Efficiency in Unification-Based Processing

Stephan Oepen, Daniel Flickinger, Jun-Ichi Tsujii, and Hans Uszkoreit (editors)

August 28, 2000

CENTER FOR THE STUDY
OF LANGUAGE
AND INFORMATION

Contents

1

Native-Code Compilation of Feature Structures

Takaki Makino, Yusuke Miyao, Kentaro Torisawa, and Jun-ichi Tsujii

1.1 Introduction

LiLFeS (Makino, Torisawa, & Tsujii, 1997) is a solution for providing a programming environment with efficient processing of typed feature structures (TFSs) by an abstract machine approach. While optimization methods specific to HPSG parsing have proven to drastically increase parsing speed (Kiefer, Krieger, Carroll, & Malouf, 1999; Oepen & Carroll, 2000; Torisawa, Nishida, Miyao, & Tsujii, 2000), efficient feature structure processing is still required not only for efficient parsing but also for various applications of unification-based processing.

The existing LiLFeS system (*LiLFeS-1*) has been developed as a byte-code emulator of an abstract machine. Because of its efficiency and seamless design of feature structure and program descriptions, large-scale applications can be easily developed and efficiently executed on the system. Currently implemented applications include wide-coverage Japanese and English grammars (Mitsuishi, Torisawa, & Tsujii, 1998; Tateisi, Torisawa, Miyao, & Tsujii, 1998), a statistical disambiguation module for the Japanese grammar (Kanayama, Torisawa, Mitsuisi, & Tsujii, 2000), and the LinGO grammar translated to LiLFeS (Miyao, Makino, Torisawa, & Tsujii, 2000).

In this chapter we propose the *LiLFeS-II* system, a further improvement of LiLFeS by an exhaustive optimization of feature structure unification. In order to eliminate redundant operations in unification code, our new system is totally redesigned along the following policies.

Fine-grained instruction set. Each instruction in the LiLFeS-II ab-

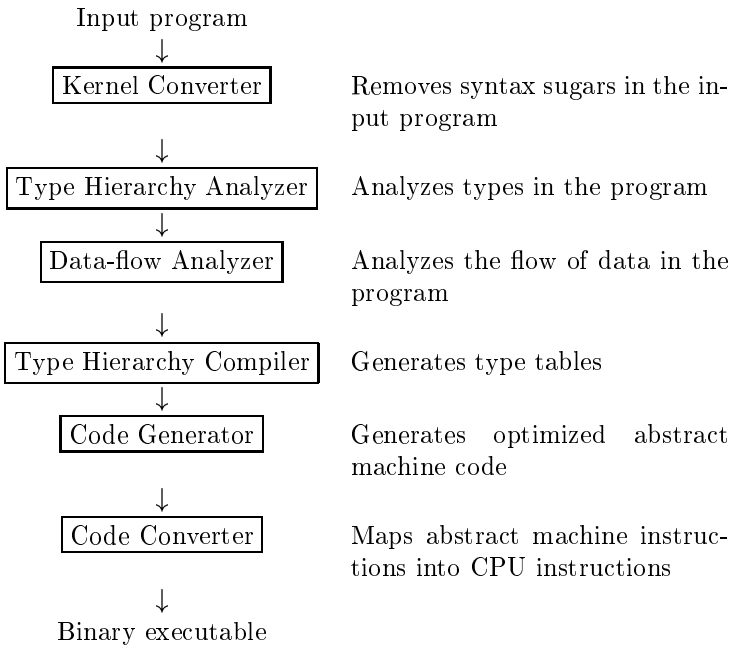


FIGURE 1 The LiLFeS native-code compiler

stract machine corresponds to a more low-level operation than that in the LiLFeS-1 abstract machine. This enables us much deeper and finer optimizations (described in Section 1.4).

Static program analyzers. The LiLFeS-II system analyzes input programs thoroughly by the Type Hierachy Analyzer and the Data-flow Analyzer. The output of these analyzers helps us to determine optimizable operations in the compiled code (described in Section 1.5).

Native-code compilation. The design of the LiLFeS-II instructions are intended to map its code directly into CPU-level instructions and run directly on the CPUs. The fine-grained instructions cannot be used without native-code compilation because the overhead of executing each instructions is too heavy in a byte-code emulation mechanism. We should note that simple-minded native code compilation does not bring much efficiency; optimization by above two analyzers are essential for efficient unification.

Figure 1 shows the flow of compilation in the LiLFeS-II system. First, Kernel Converter removes syntax sugars in the program to make

the compiler simple. Then two analyzers, Type Hierarchy Analyzer and Data-flow Analyzer, are invoked to gather information from the program. After that, Code Generator compiles a LiLFeS program into LiLFeS-II abstract machine instructions, using information from the static analyzers. Finally, machine-dependent Code Converter converts the instruction sequence into an assembly code output. This chapter does not describe Kernel Converter and Code Converter since they are not major parts for our optimization techniques.

Section 1.2 overviews the LiLFeS language. Section 1.3 describes the design of the LiLFeS system to efficiently execute the program in the LiLFeS language. Section 1.4 describes the LiLFeS-II abstract machine. Section 1.5 describes two static analyzers, Type Hierarchy Analyzer and Data-flow Analyzer. Section 1.6 describes the compilation of the LiLFeS program into abstract machine code, and optimization using the output of the static analyzers. Section 1.7 evaluates the performance of LiLFeS-II against LiLFeS-1 and LKB (Copestake, 1992) through the empirical results on the parsing performance with the translated LinGO grammar.

