

Code representation →

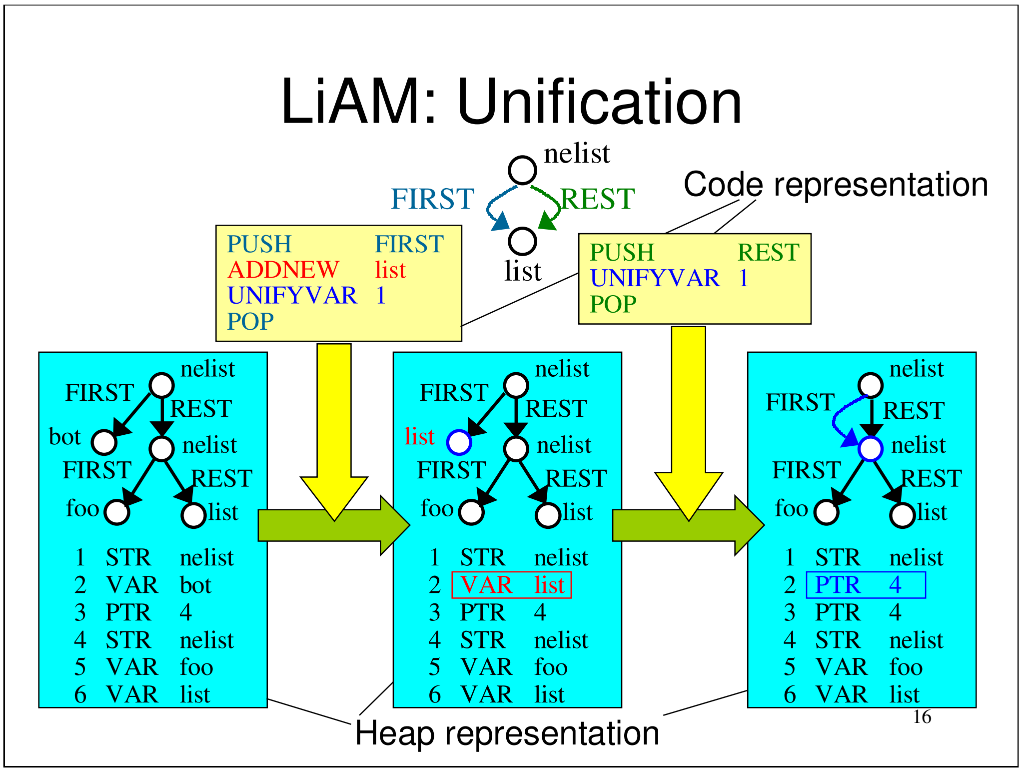


Unification part is an abstract machine for attribute-value logics. In short, the abstract machine gives two types of representation for a typed feature structure: Heap representation and Code representation.

Heap representation is a direct mapping from the graph notation of the typed feature structure. Each cell on the heap corresponds to a node or an edge of the graph notation.

On the other hand, code representation is a sequence of instructions. Each instruction corresponds to a constituent element of typed feature structure.

As you know, unification is an operation that takes two typed feature structures as inputs. In LiLFeS, it is the same, but one typed feature structure is code-represented, and the other typed feature structure is heap-represented.



This figure illustrates unification operation. These are code representation corresponds to this Typed Feature Structure. PUSH instruction corresponds to this feature, and ADDNEW instruction corresponds to the type of this node. By executing this code, the type of this node changes into LIST. These two UNIFYVAR instructions make a structure-sharing as this.

This unification method is efficient than unifying two heap-represented typed feature structure. This is because the operations are serialized in the code-representation, and it is not necessary to explore both typed feature structures.

LiAM: Clause Part (WAM)

- Full Prolog features of WAM
  - Calling Convention
  - Environment Stack Frame
  - Clause Backtracking
    - Operations of Cut, If-Then-Else, Negation as failure
  - Optimization Techniques
    - Last call optimization
    - Argument indexing

17

The other part of LiAM, the clause part, is Warren's Abstract Machine. LiLFeS Abstract Machine incorporates these features in order to realize Full Prolog features on LiLFeS. As a result, LiLFeS can handle backtracking, cut, and so on. We also introduced WAM's optimization techniques, such as argument indexing, and gained efficiency comparable to WAM.

