Implementation of an Efficient Feature Structure Abstract Machine
高速な素性構造抽象機械の実現

by

MAKINO Takaki
牧野 貴樹

A Senior Thesis
卒業論文

Submitted to
the Department of Information Science
the Faculty of Science
the University of Tokyo
on February 17, 1997
in Partial Fulfillment of the Requirements
for the Degree of Bachelor of Science

Thesis Supervisor: TSUJII Jun'ichi 辻井 潤一
Professor

ABSTRACT

This study focuses on an implementation of a feature structure abstract machine, a framework for handling feature structures efficiently, whose basic notion was proposed by Carpenter et al. The feature structures, which are usually represented as directed graphs or terms in logic programming language, are compiled into an instruction sequence of the abstract machine. High-efficiency is accomplished since an interpretative process of feature structures can be omitted. In addition to the implementation, this study pursues several extensions of the instruction set and the compilation method in the original framework. Comparison with other systems is also shown.

論文要旨

本研究では Carpenter らにより基本的骨格が提案された素性構造抽象機械について述べる。自然言語処理や、人工知能などの分野で使用される素性構造は、従来は Prolog 等の論理型言語で、言語に組み込みの単一化ルーチンを使用して処理されることが一般的であった。素性構造抽象機械では、素性構造をあらかじめ単一化手順として抽象機械の命令列に変換しておくことで、単一化を、抽象機械上の命令実行という効率の高い形で行うことができる。本研究では、実際に実装した素性構造抽象機械に対して命令セットやコンパイル方法の拡張を行い、より高い効率を追求した。また、他の実装系とのパフォーマンスの比較も行った。

Acknowledgements

First, I would like to thank Professor Tsujii Jun'ichi for helpful suggestions and comments. And I would like to thank Researcher Associate Torisawa Kentaro for his informative advice and encouragement. My special thanks are due to Dr. Tateishi Yuka for reading the draft with infinite patience and making a lot of helpful suggestions. I am indebted to Mr. Nishida Kenji for using the LSFSAM in practice and helping me find its concealed bugs. I also wish to express my gratitude to thank all the other members of Tsujii Laboratory, especially Mr. Ninomiya Takashi and Miss Takahashi Ruriko, and all the senior members of this department for encouraging me all the time.

Contents

Chapter 1

Introduction

This study focuses on a Feature Structure Abstract Machine, a framework for handling feature structures efficiently, and its realization.

A feature structure is a sort of data structure, which is able to handle partially unknown information. The notion of feature structures is proposed for representing data structures in Head-driven Phrase Structure Grammar (HPSG) [1], a framework of natural language processing.

A major problem of feature structure processing is its heavy computational costs. For example, one of the fastest parsing system based on HPSG consumes about 3 seconds only to parse a 20-word sentence [2], which is far from practical. The inefficiency of processing is one of the reason that prevents HPSG from being the mainstream in the study of natural language processing.

Recently, a basic notion for an efficient processing of feature structures is proposed by Bob Carpenter and Yan Qu [3]. The framework, Abstract Machine for Attributed-Value Logic (AMAVL hereafter), is based on Warren's Abstract Machine [4], well-known as a high speed framework for Prolog. AMAVL has a potential ability to handle feature structures much faster than conventional implementations, such as ALE [5] or LIFE [6]. However, it is only an idea on paper and no actual implementation exists.

In this work, we have fixed the specification of AMAVL and implemented it. The program is named Light-Speed Feature Structure Abstract Machine (LSFSAM). It consists of C++ source codes of more than 9,000 lines, and includes a number of extensions towards efficiency, such as reusing the result of execution, reducing memory consumption, and reducing memory indexing. It is also able to compile and execute a definitive-clause-based original logical programming language LiLFeS.

The LSFSAM can also be the basis of the system of Artificial Intelligence (AI). The feature-structure-based representation is easily extendable to the frame-based representation found in knowledge representation systems for AI. The new knowledge system which can treat a large amount of data efficiently may be the key to break new ground of AI.

Chapter 2 shows related work. Chapter 3 introduces AMAVL briefly. Chapter

4 describes the features I extended in LSFSAM from AMAVL, including LiLFeS. Chapter 5 shows the result of comparison with other existing implementations. Chapter 6 concludes the paper and describes further research directions.

Chapter 2

Related Work

The origin of the concept of the "abstract machine" for the unification-based language is Warren's Abstract Machine [4]. Currently WAM is known as one of the fastest framework for Prolog.

After the success of WAM, there have been numerous attempts on the basis of WAM to handle feature structures efficiently, such as ALE [5] and ProFIT [7].

ALE is one of the best-known feature structure interpreter. It includes various extended syntax for handling HPSG and lexicons. Because ALE's unification operation relies on Prolog system, the performance of ALE is limited.

On the other hand, ProFIT converts a feature structure into Prolog and relies on the speed of Prolog system. It is quite fast, since there are many fast Prologs system based on WAM, such as SICStus Prolog [8]. However, it is obvious that the realization is not optimal because a feature structure and a term of Prolog are essentially different. That is, a feature structure can have an arbitrary number of features, while a Prolog term has a fixed arity.

There was another work, LIFE [6]. LIFE is a programming language for ψ-terms, which is very similar to feature structures. The difference between ψ-terms and feature structures is that ψ-terms allow that any type can have any feature while feature structures do not. The arbitrariness of features will prevent ψ-term from further optimization.

Chapter 3

Carpenter's Abstract Machine Architecture

This chapter describes how an abstract machine treats feature structures. The first definition of an abstract machine architecture for feature structures is Abstract Machine for Attributed-Value Logics (AMAVL) in [3]. LSFSAM is an extension of AMAVL. I describe AMAVL briefly in this chapter, and describe the extension in LSFSAM in the next chapter.

3.1 Overview of AMAVL

This section presents a brief overview of typed feature structures and AMAVL.

3.1.1 Overview of a Feature Structure

Description of feature structures includes:

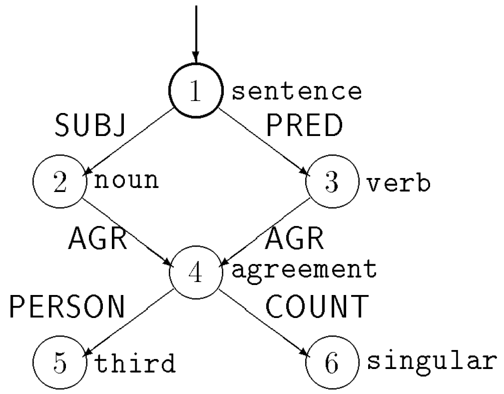

Figure 3.1: Sample Feature Structure in Graph Notation

1. Description of types and features

2. Description of type hierarchy

3. Description of partial information and unification

Types and Features

A *feature structure* is a structure which contains data and their relations. A feature structure is like a directed graph in which every node and every edge are labeled. A node's label identifies the data that the feature structure represents, and is called *type*. An edge's label identifies the part of the parent node's information that the child node has, and is called *feature*. Traditionally, a type is written in lower cases, and a feature is written in upper cases. A sequence of features is called *path*.