1.2 LiLFeS as a programming language

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

Figure 2 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 2) and definite clause programs with feature structures (the lower section in Figure 2). 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. In

```
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

FIGURE 2 A sample program written in the LiLFeS language

`head_feature_principle`, the values folowed by the `HEAD` feature of `$HEAD_DTR` and that of `$MOTHER` are structure-shared because they are indicated by the same variable `$HEAD`.

The LinGO grammar is written in $\mathcal{TDL}$ (Krieger & Schäfer, 1994), and is successfully translated to LiLFeS (Miyao et al., 2000). The performance evaluation with the LinGO grammar is reported in Section 1.7.

1.3 The LiLFeS-1 architecture

The LiLFeS-1 architecture is designed for efficient processing of TFSs as well as definite-clause programs. This section describes the key ideas for the effiency: i) the structure of the LiLFeS abstract machine, ii) the efficient representation of TFSs on the memory, and iii) the compilation of a TFS into an instruction sequence.

In this section, we first describe the architecture of LiLFeS, which is the common basis of the LiLFeS-1 and LiLFeS-II. After that, we overview the idea of compiling a TFS.

1.3.1 Structure of the LiLFeS abstract machine

LiLFeS-1 abstract machine, as well as LiLFeS-II abstract machine, is a virtual machine which is designed to perform TFS unification and controls execution of definite clause programs. LiLFeS-1 abstract machine

is designed as an amalgamation of WAM (Aït-Kaci, 1991) for definite-clause handling and AMAVL (Carpenter & Qu, 1995) for TFS unification, and implemented by a byte-code emulator on a real machine. LiLFeS-II abstract machine also inherits the design of LiLFeS-1 although its implementation is in a different and further optimized way.

The LiLFeS abstract machine consists of the following components, adopted from WAM and AMAVL:

- Static storages:
  Code area stores compiled code of TFSs and a type hierarchy.
  Type unification table is a two-dimensional array holding the result of unifying two types.
  Type instruction table is a two-dimensional array holding pointers to the code used in a general unification routine.
  Feature offset table holds feature offset of a type (mentioned in 1.3.2)
- Dynamic storages:
  Heap (global stack) consists of *cell*s and holds TFSs generated at run-time.
  Environment stack (AND stack) preserves variable values at predicate call.
  Choice-point stack (OR stack) holds backtracking information (variable and stack binding, etc.)
  Trail stack keeps track of overwritten cells that needs to be restored on backtracking.
  Push-down list stacks TFSs remaining to be unified.
- Registers: Code pointers, Pointers for dynamic storages, backtracking and cut controls, bindings of temporary variables, etc.

All the type tables are from AMAVL, and the other storages and registers are implemented based on WAM. We do not discuss them anymore; refer their original papers (Aït-Kaci, 1991; Carpenter & Qu, 1995) for details.

1.3.2 Representation of TFSs on the memory

Both LiLFeS-1 and LiLFeS-II abstract machines use the AMAVL-style TFS representation on the memory (Carpenter & Qu, 1995). Although the TFSs are restricted to totally well-typed feature structures, efficient TFS manipulation is possible on this style of TFS representation. We briefly describe this representation in this section.

A cell, the data unit to represent TFSs, is a pair of a tag and a value. Figure 3 shows the representation of various TFSs on the LiLFeS abstract machine. In this chapter, a cell with a tag `TAG` and a value

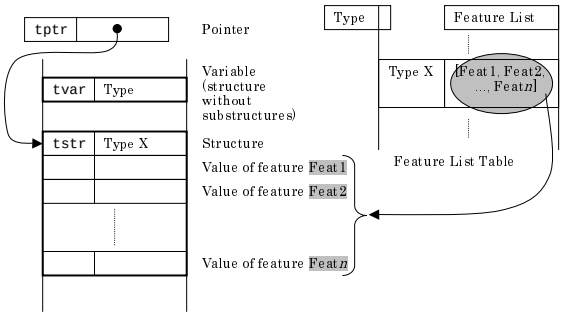


FIGURE 3 Memory representation of a TFS

`VALUE` is denoted as `TAG^VALUE`.

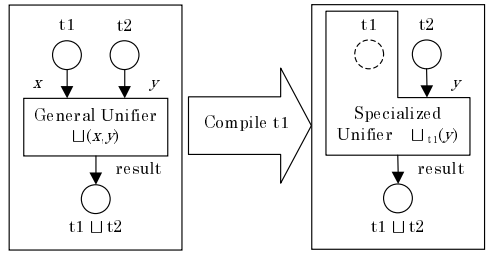
A TFS with n features is represented by $n + 1$ contiguous cells on the Heap. The first cell tagged as `tstr` holds the type of the structure, and each of the remaining n cells holds a pointer to the substructures of a feature, or the value of a feature if it is represented in a single cell. The correspondence between the features and the cell positions is stored in Feature offset table. If a feature structure has no features, or all substructures are their default values, the structure is represented by a cell with `tvar` tag instead of a full representation with a cell block with a `tstr` tag.

A cell with a `tptr` tag is a pointer and used to refer a TFS on another location. Pointers can be chained until it reaches a cell with other than a `tptr` tag; we do not use self-referencing pointers to represent unbounded variables as in WAM and Aquarius Prolog. A structure-sharing is represented by referring the same location.

1.3.3 Compiling a TFS

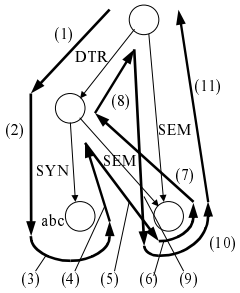
Unification is an operation defined between two feature structures. When both input structures are given at run-time, the only method to unify them is to traverse them at run-time as illustrated in Figure 4 (a). Hereafter we call a unification routine based on this method, a general unifier.