LiLFeS Built-in Predicates

- Designed for use in efficient NLP systems
- Tuned up for speed

String Operations	Array operation
Process Control	Associative Array
Equivalence Check	TFS Copy
`findall` predicate	CFG CKY Parser

and more...

18

In addition to these features, LiLFeS incorporates a set of efficient built-in predicates. Instead of implementing many general-purpose built-in predicates, we chose built-in predicates useful for efficient NLP systems, and tuned up for speed. Later I will describe how these built-ins support efficient NLP applications.

Current Status

- LiLFeS Interpreter
  - Written in C++, more than 25,000 lines
  - High portability, on either UNIX or Windows
- Practical HPSG grammars
- Efficient HPSG parsers
- All will become public soon...

19

Next, I will describe about current status of LiLFeS.

We wrote LiLFeS interpreter in C++. The program is more than 25,000 lines, and still growing. LiLFeS interpreter can be compiled by GCC, and can run on either UNIX or Windows PC.

There are also several HPSG grammars and parsers on LiLFeS.

All of them are not distributed now, but will become public soon, I hope.

HPSG Grammars on LiLFeS

- Japanese Grammar
  - Made from scratch
  - Can parse 88% of EDR corpus
  - Can give a correct parse tree for 60% of the corpus
- English Grammar
  - Converted grammar from XTAG (Tateishi, 97)
  - Lexical entries for 350,000 words
  - Under construction

20

Let me describe more deeply about grammars and parsers on LiLFeS.

We are currently developing two grammars based on HPSG.

We are making Japanese grammar from scratch. The practical grammar is already working, and can parse 88% of EDR corpus, and can give a correct parse tree for 60% of the corpus.

On the other hand, English grammar is provided by converting from XTAG grammar, which is developed by the University of Pennsylvania. We succeeded to convert a small version of this grammar, which has 600 lexical entries. The full version of this grammar contains 350 thousands lexical entries, and we are now working for conversion of this full version.

HPSG Parsers on LiLFeS (1/2)

- CKY-style Naïve Parser
  - Can parse <u>0.76sec per sentence</u>
    - Condition: 1000 sentences from EDR Japanese corpus (average length 20.8 words) parsed by Japanese Grammar, 879 sentences were parsed

21

We are also developing efficient parsers.

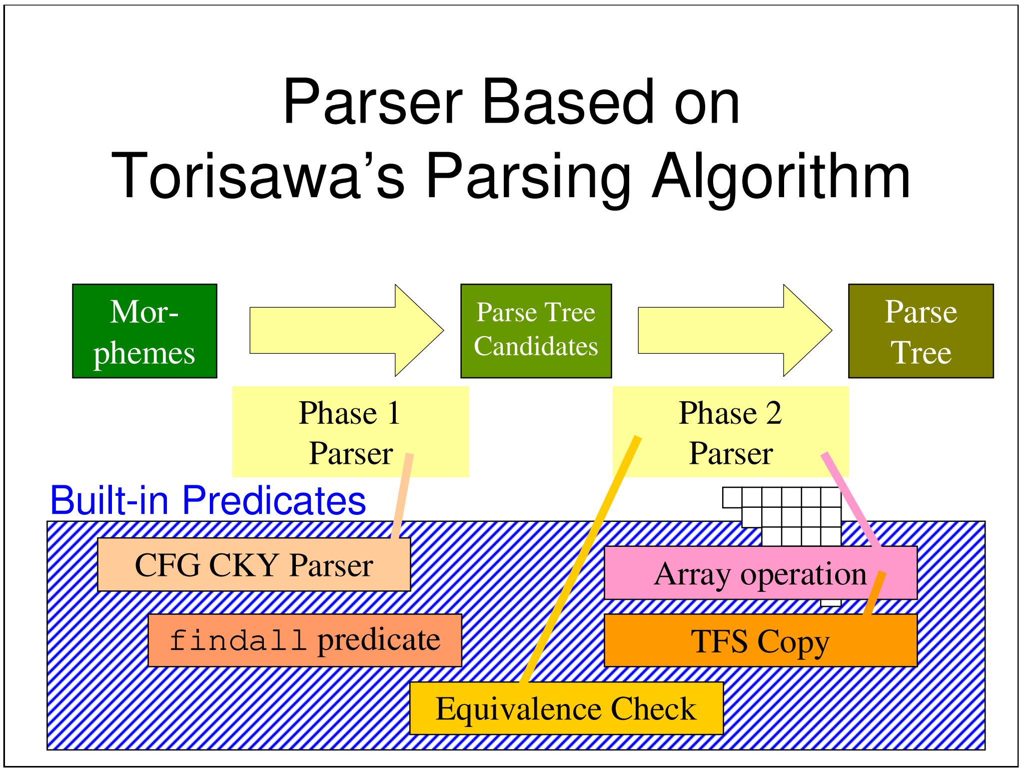
CKY-style naïve parser utilizes the full capabilities provided by LiLFeS, such as built-in predicates. This parser can parse EDR Japanese corpus in 760 millisecond per sentence, with practical HPSG grammar and EDR Japanese corpus.