In Figure 3.1, the root node's type is `sentence` and there are two edges, labeled

$$\begin{bmatrix} \texttt{sentence} \\ \mathsf{SUBJ}: \begin{bmatrix} \texttt{noun} \\ \mathsf{AGR}: \begin{bmatrix} \boxed{4}\ \texttt{agreement} \\ \mathsf{PERSON}: [\texttt{third}] \\ \mathsf{COUNT}: [\texttt{singular}] \end{bmatrix} \end{bmatrix} \\ \mathsf{PRED}: \begin{bmatrix} \texttt{verb} \\ \mathsf{AGR}: \left[\boxed{4} \right] \end{bmatrix} \end{bmatrix}$$

Figure 3.2: Sample Feature Structure in AVM Notation

SUBJ and PRED, coming from the root node. In feature structure's terms, the value of this feature structure is the type `sentence`, and, the feature structure has two features SUBJ and PRED. The value of the feature SUBJ of this feature structure is the feature structure rooted at the node 2 which has a type `noun`. In short, the value of SUBJ of the feature structure is `noun`. The value of the path ⟨SUBJ AGR⟩ of this feature structure is the same value of AGR of SUBJ, which is the feature structure rooted at the node 4. The value of ⟨SUBJ AGR⟩ and that of ⟨PRED AGR⟩ are the same (they are *structure-shared*), and shows that the agreement of the subject and that of the predicate are equal (in this example, it is 3rd person singular).

The feature structure can be denoted in another form. It is called an *attribute-value matrix (AVM) notation*. Figure 3.2 shows an AVM representation of the feature structure shown in Figure 3.1. The tag $\boxed{4}$ indicates that these two values are structure-shared.

Type Hierarchy

We give a partial-order relation called *subsumption* between types. The subsumption relation $\tau \sqsubseteq \tau'$ (τ subsumes τ') means that the objects represented by the type τ includes the objects represented by the type τ'. In other words, τ is more general than τ', or inversely, τ' is more specific than τ. For example, Figure 3.3 shows a hierarchy of types that represent four-sided figures. Two types connected by a line shows that the upper one is more specific than the lower one. The type `parallelogram` subsumes the type `rectangle` as a reflection of the fact that the rectangle is a special case of parallelogram. On the other hand, `parallelogram` is more specific than `trapezoid`. We apply transitivity, thus `rectangle` is more specific than `trapezoid`. Note that there is no subsumption relation between `parallelogram` and `kite-shape` although they have a common subtype `diamond-shape`.

Adding a type τ to the feature structure F is the same operation as replacing the type of F with the type which is more specific than both the original type and τ. For example, the result of adding `parallelogram` to `kite-shape` is `diamond-shape`.

We always have a universal type $\perp$ (called "bottom"), which is the most general

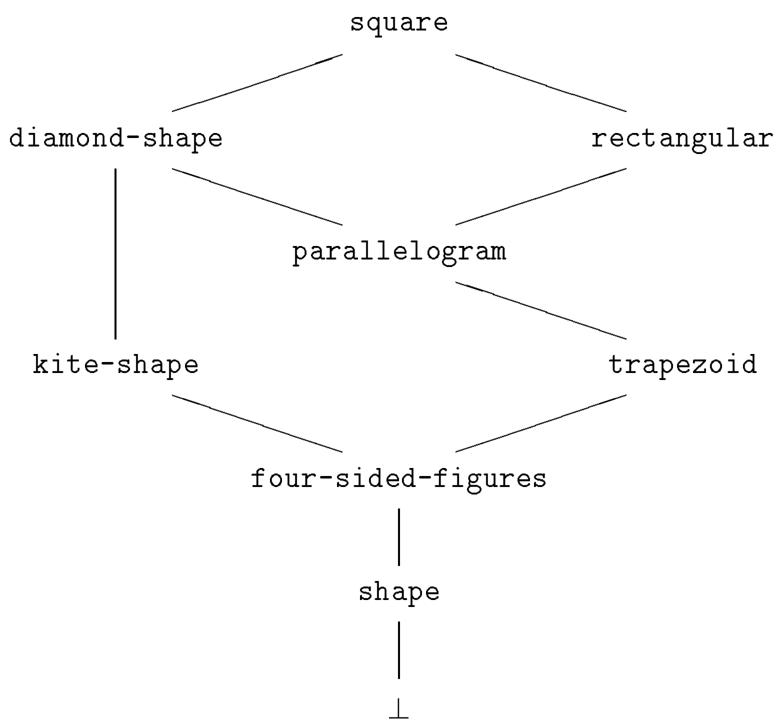

Figure 3.3: Type Hierarchy of Four-sided Objects

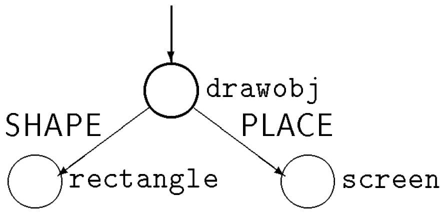

Figure 3.4: Feature Structure Representing "A rectangle is on the screen"

type of all, for mathematical simplicity. The subsumption relation gathers all the types into a lattice; the lattice is called *inheritance hierarchy*.

Unification

A major advantage of the feature-structure representation is an ability to handle partial information. For example, suppose we have three features PLACE, COLOR, and SHAPE, which every feature structure of type `drawobj` has. The type `drawobj` is a representation of a drawing object, and these features represent its place, color and shape, respectively. The information "A rectangle is drawn on the screen" will be represented in Figure 3.4. Note that the absence of feature COLOR expresses that its color is unknown.

When the previously unknown information is acquired, the operation *unification* is performed to make up the absent information. Unification combines two feature structures with partial information and makes a new and more detailed feature structure. If structures have the same feature but different type values, the

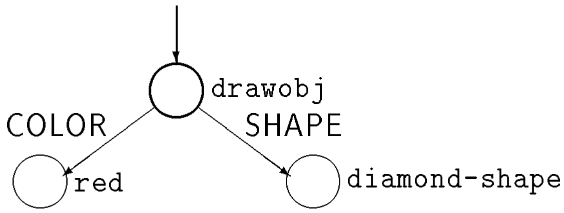

Figure 3.5: Feature Structure "It is also a red diamond-shape"

result becomes the most general type of the types which is subsumed by both type (that is, the value that satisfies both value). If these two values are not consistent, the unification fails.

For example, suppose we got a new information "It is also a red diamond-shape" in succession to the previous information. We then make a new feature structure based on the new information (Figure 3.5) and perform unification to the previous feature structure to integrate these information. The result is as Figure 3.6. Note that the feature COLOR absent from the former feature structure is made up with the feature of the latter feature structure. Note also that SHAPE's value becomes `square`, which is the most general type of the types subsumed by both `rectangle` and `diamond-shape`.

Rules Between Type and Feature

In the examples above, I depicted as if a feature structure value can have any kind of features independently of its type. However, we fix a set of features which are

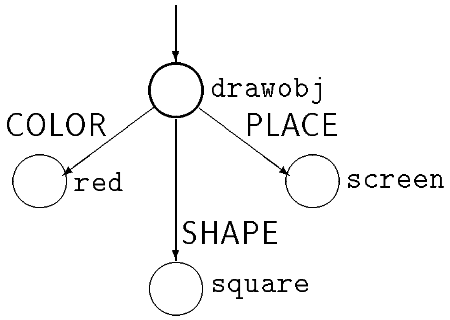

Figure 3.6: Feature Structure Unified

appropriate for a certain type because it is much heavier work to manage an arbitrary number and kind of features. For example, as we defined the type `drawobj` has three features PLACE, SHAPE, and COLOR, any node with the `drawobj` must have these three features and no others. When the value of COLOR is unknown, we store the most general type $\perp$ as the value of the feature instead of making it absent.[1]

To make the feature specification consistent with the type hierarchy, we introduce a *feature appropriateness* rule: If a feature is appropriate for a type τ, then the feature is also appropriate for any type which is more specific than τ. It means that a feature COLOR is appropriate for any type that is more specific than the type `drawobj` because the feature is appropriate for `drawobj`.

[1]We may still let features absent by creating new type. For example, We may create a new type `object-color-unknown`, which is more general than `object` and have PLACE and SHAPE, to handle an object which color is unknown. We may also create as well `object-place-unknown` and `object-shape-unknown`, `object-color-and-place-unknown`, etc.