However, in many cases, one of the two structures is known in advance at compile-time. For example, the unification in `head_feature_principle` in Figure 2 is given in the program and does not change during run-time. We can compile such a TFS into specialized code for unification, as illustrated in Figure 4 (b), rather than using a general unifier. The compiled unification routine can be much more efficient than a general unifier because we can optimize the unifier at compile-time to



(a) General Unification (a) Specialized Unification

FIGURE 4 General and specialized unification



(1)	PUSH	DTR
(2)	PUSH	SYN
(3)	ADDNEW	**abc**
(4)	POP	
(5)	PUSH	SEM
(6)	UNIFYVAR	1
(7)	POP	
(8)	POP	
(9)	PUSH	SEM
(10)	UNIFYVAR	1
(11)	POP	

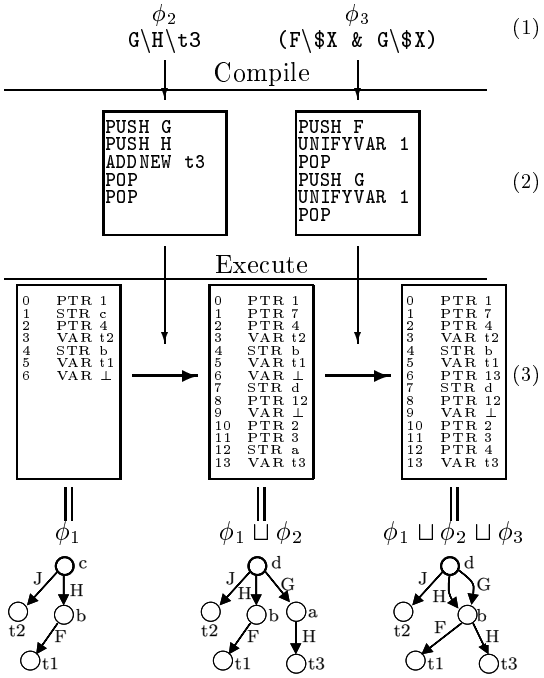
(a) Feature Structure F (b) Compiled code C

FIGURE 5 Compilation in LiLFeS-1 system

minimize the operation on the other structure at unification time, while any general unifier has to traverse both of the input structures at each unification.

The idea is similar to the compilation of Prolog first-order terms (e.g. WAM (Aït-Kaci, 1991) and Aquarius Prolog (Van Roy, 1990)), application of these studies to feature structure unification is not straightforward due to the differences between Prolog terms and TFSs, such as fixedness of arity, existence of structure-sharing, and type hierarchy.

AMAVL (Carpenter & Qu, 1995) and AMALIA (Wintner, 1997) are the pioneers for TFS compilation. Both compile the type hierarchy and given TFSs into an instruction sequence of an abstract machine, and emulate the execution of the abstract machine instructions by another program. This design is inherited to the LiLFeS-1 system, which exntended the design of AMAVL and integrated execution of definite-clause



(1) TFS of LiLFeS program, (2) Code representation, (3) Heap Representation

FIGURE 6 Unification process in LiLFeS-1 system

programs.

In the proposal of AMAVL, four kinds of instructions, ADDNEW, PUSH, POP and UNIFYVAR, are used to describe the compiled TFS. These instructions form a traversing process of a TFS. PUSH and POP correspond to following and retreating a feature respectively, ADDNEW corresponds to a type unification, and UNIFYVAR corresponds to structure-sharing. Figure 5 shows the correspondence between the instructions and the traversing process.

When a compiled code from a feature structure ϕ_2 is executed, the code modifies another feature structure ϕ_1 on the memory to the unified result $\phi_1 \sqcup \phi_2$. Figure 6 shows an example of execution. This unification is more efficient than unifying two TFSs on memory, since we do not need to inspect and traverse both TFSs recursively.

1.4 Architecture of the LiLFeS-II abstract machine

In this section we describe the architecture of the LiLFeS-II abstract machine. Since the major difference between the abstract machine ar-

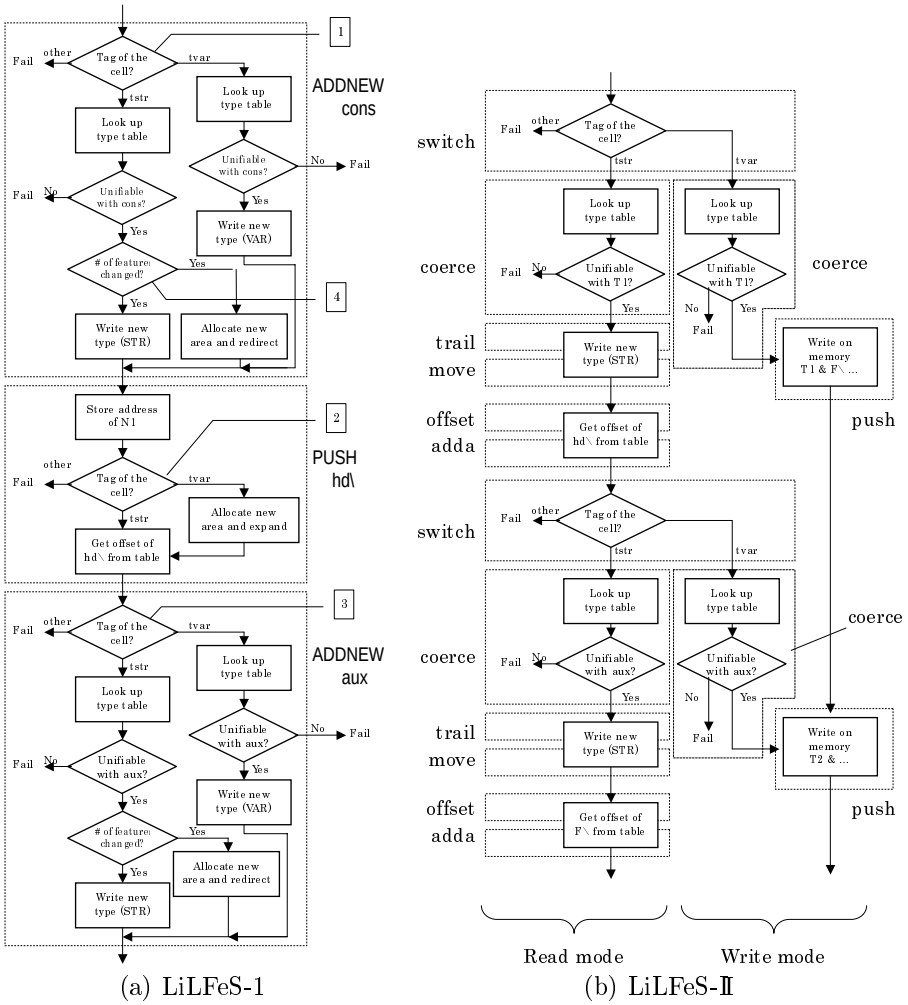


FIGURE 7 Difference of unification operation between LiLFeS-1 and LiLFeS-Ⅱ

chitecture of LiLFeS-1 and that of LiLFeS-Ⅱ is their instruction set, we focus on the instruction set in the LiLFeS-Ⅱ in this section. See section 1.3 for the other part of the architecture.

1.4.1 Problem of the compilation in the LiLFeS-1 system

Once we inspect the unification process in AMAVL and in LiLFeS-1, we can see many redundant operations in the compiled code. Figure 7 (a)

illustrates how LiLFeS-1 abstract machine operates in compiled unification code of `cons & hd\(aux & ...)` [1]. Each dashed box represents one instruction. From this chart, you can see that some conditional branches are redundant. For example, conditional branches 1 and 2 check the same thing, and can be combined. Moreover, another conditional branch 3 goes to `tvar` whenever the branches 1 and 2 were `tvar`. In addition, if we can use the information of the type hierarchy, some more operations become unnecessary. For example, a conditional branch 4 always goes to 'no' if any node with the type `cons` (a constituent of a list) has always two features, and this number does not change in the given type hierarchy.

We have few ways to optimize these redundancies in the architecture of LiLFeS-1 abstract machine, since they are embedded in a single instruction. In order to remove such redundant operations, the abstract machine have to either (i) provide an optimized version of the instructions, or (ii) divide an instruction into two or more fine-grained instructions. Both ways seem problematic; the former causes us to provide tons of slightly different versions of instructions (our preliminary study found that we will need more than 50 variations for a PUSH instruction). On the other hand, the latter increases the emulation overhead of the abstract machine, since the overhead is proportional to the number of emulating instructions. We need to divide one LiLFeS-1 instruction into 10 or more fine-grained instructions for a certain level of optimization, and 10 times of emulation overhead will drawn out the benefit of optimization.