HPSG Parsers on LiLFeS (2/2)

- Parser based on Torisawa's Parsing Algorithm (Torisawa, 96)
  - Compiles HPSG grammar into CFG and remaining part that is not covered by CFG
  - Still under development
  - Prototype: <u>0.51sec per sentence</u>

22

We are also developing a new parser based on Torisawa's Parsing Algorithm. This algorithm compiles HPSG grammar into a context-free grammar and the remainder part of the grammar that is not covered by CFG. The parser is still under development, but the prototype can parse in 510 milliseconds for the same grammar and sentences.



This figure illustrates the parser, and how built-in predicates support efficiency of NLP systems.

The parser parses in 2 phases. Phase 1 is a CFG parser, and selects parse tree candidates by parsing the input roughly. This part can be executed very efficiently because LiLFeS has efficient built-in CFG parser.

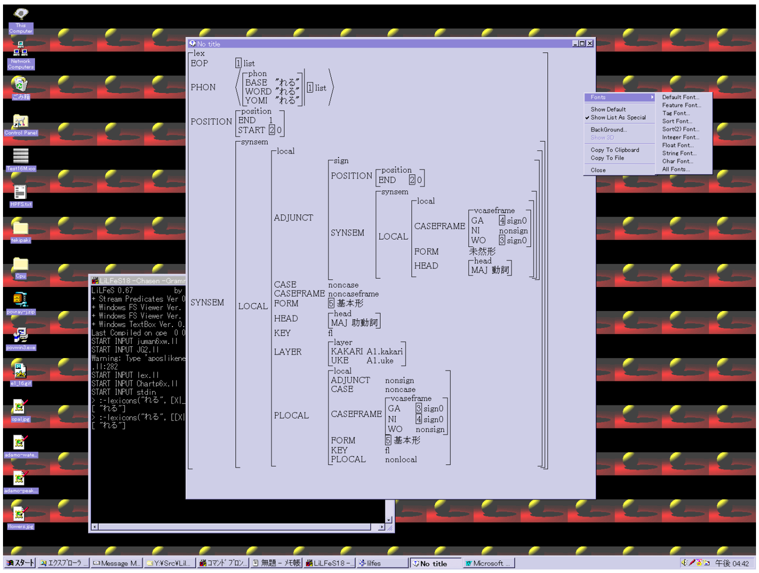
Phase 2 parses the input according to the parse tree candidates, and produces actual parse trees. In this parser, calculated subtrees are stored in an array, which is also provided in built-in predicates. Copy built-in is used to store a value into an array.

We also provide an efficient equivalence-check predicate for parsers. Equivalence check is not easy because typed feature structure may be spread on the heap. However, when two typed feature structure are just copied, equivalence check between them can be performed very efficiently, because they are collected into a chunk of memory. We can make use of this merit for efficient factoring in the parser because values in an array are already copied at storing.

In this way, built-in predicates provide efficiency for NLP applications.

LiLFeS GUI

Screenshot of the prototype



24

We are also developing a LiLFeS Graphical User Interface. This is a screenshot of the prototype.