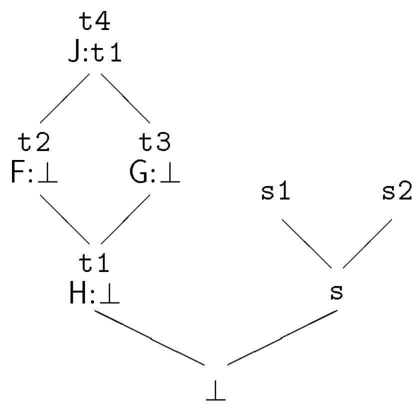

Figure 3.7: An Example Type Hierarchy

We also restrict a type which a certain feature can take as the value. That is, the feature's value can only take a feature structure of the specified type or any type more specific than the specified type. For example, the feature SHAPE may be restricted to a type `shape`. Then SHAPE may have a type `parallelogram` or `triangle`, but it must not have a type `red` or `paper`.

Figure 3.7 is an example of the application of the rule. The notation H:$\perp$ below `t1` means that the feature H is appreciate for the type `t1`, and the feature H can take $\perp$ or its subtype (namely, any type). The features appreciate for `t2` are F and H, because the feature appropriateness rule applies to the feature H. The features appreciate for `t4` are F, G, H, and J.

3.1.2 Feature Structure Abstract Machine

As we have seen, the feature structure is a useful data structure for a complex information. However, handling it is computationally complex. Unification, the main operation for feature structures, requires a lot of processing power for following their features and comparing their types.

From the practical point of view, the AMAVL consists of four techniques to handle feature structures efficiently. The main techniques are:

- Optimizing memory architecture for feature structures
- Doing the routine work used in unifications in advance
- Compiling feature structures for unifications
- Compiling predicates

Let us take a look at these issues.

Optimizing Memory Architecture

We divide memory into "cells," units of memory allocation. A cell consists of a tag and a datum. When a new temporary data area is necessary, we allocate new cells from an addressable global memory storage called *heap*. The allocated heap cells are never freed until the program returns to the user-input level.[2]

Also we fix the number and the sequence of features in a feature structure according to each type. This improves the efficiency of operations on feature structures.

Doing the Routine Work in Advance

Suppose that we are about to unify a feature structure ϕ with m features and another feature structure ϕ' with n features. We must check the duplication between the labels of features of ϕ and those of ϕ' because the unification must be applied to the values of the duplicated features recursively. However, the set of ϕ's features are fixed by the ϕ's type; the same is true on ϕ'. Thus we can prepare the information of feature duplication for any pair of types beforehand, and can improve the performance of execution.

In fact, we can prepare more information for unification because the sequences of features are fixed in the alphabetical order. Eventually the information to be prepared becomes a procedure of unification and we store the procedure as a sequence of instructions for an imaginary device called *abstract machine*.

Procedures of type coercions (converting a structure's type to more specific one

[2] Of course, it is also possible to implement a garbage collection, but it lies outside the scope of this paper.

and rearrange its features) and those of feature lookups (checking the existence of the specified feature and follow up) are also preparable. Thus we prepare sequences of instructions for these procedures as well.

Compiling Feature Structures

In many cases, one of the feature structures to be unified is a *static* structure and never changes during the execution. Instead of analyzing such a feature structure in run time, we can analyze the feature structure prior to the execution and generate a sequence of instructions. Using the "compiled" feature structures, we can improve the performance of unification.

Compiling Predicates

The feature structure is adaptable to logic programming language such as Prolog because both are based on unification. In this area, Warren's Abstract Machine is well-known as a high speed implementation. AMAVL can compile such a predicate into a sequence of instructions as WAM does.

3.2 Feature Structures — Types and Features

This section gives the formal definitions about the types, feature structures, and unification. The description is based on [9].

Assume that we are dealing with a finite set Type of types. It is partially ordered according to their specificity. The expression $\sigma \sqsubseteq \tau$ for types σ, $\tau \in \mathsf{Type}$ means that τ inherits information from σ. In other words, σ *subsumes* or is more *general* than τ (or inversely, τ is subsumed by or more *specific* than σ). If $\sigma \sqsubseteq \tau$, then σ is a *supertype* of τ, or inversely, τ is a *subtype* of σ.

The type $\perp$ is a universal type and subsumes any type. The relation $\langle \mathsf{Type}, \sqsubseteq \rangle$ must be a finite bounded complete partial order. It is called an *inheritance hierarchy.*

Assuming that we are dealing with a finite set Feat of features and an inheritance hierarchy $\langle \mathsf{Type}, \sqsubseteq \rangle$. A feature structure over Type and Feat is a tuple $F = \langle Q, \bar{q}, \theta, \delta \rangle$ where:

- Q : a finite set of nodes rooted at $\bar{q}$
- $\bar{q}$: the root node
- $\theta : Q \rightarrow \mathsf{Type}$: a total node typing function
- $\delta : \mathsf{Feat} \times Q \rightarrow Q$: a partial feature value function

The definition of unification requires some other definitions.

The *subsumption* relation between two feature structures is defined as follows.

$F = \langle Q, \bar{q}, \theta, \delta \rangle$ subsumes $F' = \langle Q', \bar{q}', \theta', \delta' \rangle$, $F \sqsubseteq F'$, if and only if there is a total function $h : Q \rightarrow Q'$, called a morphism, such that:

- $h(\bar{q}) = \bar{q}'$
- $\theta(q) \sqsubseteq \theta'(h(q))$ for every $q \in Q$
- $h(\delta(f, q)) = \delta'(f, h(q))$ for every $q \in Q$ and feature f such that $\delta(f, q)$ is defined

The relation $\sqsubseteq$ is a pre-ordering on a collection of feature structures.

If F and F' are feature structures such that $F \sqsubseteq F'$ and $F' \sqsubseteq F$, the relation is denoted as $F \sim F'$ and called that they are *alphabetic variants*. The relation $\sim$ over the collection of feature structures is an equivalence relation.

Suppose F and F' are feature structures such that $F \sim \langle Q, \bar{q}, \theta, \delta \rangle$ and $F' \sim \langle Q', \bar{q}', \theta', \delta' \rangle$ where $Q \cap Q' = \emptyset$. An equivalence relation $\bowtie$ is defined on $Q \cup Q'$ as the least equivalence relation such that $\bar{q} \bowtie \bar{q}'$ and that $\delta(f, q) \bowtie \delta'(f, q')'$ if both are defined and $q \bowtie q'$.

The unification of F and F' is then defined to be:

$$F \sqcup F' = \langle (Q \cup Q')/_{\bowtie}, [\bar{q}]_{\bowtie}, \theta^{\bowtie}, \delta^{\bowtie} \rangle$$

where:

$$\theta^{\bowtie}([q]_{\bowtie}) = \bigsqcup (\theta \cup \theta')(q') | q' \bowtie q$$

and

$$\delta^{\bowtie}(f, [q]_{\bowtie}) = \begin{cases} [(\delta \cup \delta')(f, q)]_{\bowtie} & \text{if } (\delta \cup \delta')(f, q) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

if all of the joins in the definition of $\theta^{\bowtie}$ exist. $F \sqcup F'$ is undefined otherwise.

In addition to all these definitions, we introduce a *feature appropriateness* rule for type and feature structure interaction. The rule is as follows:

Each type must specify which feature it can be defined for, and which types of value the feature must take. The feature appropriateness is inherited along the type hierarchy in two ways.

- Feature restrictions are inherited. That is, if a feature is appropriate for a type, it must be appropriate for all of the subtypes of the type.

- Type restrictions on feature values are inherited. That is, if a type is an appropriate value for a feature, then all of its subtypes are appropriate values for the feature.

3.3 The Specification of AMAVL

The rest of this chapter describes AMAVL. The operations of the abstract machine are described in this section, and the generation of an instruction sequence is described in the next section.