In the architecture of LiLFeS-II abstract machine, we choose the latter way, and adopt native-code compilation to avoid its problem. That is, use a fine-grained instruction set to make extensive optimization possible, and map the abstract machine instructions to native-code instructions so that the latest CPU can perform unification without emulation overhead.

1.4.2 Instruction set

We totally redesigned the instruction set in the LiLFeS-II. Basic instructions and definite clause control instructions are based on Berkeley Abstract Machine (Van Roy, 1990), and instructions for manipulating TFSs are newly designed for LiLFeS-II.

In the new instruction set, we tried to make instructions as fine-grained as possible. Many instruction corresponds to one or a few CPU instructions; for example, instruction `move` can be mapped to one in-

[1] Note that this figure is simplified from the actual code and operation.

struction in the CPU instructions. Thus optimization on the abstract machine code results in optimization in CPU instruction level.

For simplicity, some *complex instructions* are introduced, which is mapped to many (more than five) CPU instructions. Examples are `deref` (pointer chain dereference), `choice` (backtracking control), and `coerce` (Type table access). They are introduced since their operations are atomic, that is, the operation will never be partially optimized by the information from static analysis.

The complete instruction set of the LiLFeS abstract machine is shown in the Appendix A. In the next two section we describe the method to compile a TFS into optimized code using this instruction set.

1.5 Static analyzers

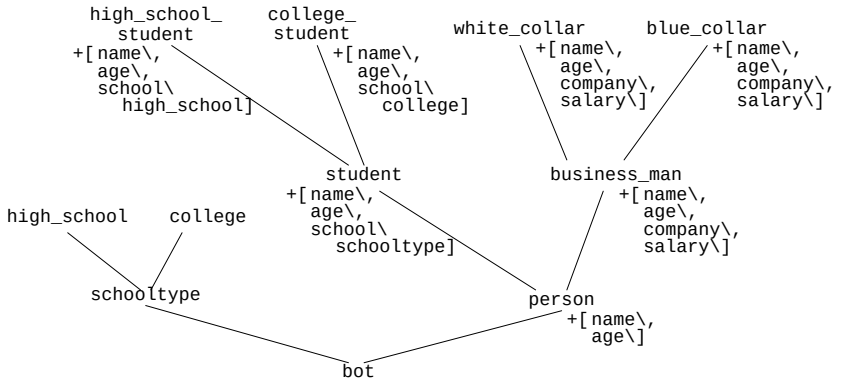
LiLFeS-Ⅱ native-code compiler runs two static analyzers on the source code in order to optimize the output code. One is Type Hierarchy Analyzer, which restricts the possible unification results of each type and helps us reduce unused conditional branches in code generation. The other is Data-Flow Analyzer, which gives value information of variables in definite clause programs by abstract interpretation technique(Van Roy, 1990). In LiLFeS, the most important information for optimization is the lower bound of variables, that is, the most general feature structure that the variable can have. This information also gives us a chance to reduce unnecessary conditional branches.

1.5.1 Type hierarchy analysis

Some types do not require 'full' treatment of unification. For example, type `nil`, which marks the end of a list, cannot have any feature, thus the part of the unification code that deals with features and substructures is not necessary. In this case, the lighter unification code without treatment of features and substructures can be used.

A condition of a type is useful for optimization when it determines conditional branches in the unification algorithm. The followings are the conditional branches used in the code generation, where T is the compiling TFS and V is another TFS given at run-time:

- Which tag V has (`tvar`, `tstr`)
- Whether the type of V is compatible to the type of T.
- What is the unified type of V and T.
- Whether the unified type t is equal to the original type of V.
- Whether t has additional features or more strict appropriateness from those of the type of V.
- In which offset the required feature of the structure V is.



Each line denotes that the upper type is the subtype of the lower type.
Notation "`+ [feat\ app, ...]`" says that the features `feat` and its appopriate type `app` (omitted if `bot`) is specified to the type.

FIGURE 8 Sample type hierarchy

From the above we derived the following conditions useful for optimization according to the type t.

Condition *Final* True if the type t has no subtypes. In this case, unified result type is always equal to the original type.

Condition *Never_Feature* True if the type t cannot be unified with any type that have features. In this case, any V with `tstr` tag results in a failure.

Condition *Preserve_StrType* True if any type that have features sufferes no change from the unification with t. In this case, the compiled code never update the type if V has `tstr` tag.

Condition *Fixed_NF* True if any subtype of the type t have no additional feature. Condition *Final* implies this condition.

Condition *Preserve_NF* True if a TFS of any type suffers no change on its feature set from successful unification with the type t. Condition *Preserve_StrType* implies this condition.

Condition *Fixed_Approp* True if any subtype of t has the same appropriateness condition and constraints as t has. Condition *Final* implies this condition.

Condition *Preserve_Approp* True if a TFS of any type suffers no change on its appropriate conditions or constraints from successful unification with the type t. Condition *Preserve_StrType* implies this condition.

Type \ Condition	*Final*	*Never_ Feature*	*Preserve_ StrType*	*Fixed_ NF*	*Preserve_ NF*	*Fixed_ Approp*	*Preserve_ Approp*
`bot`							
`schooltype`		X					
`high_school`	X	X					
`college`	X	X					
`person`			X		X		X
`student`				X			
`high_school_student`	X			X		X	