Outline

- Goal
- Background
- Design Policy
- Overview of LiLFeS Language
- Architecture of the LiLFeS Interpreter
- Current Status
- Performance Evaluation
- Future of LiLFeS System

25

At last, I will show the performance evaluation, and the future of the LiLFeS system.

Performance Evaluation

- Unification performance test
  Comparison to 2 systems, without built-ins
  - ALE (Carpenter et al, 94) : Parser that interprets feature structures, written in Prolog
  - ProFIT(Erbach, 94): Compiler from feature structures to Prolog terms
- Parsing performance test with built-ins
  - Evaluation of parsing time for real-world text

26

We evaluated the performance of LiLFeS in two aspects.

The first test is an evaluation on typed feature structure handling. We compared LiLFeS to ALE and ProFIT.

The second test is an evaluation on parsing, and measured parsing time for real-world text.

Performance Evaluation: Performance without Built-ins

of parsed sentences per second

		(α)	(β)
DEC Alpha	LiLFeS Interpreter	347.22	4.63
	ProFIT($*$)	425.77	7.12
	ALE	5.902	0.126
Sun Ultra-Sparc	LiLFeS Interpreter	217.10	1.61
	ProFIT($*$)	342.86	7.86
	ProFIT($\dagger$)	677.39	15.39

Simple chart parser with a toy grammar

(α) 9-word sentence, 1 output (β) 9-word sentence, 64 outputs
($*$) On SICStus WAM interpreter ($\dagger$) On SICStus native-code compiler

27

This table shows the result of the performance comparison.

The application is a simple bottom-up parser with a toy grammar, and these digits are the number of parsed sentences per second.

As you can see, LiLFeS is more than 50 times faster than ALE. But, unfortunately, LiLFeS is about 1.5 times slower than ProFIT. In the case of ProFIT on a Prolog native code compiler, the gap becomes larger.

Performance Evaluation: Parsing with Built-ins

- CKY-style Naïve Parser without built-ins
  - <u>1.75sec per sentence</u>
- CKY-style Naïve Parser with built-ins
  - <u>0.76sec per sentence</u>
- Parser based on Torisawa's Algorithm
  - Prototype: <u>0.51sec per sentence</u>
- LiLFeS can outperform ProFIT

Condition: 1000 sentences from EDR Japanese corpus (average length 20.8 words) parsed by Japanese Grammar, 879 sentences were parsed

28

However, we can make up the gap with the efficiency of built-in predicates.

This is the result of parsing evaluation.

As you can see, the performance of CKY-style parser becomes more than 2 times faster by using built-in predicates. The parser based on Torisawa's algorithm is still a prototype, but it can parse a sentence in about 500 ms.

From these results, I can say that LiLFeS can outperform ProFIT in the parsing performance.

Future of LiLFeS Systems

- LiLFeS Native-code compiler
  - Prototype running, 10 times faster in simple Fibonucci
- Parallel parser on LiLFeS
  - CKY-style parser is parallelized
  - 10 times speedup on 60 processors

29

Note that this is not the final speed of the system. The speed will be further improved by the improvements of LiLFeS itself.

As one approach, a LiLFeS native-code compiler is on the way. The prototype is already running, and it outperforms current LiLFeS by ten times on a simple Fibonucci calculation program.

Another approach is parallelization. We didn't take the way to parallelize LiLFeS itself since this is too expensive to develop. We therefore parallelize only the parser on LiLFeS. The CKY-style parser on the LiLFeS Interpreter is already parallelized. It gained 10 times speedup on 60 processors. When the parser based on Torisawa's parsing algorithm is parallelized on the native-code compiler, the parser will achieve greater performance.

Conclusion

- We developed LiLFeS,
  a unification-based programming language
  with typed feature structures
- Efficient abstract machine and
  built-in predicates are designed for
  efficient NLP systems
- LiLFeS is still developing rapidly,
  and getting more and more powerful

30

Let me quickly summarize.

We developed LiLFeS, a unification-based programming language with typed feature structure.

LiLFeS has efficient abstract machine and built-in predicates, and we can construct efficient NLP systems on LiLFeS.

LiLFeS is still developing rapidly, and getting more and more powerful.

Thank you very much for your attention.