3.3.1 Memory Areas

The memory in the abstract machine is divided into two major logical areas: static and dynamic. The static memory holds the program and its associated type declarations. This includes the symbol types, type unification and type inference information.

The dynamic memory holds the program execution state. Its primary areas include the following:

- The *heap*, which contains the representations of feature structures.

- The *stack*, which keep track of the local bindings of arguments, variables and structures. A stack provides a natural method for handling nested feature values. The values of variables reside at the bottom of the stack.

  The stack also holds *choice point frames*, which contains information about variables and stack binding, an indicator of where to resume control, how far to unwind the trail, as well as information on recovery points in the heap and the latest choice point.

- The *trail*, which keeps track of information that needs to be restored on backtracking. The trail records pairs of overwritten addresses and their previous values.

Register	Function
HTOP	Top of the heap
STOP	Top of the stack
P	Instruction Pointer (program counter)
CB	Continuation code counter
HB	Heap pointer indicating the range which is necessary to trail
TR	Trail pointer
$x_1, x_2, \ldots, x_n$	Registers for argument passing and temporary storage

Table 3.1: Execution state

- The *push down list*, another stack structure which is used to store sequences of feature structures remaining to be unified for the unification algorithm.

3.3.2 Execution State

Table 3.1 lists the information stored to represent a given execution state of the machine.

3.3.3 The Instruction Set

Table 3.2 contains the instruction set for the abstract machine with a brief description of what each instruction does. The registers and arguments following some instructions are left out.[3]

[3]For details of these instructions, see [10].

Instructions for type coercion code	
BIND	Bind a new feature cell to an old feature cell
FRESH	Create a new feature cell
Instructions for procedural control	
ALLOCATE	Allocate a new environment frame on the stack
DEALLOCATE	Deallocate an environment frame on the stack
JUMP	Jump to labeled clause and continue execution
Instructions for dynamic unification code	
SKIP1	Skip one heap cell for feature structure one
SKIP2	Skip one heap cell for feature structure two
UNIFY	Unify two heap cells
COPY1	Copy the heap cell of feature structure one
COPY2	Copy the heap cell of feature structure two
Choice instructions	
TRY	Allocate a new choice point
RETRY	Unwind heap and try next alternative (branch)
LASTTRY	Unwind heap and try last alternative (tail recursive)
Instructions for compiled unification code	
ADDNEW	Add a new type onto the heap
PUSH	Push the feature value to the stack
POP	Pop a value off stack
UNIFYVAR	Update the binding of variables

Table 3.2: Abstract Machine Instructions

3.3.4 Memory Architecture

A feature structure with n features is represented using $n+1$ contiguous cells on the heap. The first cell contains the type of the structure's root node with a STR tag, while the other n cells have PTR tags and each of them contains a pointer of the structure's feature value in the alphabetical order. Remember that a type has a fixed set of features; we can determine which feature's data a cell contains without including feature name information in data structure.

In addition to these tags, We introduce another tag VAR. VAR cell represents a feature structure as a cell sequence beginning with STR do, but it uses only one cell and include no information about its features. The VAR structure can be placed at the STR structure's feature value instead of redirecting by a pointer and put a structure in another place (See Figure 3.8). When nothing is known about structure's feature values, we can leave them as variables, thus saving space and time.

For example, suppose we have a type hierarchy and feature appropriateness as in Figure 3.7. Figure 3.10 shows a heap representation for the feature structure of Figure 3.9 starts at heap address 11. Note that the feature `F`'s value is represented at heap address 12 as a PTR to a STR cell at address 14 because the feature value for feature `G` of type `t3` is explicitly specified. Unspecified information on the type

(a) Without VAR

Address	Cell
2	STR `t2`
3	PTR 5
4	PTR 8
5	STR `t1`
6	PTR 7
7	STR ⊥
8	STR ⊥

(b) With VAR

Address	Cell
2	STR `t2`
3	VAR `t1`
4	VAR ⊥

Figure 3.8: Change of Heap Representation by the introduction of VAR tag

$$\begin{bmatrix} \texttt{t2} \\ \mathsf{F}\text{:} \begin{bmatrix} \texttt{t3} \\ \mathsf{G}\text{:}[\texttt{t3}] \end{bmatrix} \\ \mathsf{H}\text{:}[\texttt{t4}] \end{bmatrix}$$

Figure 3.9: An Example Feature Structure

appropriateness of feature H of type `t3` is automatically filled up at address 16 with the default type ⊥, which is specified as the type restriction for the feature H. The feature H's value of the main structure is represented as a VAR cell at address 13 rather than a STR cell, as no explicit feature information is specified for the structure of type `t4`.

Address	Cell
11	STR `t2`
12	PTR 14
13	VAR `t4`
14	STR `t3`
15	VAR `t3`
16	VAR $\perp$

Figure 3.10: An Example Heap Representation

3.4 Compiling with the Abstract Machine

This section describes how the typed feature structure logic can be compiled and executed using the abstract machine. The compilation of type definitions is described first and then the compilation of feature structures is described.

3.4.1 Compiling Type Definitions

Carpenter's abstract machine requires strict type definitions prior to the program. We can prepare a large amount of procedures from the type definitions. The preparable procedures can be divided into three sorts:

Type Coercion Procedures Procedures to add a type on a feature structure. The procedures are stored in a table indexed by the specified type and the feature structure's type.

Feature Lookup Procedures Procedures to follow a feature's value at the feature structure of a type, including any necessary type coercion. The procedures are stored in a table indexed by the feature and the type.

Unification Instructions Procedures to determine whether a pair of types are consistent, and if so, what the value type of the result is and how the features match up for further unification and type inference. The procedures are stored in a table indexed by the pair of types.

All procedures might also indicate failure.

In the next subsection, employment of compiled instructions from type definitions and generation of the instructions are described.

3.4.2 Generation and Application of Compiled Type Definitions

Type Coercions

A type name in a feature structure is compiled into a type coercion instruction. When executed, it adds the information associated with the type to a structure located at a specific heap address.

After dereferencing the structure by following PTR values, the result may be either a VAR cell or a STR cell. When the result is a VAR cell, we can just look up the result of coerced type in a table and update the type of the variable cell. If the type on the instruction and the type of the VAR cell are inconsistent, the instruction fails and backtrack occurs. When the result is STR cell, assuming the type being added is τ and the type of the feature structure being coerced is τ', there are three possibilities.

- When τ and τ' are inconsistent, the instruction fails and triggers backtrack.
- If the new type τ is subsumed by the type τ' of the structure in the heap, nothing needs to be done and the command merely succeeds.
- When τ and τ' are consistent but the result type is not τ', we need to create a new structure on the heap for the result of the type coercion.

For example, suppose type `t2` is to be added to heap address 14 in Figure 3.11(a). From the type hierarchy in Figure 3.7, we know that `t2` and `t3` unify to a new type `t4`. This is the third case of the list above, and a new structure is created on the heap for the result. The old structure must be redirected to the new one by means of a PTR cell replacing the old value at 14. Then the precompiled instructions for adding `t3` to a structure of `t2` is called. The sequence of the instructions and its effect is on Figure 3.11 (c), and the result of execution

Addr	Cell
14	STR `t2`
15	VAR `t2`
16	VAR `bot`

Addr	Cell
14	PTR 17
15	VAR `t2`
16	VAR `bot`
17	STR `t4`
18	VAR `bot`
19	PTR 15
20	PTR 16
21	VAR `t1`

(a) Heap before add (b) Heap after add

FRESH	`bot`	(Creates new feature F for `t4`)
BIND		(Binds to previous value for feature G)
BIND		(Binds to previous value for feature H)
FRESH	`t1`	(Creates new feature J for `t4`)

(c) Instruction sequence

Figure 3.11: Sample of type coercion

is on Figure 3.11 (b). Because the features are in alphabetic order, BIND and FRESH are all the instructions needed for type coercion.