`college_student`	X			X		X	
`business_man`				X		X	
`white_collar`	X			X		X	
`blue_collar`	X			X		X	

X denotes the type satisfies the condition.

TABLE 1 Condition table of the sample type hierarchy

Suppose a type hierarchy is given as in Figure 8. Optimization conditions of the types on the type hierarchy is shown in Table 1. As you can see, most of the types satisfies some conditions in this example. This means that unification code of these types can be optimized.

The type `schooltype` satisfies *Never_Feature* condition, since all of its subtype has no feature. This condition can reduce compiled code drastically, since the no check for features are necessary.

The type `person` satisfies *Preserve_StrType* condition, since any type with features suffers no change from unifying this type. Code for changing type on a `tstr`-tagged feature strcture can be safely removed from compiled code for this type.

The type `student` satisfies *Fixed_NF* condition, since any subtype of this type has no additional features. This lets us replace a table access for an offset of substructures to a fixed integer value.

These optimization conditions are passed to the compiled code generator to help generation of an optimized code. The generation of the code is desctibed in the Section 1.6.

1.5.2 Global data-flow analysis

Global data-flow analysis is a method to derive information of variables from the source code. Aquarius Prolog (Van Roy, 1990) performs the global data-flow analysis and obtains speed-up around 10%. Since TFSs have types that Prolog does not have and data-flow analysis gives type information for variables, the benefit is expected to be larger than that on Prolog.

In the LiLFeS-II native-code compiler, global data-flow analysis module developed by Yoshida (Yoshida, Makino, Torisawa, & Tsujii, 1998) is

implemented. Data-flow analyzer traces the execution of definite clause programs with abstract value of variables and obtain the lower-bound value of a variable, that is, the common-part (meet) of all the possible values of the variable.

Code generator uses this lower-bound value information to optimize the generated code in the following ways:

T is subsumed by the lower-bound value of V

In the case that T is equal to, or more specific than, the lower-bound value of V, all of the unification operations are not necessary except binding variables in T to V.

Lower-bound value with substructures

When the lower-bound value has substructures, it means that the variable V always holds a value with `tstr` tag. This means that code for tag checking and code corresponding to `tvar` can be safely removed.

Lower-bound value restricts unifiable type set

A table access can be removed if *unifiable type set,* a set of types that is possible to appear as V and compatible to T, is small enough. For example, suppose the lower-bound value of V is `list`, and T's type is `nil`. Since Type Hierarchy Analyzer determins that `cons` is *Final*, it restricts that unifiable type set to $\{\texttt{list}, \texttt{nil}\}$. Thus we can replace the `coerchk` instruction, which accesses Type table to check whether V's type is unifiable, into a set of simple conditional jumps.

1.6 Compilation

The Code Generator in the LiLFeS-II native-code compiler compiles a LiLFeS program into an instruction sequence. Since the instruction set is fine-grained, we can optimize the code by the result of static analyses in Section 1.5. In this section we describe the actual algorithm to generate abstract machine code of feature structure unification, and how the optimization condition are used to optimize the code.

In the following, the abstract machine code for the unification $V = T$ is provided, where V and T are TFSs and T is given at compile-time. The code is described for each case of the representation of T, `tvar` or `tstr`.

1.6.1 TFS without substructures

If T has no substructures, the unification action is mainly a manipulation of types, that is, updating V's type to the unified type. Since the memory representation of V can be tagged by either `tvar` or `tstr`, we need a separated instruction sequence for these tags. Note that, in case of `tstr`,

Instruction		Description
deref	V, W, X	Dereference the pointer V
switch	tstr, X, L1, L2, fail	Branch according to the tag: tvar or tstr.
label L1		Case of tvar
exttype	X, Y	Extract V's type to Y
coerce	Y, t_T, Z, fail	Get the unified type (fail if incompatible)
jump	eq, Y, Z, L3	If Y=Z, jump to the exit
trail	W	In case of Y≠Z, save the original value for backtracking
move	tvar^Z, [W]	Overwrite the new value (tvar^t')
jump	L3	Jump to the exit
label L2		Case of tstr
exttype	X, Y'	Extract V's type to Y'
coerce	Y', t_T, Z', fail	Get the unified type (fail if incompatible)
jump	eq, Y', Z', L3	If Y'=Z', jump to the end
jccifa	Y', Z', W, L3	Jump to the general unification routine if needed
trail	W	If no operations are needed, save the original value for backtracking
move	tstr^Z', [W]	Overwrite new value (tstr^t')
label L3		End of unification

FIGURE 9 The most generic code for $V = T$ when T do not have substructures

```
deref r0, r1, r2
switch tstr, r2, L1, L2, fail
label L1
 exttype r2, r3
 coerce r3, t_T, r4, fail
 jump eq, r3, r4, L3
 trail r1
 move tvar^r4, [r1]
 jump L3
label L2
 exttype r2, r3
 coerce r3, t_T, r4, fail
 jump eq, r3, r4, L3
 jccifa r3, r4, r1, L3
 trail r2
 move tstr^r4, [r1]
label L3
```

(a) Most generic code

```
deref r0, r1, r2
exttag r2, r3
jump ne, r3, tvar, fail
exttype r2, r3
coerce r3, t_T, r4, fail
jump eq, r3, r4, L3
trail r1
move tvar^r4, [r1]
label L3
```

(b) Optimization by type hierarchy analysis

```
deref r0, r1, r2
jump eq, r2, tvar^nil, L3
jump ne, r3, tvar^list, fail
trail r1
move tvar^nil, [r1]
label L3
```

(c) Optimization by data-flow analysis

FIGURE 10 The code for $V =$nil

the change of the type may affect V's features according to the total well-typedness and constraints; In this case, we call the compiled code for general unification to perform the type-specific manipulation.

Figure 9 shows the most generic (i.e. unoptimized) abstract machine code for this case. Note that, in LiLFeS-1, only one instruction (ADDNEW) represents this sequence of operations. This means that now we are able to optimize the code by manipulating the code in the LiLFeS-II.

For example, suppose we are about to compile a unification "V = `nil`," where V is a variable and pointed by a register `r(0)`. The compiler has analyzed the type hierarchy and found that `nil` satisfies *Never_Feature* condition, i.e., any feature structure unifiable with `nil` has no features. In this case the code can be optimized from the most generic code, shown in Figure 10 (a), to the optimized code, shown in Figure 10 (b).

In addition to that, suppose the case that `nil` satisfies *Final* condition (i.e. `nil` has no subtype) and the data-flow analyzer detects the lower-bound of V is `list` (i.e. the unification V =`nil` is in the context where V is always `list` or its subtype). Although this example may look like a rare case, we often encounter such a case, e.g. the first element of the traditional `append` predicate. In this case, now it is unnecessary to access the type table, since the unification succeeds only if V is `list` or V is `nil`, and the result is always `nil`. The code can be further optimized as shown in shown in Figure 10 (c).

As you can see in this example, the code generater reduces unnecessary conditional branches and replaces some other condition checks to more efficient ones according to the information from static analyses. This optimization is the key to the efficient TFS unification.

1.6.2 TFS with substructures

When T has one or more substructures, the algorithm gets more complicated. However, the basic principle is the same as that of Prolog: to construct T on the fly if V doesn't have any substructures (write mode), or apply this algorithm recursively if V has substructures (read mode). In both modes, V's type is updated and features/constraints might be added according to the updated type.

Figure 11 is the most generic version of the abstract machine code. Vertical dots are the placeholder where generated code for substructures should be stored. One major difference of this code from LiLFeS-1 is that this supports 2-streamed unification, which is described in (Roy, 2000). To eliminate excess conditional jumps checking whether we are in read-mode or in write-mode, we split the compiled code into two streams, read-mode stream and write-mode stream, and connect them

Instruction		Description
deref	V, W, X	Dereference the pointer V
switch	tstr, X, L1, L2, fail	Branch according to the tag: whether `tstr` or `tvar`
label L2		Case of `tstr`
exttype	X, Y	Extract V's type to Y
coerce	Y, *type*, Z, fail	Get the unified type (fail if incompatible)
jump	eq, Y, Z, L6	In case of Y=Z, go to the recursive code directly
jccifa	Y, Z, W, L7	Call general unification if needed
trail	W	If no operations are needed, save the original value for backtracking
move	tstr^Z, [W]	Overwrite the new value (`tstr^t'`)
jump	L6	Jump to the recursive part
label L7		Return from general unification
deref	W, W	Extra dereference may be necessary
label L6		Start of the recursive part (read-mode)
offset	Z, feat1, O1	Read offset of the feature `feat1` from the table
adda	W, O1, V1	Calculate the address of the substructure in `feat1`
⋮		Recursive unification code for the substructure in `feat1` is put here
offset	Z, feat2, O2	Read offset of the feature `feat2` from the table
adda	W, O2, V2	Calculate the address of the substructure in `feat2`
⋮		Recursive unification code for the substructure in `feat1` is put here
⋮		Repeat until all necessary substructures are processed
jump	L3	Jump to exit
label L1		Case of `tvar`
exttype	X, Y'	Extract V's type to Y'
coerce	Y', *type*, Z, fail	Get the unified type (fail if incompatible)
trail	W	Save the original value for backtracking
move	r(h), [W]	Bind the variable to the top of Heap (though there is no structure yet)
jump	ne, *type*, Z, L4	Branch whether Z=*type*
push	tstr^*type*, r(h), 1	Store the top of the structure
jump	L5	Jump to the recursive part
label L4		
jecifa	type, Z, W, L7	Call general unification if needed
push	tstr^Z, r(h), 1	if no operations are needed, store the top of the structure
jump	L5	Jump to the recursive part
label L5		Start of the recursive part (write-mode)
push	...., r(h), 1	Put the value of the feature `feat1`
push	...., r(h), 1	Put the value of the feature `feat2`
⋮		Build all substructures in the same way
jump	L3	
label L3		End of the unification

FIGURE 11 The most generic codefor $V = T$ when T has substructures

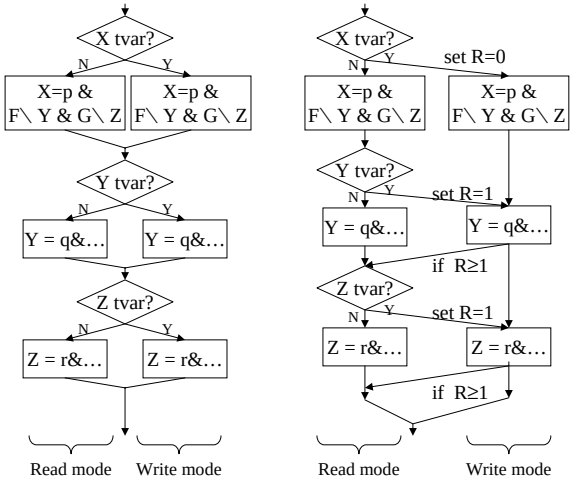


FIGURE 12 2-streamed unification

by conditional branches as shown in Figure 12. 2-streamed unification code makes write-mode unification more efficient because checking the value of R is much efficient than checking a tag on a cell, which includes memory access and dereferencing, [2].

Optimization to this code can be done in various ways. Suppose that the type of T satisfies conditions *Final*, *Fixed_NF*, *Fixed_Approp*, described in Section 1.5. These three conditions indicate that the result of the unified structure is partly predictable in compilation time. Thus the operations that uses the result type and feature offsets, which used to be retrieved from type table, can be reduced to operations with constant values, when the appropriate condition is true. An example of optimization on a unification code for TFS with substructures is shown in the following list.