Feature lookups

The instruction that look into a feature of the structure invokes a feature lookup. Feature lookup follows the feature structure to its feature's value. If the specified feature does not exist on the structure, then the instruction tries to create the feature on the structure.

After dereferencing, the cell for the structure is either a VAR cell or an STR cell. If it is a VAR cell, it is necessary to expand the structure into STR representation with its appropriate features with variable values of the appropriate type. Next, if the feature is absent from the structure, the structure is coerced to one that is appropriate for the feature by adding the most general type appropriate for the feature, which may result in failure and cause backtracking. Finally, we push the value of the feature that is now guaranteed to exist onto the top of the stack and recursively descend the structure. When the descending are finished, we can pop the value from the stack.

The compiled instruction for feature lookups covers performing type coercion (if necessary) and the index of the feature cell relative to the STR cell of the structure.

Dynamic Unification

The instruction that performs a structure sharing invokes a dynamic unification (that is, the unification that both of the feature structures varies in run-time). Note that it is the only instruction that call the dynamic unification operation. All other structural manipulations are carried out by the other primitive instructions.

Instructions for unifying two feature structures at two different heap addresses are stored in a table, which can be represented as a (sparse) two dimensional array. There are a number of cases depending on the types of the structures, and whether the structures are variables or not.

- If the two structures are on the same heap pointer after dereferencing, the unification simply succeeds.
- If one of the structures is a variable, it is updated as a pointer to the other structure, and its type is added to the other structure.
- If both structures are STR structures:
  - If one of the structures has a type subsuming the other, the more general structure is made to point to the more specific structure, and the features that need to be unified are read out of the table entry for the two types. These are indicated in terms of UNIFY and SKIP instructions, indicating whether the feature in the more specific structure should be unified with the next feature in the more general structure or skipped.
  - If the two structures unify to a new type of structure, a new structure must be allocated of the appropriate type, and both structures are made to point to it. Then the new values are constructed as either pointers to a value in one of the existing structures or by creating a

fresh variable cell. If both structures have a feature, these value must be pushed onto the unification stack so that their values are eventually unified. The instruction used is COPY1/COPY2 (the value is simply pointing to the value in the first or second structure), UNIFY (the value is determined by unifying the next value in both structures), or FRESH (a new variable structure should be created).

As usual, any failure causes backtracking.

3.4.3 Compiling a Feature Structure

A feature structure can be decomposed into four basic elements: types, features, structure sharing, and conjunctions. AMAVL can handle disjunctions in addition to these. For each element, we describe how the element is compiled into a sequence of instructions for unification.

Types

When an element of a type τ is in the compiled feature structure, the compiled instructions should indicate that the type of the unifying feature structure should be changed into the type subsumed by τ in execution time. The instruction is ADDNEW, which adds a type to the feature structure, and may call the precompiled instructions.

Features

When an element of a feature F is in the compiled feature structure, the compiled instruction should indicate that the feature of the unifying feature structure should be followed and substructures of both compiled and unifying feature structure should be unified.

If the substructure of the compiling feature structure has a feature element, features are followed recursively. Thus it is necessary to save the address of unifying feature structure before the unification of substructures, and recover the address after the unification completes. For that purpose we use a stack and treat the top of the stack as the current feature structure pointer. We use PUSH instructions to follow the feature, and use POP instructions to recover the pointer value after the substructures are unified.

Structure Sharing

When a tag indicating structure sharing is in the compiling feature structure, we should remember the feature structure at the first occurrence of the tag and perform a unification at the later occurrences of the tag. For example, suppose we are compiling the feature structure shown in Figure 3.12. Figure 3.13 shows the instruction codes that generates from the feature structure.

$$\begin{bmatrix} \texttt{t2} & \\ \text{F:} & \begin{bmatrix} \boxed{1}\ \texttt{t3} \\ \text{G:}[\texttt{t3}] \end{bmatrix} \\ \text{H:} & \begin{bmatrix} \boxed{1} \end{bmatrix} \end{bmatrix}$$

Figure 3.12: Structure Sharing: Traditional Notation

```
ADDNEW    t2
PUSH      F
UNIFYVAR  1
PUSH      G
ADDNEW    t3
POP
POP
PUSH      H
UNIFYVAR  1
POP
```

Figure 3.13: Structure Sharing: Sample Instruction

In the abstract machine, a space is allocated at the bottom of the stack for all of the local variables. At the first occurrence of a tag, the variable correspond to the tag is bound to the current structure. Any later occurrence of the same tag will simply invoke unification for the current structure with the value of the variable. Both of the feature structures used in the unification are dynamic; the dynamic unification procedure described in 3.4.2 is used. Note that the structure sharing is the only element that invokes a dynamic unification.

$$\begin{bmatrix} \texttt{t2} \\ \mathsf{F}\text{:}\left(\begin{bmatrix} \texttt{t1} \end{bmatrix} \vee \begin{bmatrix} \texttt{t2} \\ \mathsf{G}\text{:}[\texttt{t1}] \end{bmatrix} \vee \begin{bmatrix} \texttt{t3} \\ \mathsf{H}\text{:}[\texttt{t1}] \end{bmatrix} \right) \end{bmatrix}$$

(a) Disjunctive Notation

$$\begin{bmatrix} \texttt{t2} \\ \mathsf{F}\text{:}\begin{bmatrix} \texttt{t1} \end{bmatrix} \end{bmatrix} \vee \begin{bmatrix} \texttt{t2} \\ \mathsf{F}\text{:}\begin{bmatrix} \texttt{t2} \\ \mathsf{G}\text{:}[\texttt{t1}] \end{bmatrix} \end{bmatrix} \vee \begin{bmatrix} \texttt{t2} \\ \mathsf{F}\text{:}\begin{bmatrix} \texttt{t3} \\ \mathsf{H}\text{:}[\texttt{t1}] \end{bmatrix} \end{bmatrix}$$

(b) Normal Notation; (a) is equal to one of them

Figure 3.14: Disjunctions in Feature Structure

Conjunctions

A conjunction means two or more elements exist in parallel in a feature structure. To represent a conjunction in the level of instructions, we should simply concatenate two or more sequences of instructions for each element.

Disjunctions

A disjunction means two or more elements exist at the same place on a feature structure and indicates that the feature structure satisfies one of these elements (See Figure 3.14).

In the abstract machine, disjunctions are treated by means of backtracking. The direction of execution is from left to right, and depth-first. When encountering a failure or a call for additional solutions, execution backtracks to the last considered choice point, restores information about variable bindings, and continues execution.

Disjunction is executed as follows: When a disjunct is evaluated at the first time, it may create side effects on the stack and the heap by updating the variable bindings on the stack and changing the feature structure values of the heap. These effects must be undone when considering an alternative. The data area TRAIL is used to keep track of all the heap cells that have been updated for the chosen disjunct and that need to be restored for another disjunct. TRAIL is an array of pairs of cells, and records the heap address about to be overwritten and its current value.

Only conditional bindings, those affecting a variable existing before creation of the current choice point, need to be trailed. In this case, a conditional binding is the one affecting the content of the heap cell before creation of the current choice point. To determine the condition, a global register HB is used to contain the value of the heap-top pointer at the time of the latest choice point.

As WAM does, a choice point frame is used to save all the necessary states for restoring and continuing execution upon backtracking. Every choice point frame is initially pushed on the top of the choice point stack. The following information needs to be stored in a choice point frame:

- The continuation pointer: The next instruction pointer to try upon backtracking.
- The current trail pointer: It is the boundary where to unwind the trail upon attempting an alternative.
- The current top of the heap: The boundary of recording trails. Heap cells after this point are created by the current choice point and need not to be trailed.
- Variable bindings: $x_1, \ldots, x_n$, where n is the number of variables in this description.