Original Code	Optimized Code	Condition
`deref V, W, X`	`deref V, W, X`	
`switch tstr, X, L1, L2,`	`switch tstr, X, L1, L2,`	
`fail`	`fail`	
`label L2`	`label L2`	
`exttype X, Y`	`exttype X, Y`	

continued on next page

[2]Note that the introduction of 2-streamed unification makes it difficult to direct mapping from abstract machine instructions in LiLFeS-1 to those in LiLFeS-II.

Original Code	Optimized Code	Condition
`coerce Y, type, Z, fail`	**`coerchk Y, type, fail`**	*Final*
`jump eq, Y, Z, L6`	`jump eq, Y, type, fail`	*Final*
`jccifa Y, Z, W, L7`	`jccifa Y, type, W, L7`	
`trail W`	`trail W`	
`move tstr^Z, [W]`	`move tstr^type, [W]`	
`jump L6`	`jump L6`	
`label L7`	`label L7`	
`deref W, W`	`deref W, W`	
`label L6`	`label L6`	
`offset Z, feat1\, O1`		*Fixed_NF*
`adda W, O1, V1`	`adda W, 1, V1`	*Fixed_NF*
⋮	⋮	
`offset Z, feat2\, O2`		*Fixed_NF*
`adda W, O2, V2`	`adda W, 2, V1`	*Fixed_NF*
⋮	⋮	
`jump L3`	`jump L3`	
`label L1`	`label L1`	
`exttype X, Y'`	`exttype X, Y'`	
`coerce Y', type, Z, fail`	**`coerchk`** `Y', type, fail`	*Final*
`trail W`	`trail W`	
`move r(h), [W]`	`move r(h), [W]`	
`jump ne, type, Z, L4`		*Final*
`push tstr^type, r(h), 1`	`push tstr^type, r(h), 1`	
`jump L5`	`jump L5`	
`label L4`		
`jecifa type, Z, W, L7`		*Fixed_NF* and *Fixed_Approp*
`push tstr^Z, r(h), 1`		*Final*
`jump L5`		*Final*
`label L5`	`label L5`	
`push ...., r(h), 1`	`push ...., r(h), 1`	
`push ...., r(h), 1`	`push ...., r(h), 1`	
⋮	⋮	
`jump L3`	`jump L3`	
`label L3`	`label L3`	

Another optimization case is when the type of T satisfies conditions *Preserve_StrType*, *Preserve_NF*, *Preserve_Approp*. These three conditions indicate that the compiling unification does not affect to some elements of the unifying TFS. This means that instructions for changing these elements can be removed without any problem. An example of optimization on a unification code for TFS with substructure is shown in the following list.

Original Code	Optimized Code	Condition
`deref V, W, X`	`deref V, W, X`	
`switch tstr, X, L1, L2, fail`	`switch tstr, X, L1, L2, fail`	

continued on next page

Original Code	Optimized Code	Condition
`label L2`	`label L2`	
`exttype X, Y`	`exttype X, Y`	
`coerce Y, type, Z, fail`	**`coerchk Y, type, fail`**	*Preserve_StrType*
`jump eq, Y, Z, L6`		*Preserve_StrType*
`jccifa Y, Z, W, L7`		*Preserve_NF* and *Preserve_Approp*
`trail W`		*Preserve_StrType*
`move tstr^Z, [W]`		*Preserve_StrType*
`jump L6`	`jump L6`	
`label L7`	`label L7`	
`deref W, W`		*Preserve_NF*
`label L6`	`label L6`	
`offset Z, feat1\, O1`	`offset Z, feat1\, O1`	
`adda W, O1, V1`	`adda W, O1, V1`	
⋮	⋮	
`offset Z, feat2\, O2`	`offset Z, feat2\, O2`	
`adda W, O2, V2`	`adda W, O2, V1`	
⋮	⋮	
`jump L3`	`jump L3`	
`label L1`	`label L1`	
`exttype X, Y'`	`exttype X, Y'`	
`coerce Y', type, Z, fail`	`coerce Y', type, Z, fail`	
`trail W`	`trail W`	
`move r(h), [W]`	`move r(h), [W]`	
`jump ne, type, Z, L4`	`jump ne, type, Z, L4`	*Final*
`push tstr^type, r(h), 1`	`push tstr^type, r(h), 1`	
`jump L5`	`jump L5`	
`label L4`	`label L4`	
`jecifa type, Z, W, L7`	`jecifa type, Z, W, L7`	
`push tstr^Z, r(h), 1`	`push tstr^Z, r(h), 1`	
`jump L5`	`jump L5`	
`label L5`	`label L5`	
`push ...., r(h), 1`	`push ...., r(h), 1`	
`push ...., r(h), 1`	`push ...., r(h), 1`	
⋮	⋮	
`jump L3`	`jump L3`	
`label L3`	`label L3`	

You can see that optimizations in these examples are also based on information from the static analyzers. It is the combination of the deep static analysis and an optimizable fine-grained instruction set that enables optimization of the unification routine to this extent.

1.7 Evaluation

This section evaluates the performance of the LiLFeS-II system through parsing tasks with the LinGO grammar. The current implementation of the compiler generates Pentium assembly code, and the following experiments are performed on Intel Pentium III 550 MHz.

Table 2 shows the performance of the LiLFeS-II system compared to the LiLFeS-1 system and LKB. We used naive parser in the parsing test

		LiLFeS-Ⅱ	LiLFeS-1	LKB
Total time (sec)	csli	0.30	0.62	0.17
	aged	1.19	2.33	0.65
	fuse	8.53	12.46	2.85
etasks	csli	47,782	47,782	476
	aged	186,813	186,813	1,635
	fuse	1,160,797	1,160,797	5,946
stasks	csli	5,724	5,724	222
	aged	25,006	25,006	776
	fuse	144,528	144,528	2,695
etasks/sec	csli	159,273	77,067	2,800
	aged	156,895	80,177	2,515
	fuse	136,084	93,162	2,359
stasks/sec	csli	19,080	9,232	1,305
	aged	21,013	10,732	1,194
	fuse	16,943	11,599	945

TABLE 2 Evaluation of the LiLFeS-Ⅱ system

		LiLFeS-Ⅱ w/ analyzers	LiLFeS-Ⅱ w/o analyzers	LiLFeS-1
Total Time (sec)	csli	0.30	0.48	0.62
etasks/sec	csli	159,273	99,546	77,067
stasks/sec	csli	19,080	11,925	9,232

TABLE 3 Evaluation of the optimization

of LiLFeS systems [3]. In the results, LiLFeS-Ⅱ achieved a speedup of a factor of 1.5 to 2 from LiLFeS-1. Since the parsing algorithms are the same, this speedup is obtained by the efficiency of the LiLFeS-Ⅱ system.

Due to the difference of parsing algorithm, we cannot directly compare the efficiency of unification between LiLFeS-Ⅱ and LKB. However, the etasks/sec and stasks/sec give us information on unification efficiency. Even LiLFeS-1 have stasks/sec far better than LKB, LiLFeS-Ⅱ outperforms LiLFeS-1 in all testsuits.

Table 2 shows the performance obtained by the optimization techniques. The entry "LiLFeS-Ⅱ w/ analyzers" shows performance of fully optimized code, and "LiLFeS-Ⅱ w/o analyzers" shows performance of

[3]Since the naive parser uses different factoring algorithm from the naive parser in Torisawa's chapter, the parsers' performance is slightly different.

generated code without any static analyzers (In other words, a simple-minded native-code compiler of LiLFeS). As you can see, performance of LiLFeS-II gets much worse without analyzers and gets nearer to the performance of LiLFeS-1. From this result we can say that a simple-minded native-code compilation is not effective for the performance, and extensive optimization on the compiled code is indispensable to achieve high efficiency.

1.8 Conclusion

The native-code compilation of LiLFeS enables generating efficient code for feature structure unification. First, a fine-grained instruction set is proposed to remove redundancy of abstract machine code. Second, analyses of type hierarchy and data flow make further elimination of useless conditional branches in the compiled code. Evaluation with the LinGO grammar reports the improvement of parsing speed by a factor of 1.5 to 2 compared to the LiLFeS byte-code emulator. Integrated with other optimization techniques (Torisawa et al., 2000), LiLFeS achieves efficient processing with unification-based formalisms, such as HPSG.

A

LiLFeS Instruction set

Basic Instructions	
Instruction	Meaning
`move X,Y`	Move `X` to `Y`.
`push X,Y,N`	Push `X` on stack with stack pointer `Y` and post-increment `N`.
`adda X,N,Y`	Add an address offset. Advance the pointer `X` by `N` cells forward (backward if `N<0`) and store the result in `Z`.
`switch Tag,X,LV,LT,LF`	Three-way branch; branch to `LV`, `LT`, `LF` depending on whether the tag of `X` is `tvar`, `Tag`, or any other value.
`jump Cond,X,Y,L`	Jump to `L` if `Cond` is satisfied between `X` and `Y`.
`jump L`	Jump unconditionally to `L`.
`label L`	L is a branch destination.
`procedure P`	Mark the beginning of a procedure `P`.
`call P`	Call the procedure `P`.
`execute P`	Jump to the procedure `P`.
`return`	Return from a procedure call.

Complex Instructions	
Instruction	Meaning
`unify X,Y,Tx,Ty,L`	General Unification of `X` and `Y`, branch to `L` if fail. The extra parameters `Tx` and `Ty` give information to improve the translation (They are not needed for correctness).
`allocate X`	Create an environment of size `N` on the local stack.
`deallocate X`	Remove the top-most environment from the local stack.
`choice 1/N,RegList,L`	Create a choice point containing the registers listed in `RegList` and set the retry address to `L`.
`choice I/N,RegList,L` $(1 < \mathtt{I} < \mathtt{N})$	Restore the argument registers listed in `RegList` from the current choice point, and modify the retry address to `L`.
`choice N/N,RegList,L`	Restore the argument registers listed in `RegList` from the current choice point, and pop the current choice point from the choice point stack.
`cut X`	Make the choice point pointed to by `X` the new top of the choice point stack.

TABLE 4 The Instruction Set in LiLFeS-II (1) Imported Instructions from BAM

Memory Operations	
Instruction	Meaning
`deref X,Y,Z`	Dereference `X` and store the result and its pointer to `Z` and `Y`, respectively[4].
`deref X,Z`	Dereference without preserving a pointer.
`unify_queue X,Y`	Put a pair of values `X` and `Y` into Unification Queue.
`proceed`	Retrieve a pair of values from Unification Queue and performs unification on them. Return if the queue is empty.
`exttype X,T`	extract a type from the value `X` and store to `T`.
`exttag X,T`	extract a t.ag from the value `X` and store to `T`.
`coerce T1,T2,T,FailLabel`	access the type table and set the register `T` as the result of the unified type of `T1` and `T2`. If `T1` and `T2` are incompatible, this instruction transfers the control to the label `FailLabel`.
`coerchk T1,T2,FailLabel`	The same as `coerce` instruction except the result type is not stored to any register.
`jccifa T1,T2,X,Label`	Check the requirement and perform the reallocation of the structure pointed by `X`, which is caused by the change of the type from `T1` to `T2`.
`jcc T1,T2,X,Label`	A variant of the `jccifa` instruction, which does not check the requirement of reallocation and always execute the reallocation routine.
`jec T,X,Label`	Expand the `var`-tagged cell pointed by `X` to the equivalent `str`-represented structure.
`jecifa T1,T2,X,Label`	A variant of `jec T2, X, Label`, which checks if the number of features is changed from the type `T1` to the type `T2`.
`offset T,F,X`	Store the offset of the feature `F` in the structure of type `T` into `X`.

TABLE 5 The Instruction Set in LiLFeS-II (2) Extended Instructions

References

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

Kanayama, H., Torisawa, K., Mitsuisi, Y., & Tsujii, J. ichi. (2000). A hybrid Japanese parse wirh hand-crafted grammar and statitics. In *Proceedings of the 18th International Conference on Computational Linguistics* (pp. 411 – 7). Saarbrücken, Germany.

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.

Krieger, H.-U., & Schäfer, U. (1994). *TDL* — A type description language for constraint-based grammars. In *Proceedings of the 15th International Conference on Computational Linguistics* (pp. 893 – 899). Kyoto, Japan.

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.*

Mitsuishi, Y., Torisawa, K., & Tsujii, J. ichi. (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 – 880). Montreal, Canada.

Miyao, Y., Makino, T., Torisawa, K., & Tsujii, J. ichi. (2000). The LiLFeS abstract machine and its evaluation with the LinGO grammar. *Natural Language Engineering, 6 (1) (Special Issue on Efficient Processing with HPSG)*.

Oepen, S., & Carroll, J. (2000). Performance profiling for parser engineering. *Natural Language Engineering, 6 (1) (Special Issue on Efficient Processing with HPSG)*, 81 – 97.

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

Roy, P. V. (2000). *Issues in implementing constraint logic languages.* http://www.info.ucl.ac.be/people/PVR/impltalk.html.

Tateisi, Y., Torisawa, K., Miyao, Y., & Tsujii, J. ichi. (1998). Translating the XTAG English grammar to HPSG. In *Proceedings of the 4th Workshop on Tree-adjoining Grammars and Related Frameworks (TAG+)* (pp. 172 – 175). Philadelphia, PA.

Torisawa, K., Nishida, K., Miyao, Y., & Tsujii, J. ichi. (2000). An HPSG parser with CFG filtering. *Natural Language Engineering, 6 (1) (Special Issue on Efficient Processing with HPSG)*.

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

Wintner, S. (1997). *An abstract machine for unification grammars with applications to an HPSG grammar for Hebrew.* Unpublished doctoral dissertation, Technion, Israel Institute of Technology, Haifa, Israel.

Yoshida, M., Makino, T., Torisawa, K., & Tsujii, J. ichi. (1998). Optimization techniques for a feature structure processing language LiLFeS. In *Proceedings of the 4th annual meeting of the Association for Natural Language Processing.* Fukuoka city, Japan. ((in Japanese))