There are four instructions related to compiling disjunctions: TRY, RETRY, LASTTRY and JUMP. TRY allocates a choice point frame on the stack, and then continues execution of the following instructions. RETRY resets all necessary information after backtracking, allocates a new choice point, and then continues execution of the following instructions. LASTTRY is like RETRY, except that it indicates that variables need no longer be trailed because there are no more alternatives for backtracking. JUMP directs execution to the end of disjunction block.

Chapter 4

Extension of the Abstract Machine

This chapter describes what is actually implemented in LSFSAM. LSFSAM is a fixed implementation of AMAVL, which is only a basic notion, and has been extended at many points from AMAVL in order to improve efficiency and handle Definitive clauses. The full details on LSFSAM implementation is found in Appendix A.

4.1 Fixed Specifications in LSFSAM

In [3], the followings are not mentioned.

- Method for initialization of variables. Without initialization of variables, it is unable to determine whether the variables are used at the first time or not.

- Avoidance of collision between choice point frames and the operations of PUSH/POP instructions. They may produce side effects hampering correct execution.

I fixed these points so that LSFSAM can work properly and efficiently. Details on them are described below.

4.1.1 Initialization of Variables

It is important to determine whether a variable appears for the first time in a (Definitive) clause, because when it appears for the first time, we have to initialize the variable's value. In Prolog the first appearance can be judged at the compilation time and the results can be reflected in the compiled instructions. However, as Carpenter pointed out in [3], we cannot determine at the compilation time whether a variable appears for the first time in AMAVL because we allow disjunctions in a feature structure. Carpenter's solution is quite simple: for each time a variable is referred, AMAVL examines whether the variable is already used. Although it is clear that initialization of variables are necessary for doing this, there is no description in [3] about how to initialize variables.

My new scheme for the initialization of variables is as follows:

- A new register RP, which indicates the number of local variable in use, is introduced.

- A new instruction CLRTEMP is introduced. This instruction takes one argument n for the number of local variable registers. When executed, it clears the registers $x_0, x_1, \ldots, x_{n-1}$ and sets RP to n.

- The CALL instruction and EXECUTE instruction are modified to set RP to the arity of the predicate.

- The ALLOCATE instruction now requires two arguments n_p and n_t, where n_p is the number of permanent variables used in the clause, and n_t is the sum of the arity of the clause and the number of temporary variables used at the beginning of the clause. In addition to making an environment frame of size p, the instruction clears the local-variable registers $x_{\mathrm{RP}}, x_{\mathrm{RP}+1}, \ldots, x_{t-1}$, and sets RP to t. Note that RP contains the arity of the predicate when this instruction is executed, so that $x_0, x_1, \ldots, x_{\mathrm{RP}-1}$ contain the arguments for the clause and should not be cleared.

In addition to these, the TRY instruction is changed to use RP to determine the number of local variables, which is necessary for creating a choice-point frame.

We also need to initialize the *current heap pointer*, which is on the top of the stack. ADDNEW, PUSH, and UNIFYVAR refer to the current heap pointer. The current heap pointer should be initialized at:

- The beginning of a predicate, and
- The beginning of a new description

To perform the first case, the ALLOCATE instruction now pushes one extra empty cell on the stack. For the second case, the START instruction that clears the current heap pointer is introduced.

4.1.2 Changes in Handling of PUSH/POP Instructions

In AMAVL, the PUSH instruction pushes a data onto the stack. It causes some collision with choice point frames because they share the same stack. Figure 4.1 (a) shows the compiled instruction code from the feature structure Figure 4.1 (b). When the code is executed, a choice point frame is created on the top of the stack at (*2). When the execution reaches the instruction POP at (*3), a data of the choice point pushed at (*2) is popped although it should pop the datum pushed at (*1). Even if this difficulty is avoided, for example, by creating a new stack area for only choice points, the PUSH instruction at (*4) overwrites the previous stack data, which have to revert to the state at (*2) at the time of backtracking.

To resolve this difficulty, I changed the PUSH/POP instructions so that, instead of accessing the stack data area, they regard local variable registers as a stack

```
          ADDNEW    t2
          PUSH      F
          ADDNEW    t1
          PUSH      H         (*1)
          UNIFYVAR  3
          TRY       try2      (*2)
          ADDNEW    s1
          JUMP      tryend
try2:     LASTTRY
          ADDNEW    s2
tryend:   POP                 (*3)
          POP
          PUSH      G
          ADDNEW    t1
          PUSH      H         (*4)
          UNIFYVAR  3
          POP
          POP
```

(a) Compiled feature structure

$$\begin{bmatrix} \texttt{t2} & \\ \mathsf{F}: & \begin{bmatrix} \texttt{t1} \\ \mathsf{H}: \left[\boxed{3} \; (\texttt{s1} \vee \texttt{s2}) \right] \end{bmatrix} \\ \mathsf{G}: & \begin{bmatrix} \texttt{t1} \\ \mathsf{H}: \left[\boxed{3} \right] \end{bmatrix} \end{bmatrix}$$

(b) The feature structure before compiling

Figure 4.1: Collision of PUSH and Choice Point Frame

and push onto/pop from the local variable register area. The change of the section 4.1.1 supplies the new register RP which indicates the upper bound of the range in use, and can serve as a stack pointer on the stack variable register. Thus we can treat the local-variable registers as a new stack and the PUSH/POP instructions act upon them. This method can handle PUSH and POP correctly across choice point creation/destruction, since all of the local variable registers currently in use are stored in a choice point frame when it is created, and reverted safely when backtracking occurs.

4.2 Extension for Efficiency

In this section, LSFSAM's extended mechanisms for improving efficiency are described.

4.2.1 Heap Attention Pointer

The *current heap pointer* is stored on the top of the stack in AMAVL. Most instructions into which feature structures are compiled work on the current heap pointer. We can avoid unnecessary indexing by not storing the current heap pointer on the stack but in a fixed register. The newly created register is called heap attention pointer (HA).

Creation of the HA affects some instructions.

	Register value of HA	$\uparrow$
	Temporary variable $RP-2$	$\vdots$
	$\vdots$	(RP cells)
	Temporary variable 1	$\vdots$
	Temporary variable 0	$\downarrow$
	The value of RP (the number of temporary variables)	
	The previous value of HB (Heap boundary of trailing)	
	The value of TP (Trail pointer)	
	The value of FP (Environment frame address)	
	The previous value of CP (Continuation pointer)	
	The code pointer to next TRY block address	
CP$\rightarrow$	The previous choice point frame	

Figure 4.2: New Structure of a Choice Point Frame

- The structure of a choice point frame has to be changed. The HA acts like the last local variable register, thus:
  - If RP is 0, neither HA nor local variables are written to the choice point frame.
  - Otherwise, (RP−1) local variables and the HA are written to the choice point frame.

  The new structure of a choice point frame is illustrated in Figure 4.2.

- PUSH instruction now pushes the content of the HA register, and revises the HA to the address of the specified feature's value. The POP instruction overwrites the HA with the popped data.

- The ALLOCATE instruction no longer allocates an extra cell on the top of the register-stack, but clears the HA.
- The START instruction is also changed to clear the HA.
- The UNIFYVAR instruction needs not be changed to adapt to the addition of the HA because the UNIFYVAR instruction never operates on the register accessed by PUSH/POP instructions.

4.2.2 Registers and Instructions for Type Coercion/Dynamic Unification

Type coercion instructions and dynamic unification instructions are treated as "microcodes," and must hold all the other machine states unchanged. These instructions do not call any other sequences of instructions, thus the registers used in these instructions can be fixed and independent from the registers used in the other instructions.

For this reason, I introduce three registers and one instruction.

- UIP (Unification Instruction Pointer) is a register to hold the return address when the type coercion or dynamic unification procedure finishes.

- HU1 and HU2 (Heap Unification Pointer 1/2) are registers to hold the addresses of the feature structures being unified in dynamic unification instructions. HU1 also holds the address of the feature structure being coerced in type coercion instructions.

- The PROCEED instruction is placed at the end of type coercion and dynamic unification instructions, and:

  - If the unification push down list is not empty, gets a pair of addresses from the push down list and sets these values to HU1 and HU2, then calls the dynamic unification routine.

  - Otherwise, sets IP to the value of UIP (type coercion/unification is complete).

Another set of instructions are added for in order to embed more information to sequences of instructions and reduce run-time conditional branching, and to avoid unnecessary heap allocation.

- The ADDSTR instruction is used at the top of type coercion instructions, and indicates that the new structure of a type is the type specified by the argument.

- The UNIFY_S instruction is used instead of UNIFY and indicates that the unified structure is also bound to the newly allocated cell.

- BINDxxx series of instructions are used at the top of type coercion instructions, and indicate that the unified type of the structures and how it is allocated. In the following, the unification now performing are between a feature structure S_1 whose type is τ_1 and a feature structure S_2 whose type is τ_2, and the result type is τ. $F(\tau)$ is the set of features appropriate for the type τ.

  - BINDEO (each other) is used when $\tau_1 = \tau_2$. In this case the result may be bound to the memory of either S_1 or S_2.

  - BINDEOT (each other, typed) is used when $\tau_1 \not\sqsubseteq \tau_2$ and $\tau_2 \not\sqsubseteq \tau_1$, but $F(\tau_1) = F(\tau)$ and $F(\tau_2) = F(\tau)$. This is almost same as BINDEO except that it write the result type τ to the memory.

  - BIND1 (to 1) is used when $\tau_1 \sqsubseteq \tau_2$. In this case the result will be bound to S_1.

  - BIND1T (to 1, typed) is used when $\tau_1 \not\sqsubseteq \tau_2$ but $F(\tau_1) = F(\tau)$. In this case the result can be bound to S_1 but the type should be overwritten.

  - BIND2 and BIND2T are same as BIND1 and BIND1T except that the direction of subsumption is reversed.

– BINDNEW (new) is used when none of the above conditions is satisfied. The result is bound to the newly allocated heap cells.

4.2.3 Copy of Feature Structures

In natural language processing, most parsing algorithms reduce the parsing costs by keeping intermediate results and reusing it several times. Feature structure unification in an AMAVL framework is a destructive operation, so that copying of feature structures is necessary to keep and reuse the intermediate results.

To copy feature structures, we introduce a new data area BACKHEAP, which is used to store the copied feature structures. BACKHEAP and HEAP are assigned to the same address space but different address so that the cell with PTR tag can point to either of them.

A copying algorithm of feature structure is not trivial since structure sharing must be copied. The key concept of my algorithm is (1) overwrite the copying structure with the pointer of copied structure, saving the original data into TRAIL, and (2) when the copy is completed, restore the overwritten data by TRAIL information.

The copying algorithm is on Figure 4.3. Function CopyStructure is a self-

Function Copy(heapaddr p) : heapaddr;
  While p points to HEAP and HEAP[p] is a PTR cell,
    Repeat p := (address of pointer at HEAP[p]) .
  If p is a pointer to BACKHEAP area,
    No need to copy, return p as the address of copied structure.
  Or, If HEAP[p] is a VAR cell,
    Allocate $n+1$ cells from BACKHEAP (address p'),
    BACKHEAP[p'] := HEAP[p],
    Return p' as the address of copied structure.
  Otherwise (the case that HEAP[q] is a STR cell)
    Set a choice point.
    p' := CopyStructure(p).
    Revert the state of HEAP to the choice point
End of function.

Function CopyStructure(heapaddr p) : heapaddr;
  τ := (the type of feature structure at p) .
  n := (the number of features of τ) .
  Allocate $n+1$ cells from BACKHEAP. p' := (the allocated memory address) .
  Put the tuple (p, HEAP[p]) to TRAIL.
  BACKHEAP[p'] := HEAP[p]. HEAP[p] := $\langle$PTR, $p'\rangle$.
  For i from 1 to n, repeat following
    $q := p + i$.
    While HEAP[q] is a pointer to HEAP area,
      Repeat q := (address of pointer at HEAP[q]) .
    If HEAP[q] is a pointer to BACKHEAP area,
      BACKHEAP[$p' + i$] := HEAP[q].
    Or, if HEAP[q] is a VAR cell,
      Put the tuple (p, HEAP[p]) to TRAIL, and
      BACKHEAP[$p' + i$] := HEAP[q], HEAP[q] := $\langle$PTR, $p' + i\rangle$.
    Otherwise (the case that HEAP[q] is a STR cell)
      BACKHEAP[$p' + i$] := $\langle$PTR, CopyStructure(q)$\rangle$.
  Return p' as the address of copied structure.
End of function.

Figure 4.3: Feature Structure Copying Algorithm

recursive function that copies feature structures to BACKHEAP, while Function Copy is a shell function that calls CopyStructure and restores HEAP after the copy is complete.

The copied feature structure is *normalized*; the feature structure is put on a certain area of address and the structure is ordered in top-down depth-first order. Therefore, equivalence check between normalized feature structure can be performed by linear memory comparison rather than traversing the two feature structure. For the details of the technique, See [11].

4.3 LiLFeS

LSFSAM can compile and run a logic program based on Definitive clauses. To express the program with feature structures neatly, I have designed a new programming language LiLFeS (stands for Light-speed Language for Feature Structure) and integrated a LiLFeS parser with LSFSAM.

To enable LSFSAM to handle LiLFeS programs, I also designed a register allocation algorithm and backtracking among clauses.

4.3.1 Syntax of LiLFeS

LiLFeS's syntax is basically a mixture of Prolog and ALE [5]. See Appendix B for full details of the syntax in BNF.

Notation of identifiers

In LiLFeS, a type's name must begin with a lowercase letter, while a variable's name must begin with an uppercase letter or an underscore(_). Note that a feature's label and a predicate's name may begin with either an uppercase or lowercase letter, because they are recognized in another way.

A variable whose name begins with an underscore(_) is called an *anonymous* variable and means that no unification is necessary.

Type Definition

The syntax of a command for defining a type is as follows:

newtype `<-` *supertype1* [`,` *supertype2* ...] [`+` *feature*[`:` *restrict-type*] `,` ...] `.`

An operator `<-` means "is subsumed by" and is used to define a type. When we define a type, we specify one or more supertype of it separated by commas, and zero or more features newly introduced into it. Introduced features may have a type restriction.

For example, Figure 4.4 shows the definition of a type hierarchy shown in Figure 3.7. `bot` is the built-in type indicating $\perp$. The definition of `t4` shows that the type has two supertypes, `t2` and `t3`, and its newly introduced feature `J` is restricted to `t1`.

```
s <- bot.
s1 <- s.
s2 <- s.
t1 <- bot + H.
t2 <- t1 + F.
t3 <- t1 + G.
t4 <- t2, t3 + J:t1.
```

Figure 4.4: Sample Type Definition

Representation of Feature Structure

A feature structure corresponds to a nonterminal ⟨structure⟩ in the Appendix B. Figure 4.5 shows an example of ⟨structure⟩.

Definitive clause

The notation of Definitive clauses is the same as Horn clauses of Prolog. It is as follows:

predicate `(` [⟨structure⟩ [`,` ⟨structure⟩ ...]] `)` [`:-` ⟨goal⟩ [`,` ⟨goal⟩ ...]]

where ⟨goal⟩ is one of the followings:

$$\begin{cases} \textit{predicate}\ \texttt{(}\ [\langle\text{structure}\rangle\ [\texttt{,}\ \langle\text{structure}\rangle \ldots]]\ \texttt{)} \\ \langle\text{structure}\rangle\ \texttt{=}\ \langle\text{structure}\rangle \end{cases}$$

The goal is either a call of another predicate, or the form *structure*`=`*structure* that performs unification.

$$\begin{bmatrix} \texttt{person} & \\ \text{NAME:} & \begin{bmatrix}\texttt{name} & \\ \text{FIRSTNAME:} & [\texttt{"Phillip"}] \\ \text{LASTNAME:} & [\,\boxed{3}\ \texttt{"Dick"}\,]\end{bmatrix} \\ \text{WIFE:} & [\texttt{nobody}] \vee \begin{bmatrix}\texttt{person} & \\ \text{NAME:} & \begin{bmatrix}\texttt{name} & \\ \text{LASTNAME:} & [\,\boxed{3}\,]\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

(a) Sample Feature Structure

```
(person, NAME:(name, FIRSTNAME:"Phillip", LASTNAME:(X, "Dick")),
 WIFE:(nobody; (person, NAME:(name, LASTNAME:X))))
```

(b) A ⟨structure⟩ Representing the Feature Structure (a)

Figure 4.5: Sample Program Piece Representing a Feature Structure

```
list <- bot.
nil <- list.
cons <- list + hd, tl:list.
```

Figure 4.6: Definition of Built-In List Type

Lists

We prepared some abbreviations for linear lists. It is similar to the list representation in Prolog. Figure 4.6 shows the built-in types for representing lists. Figure 4.7 shows abbreviations for lists.

For example, the predicate `append` that appends two lists can be represented as follows:

Abbreviation form	Non-abbreviation form
`[]`	`nil`
`[A]`	`(hd:A, tl:nil)`
`[A|B]`	`(hd:A, tl:B)`
`[A,B]`	`(hd:A, tl:(hd:B, tl:nil) )`
`[A,B|C]`	`(hd:A, tl:(hd:B, tl:C) )`

Figure 4.7: Abbreviation Form for List Type

```
append([], X, X).
append([A|X], Y, [A|Z]) :- append(X, Y, Z).
```

Queries

The syntax for a query is as follows:

?- ⟨goal⟩ [, ⟨goal⟩ ...] .

where ⟨goal⟩ is the same as that of Definitive clauses. Following is an example of queries, the result of which is to append the list `[s1,s2]` to the list `[t1,t2,t3]`:

```
?- append([t1,t2,t3], [s1,s2], X).
```

4.3.2 Compiling LiLFeS

To execute LiLFeS on LSFSAM, we need to compile LiLFeS clauses into a sequence of instructions for LSFSAM. During compilation, local variable registers are used for structure sharing and argument passing. I have established a register managing system for LSFSAM based on that of WAM.

Use of Registers

We use the local variable registers to pass arguments to a predicate. Variable 1 holds the first argument, variable 2 holds the second argument, and so on. The local variable registers that are not used as argument passing can be used for local variables.

Although LSFSAM has an infinite number of registers for local variables, they are destroyed each time a predicate is called. Thus we provide *permanent variables* which are variables living over one or more predicate calls. We still use local variable registers for *temporary variables* for the variables which are not necessary to be saved. We allocate permanent variables in the environment frame on the stack as auto variables of C language do, and allocate temporary variables on registers

For example, suppose we are about to compile a following definitive clause.

```
p(X,Y) :- q(X,Z), r(Y,Z).
```

The variables `Y` and `Z` are permanent because their value must hold over the predicate call of `q`. `X` is not permanent because the value is not necessary after the call. Therefore `X` is allocated on a temporary variable register, and `Y` and `Z` are allocated on the stack.

	Permanent variable n
	$\vdots$
	Permanent variable 2
	Permanent variable 1
	The number of permanent variables
FP→	The previous environment frame

Figure 4.8: Structure of an Environment Frame

A new internal register FP is introduced to store the current environment frame. The structure of an environment frame is shown in Figure 4.8. In addition to these, the UNIFYVAR instruction is extended to distinguish permanent variables from temporary variables.

Figure 4.9 shows a sample of compiled codes.

Backtracking

We have already seen the method for handling backtracks at the description about the compilation of disjunctions. The same framework is usable to backtracks on predicates.

However, here we introduce a new set of instructions for backtracks on predicates. The reason is not only its easiness to implement, but also that these instructions are effective on indexing optimization [4], which avoids unnecessary unification for partially instantiated predicate calls.

```
ALLOC       2, 3     | p(
START                |
UNIFYVAR    TEMP-0   |
UNIFYVAR    TEMP-2   |   X,
START                |
UNIFYVAR    TEMP-1   |
UNIFYVAR    PERM-2   |     Y) :-
CLRTEMP     2        |
UNIFYVAR    TEMP-2   |           q(X,
UNIFYVAR    TEMP-0   |
START                |
UNIFYVAR    PERM-1   |               Z
UNIFYVAR    TEMP-1   |
CALL        q/2      |                ),
CLRTEMP     2        |
UNIFYVAR    PERM-2   |                    r(Y
UNIFYVAR    TEMP-0   |
CALL        r/2      |                       ),
CLRTEMP     1        |
UNIFYVAR    PERM-1   |                          s(Z
UNIFYVAR    TEMP-0   |
CALL        s/1      |                             )
DEALLOC              |
RETURN               |                              .
```

Figure 4.9: Compiled Code of `p(X,Y) :- q(X,Z), r(Y,Z), s(Y).`

```
     TRY l1
     (First disjunction block)
     JUMP l3
l1:  RETRY l2
     (Second disjunction block)
     JUMP l3
l2:  LASTTRY
     (Last disjunction block)
l3:  RETURN (or other instruction sequence)
```

(a) Original instructions

```
     TRYAT l1
     RETRYAT l2
     LAST l3
l1:  (First disjunction block)
     RETURN
l2:  (Second disjunction block)
     RETURN
l3:  (Last disjunction block)
     RETURN
```

(b) New instructions

Figure 4.10: Difference Between Original and New Backtracking Instructions

The newly introduced instructions are listed in Table 4.1. The differences of these instructions from the original backtracking instructions are:

- The instruction in the original set first continues its execution at its follower, and when the backtracking occurs, jumps back to the specified instruction address.

- The instruction in the new set first jumps to the specified instruction address, and when the backtracking occurs, continues its execution at its follower.

The difference is illustrated on Figure 4.10. The original instructions are still used on a disjunction in a feature structure.

Instruction	Argument	Description
TRYAT	L	Allocate a new choice point frame setting its next clause field to the following instruction and jump to instruction labeled L.
RETRYAT	L	Having backtracked to the current choice point, reset all the necessary information from it, update its next clause field to the following instruction, and jump to the instruction labeled L.
LASTAT	L	Having backtracked to the current choice point, reset all the necessary information from it, and discard it, and jump to the instruction labeled L.

Table 4.1: Newly Introduced Instructions for Backtracking

Chapter 5

Comparison With Other Implementations

In this chapter we compare LSFSAM with 4 other implementations. We prepare 3 sorts of experiments to measure various aspects of these implementations.

5.1 Paddock

In addition to LSFSAM, we measure the speed on ALE [5], ProFIT [7], Wild LIFE [12], and SICStus [8].

ALE is one of the major implementations of typed feature logic by Bob Carpenter and Gerald Penn. It runs on SICStus Prolog and has various extended syntax for handling HPSG and lexicons.

ProFIT is another implementation of typed feature logic developed by Gregor Erbach. Its strategy is to compile feature structure term into Prolog terms.

ProFIT also runs on SICStus Prolog.

LIFE [6] is a programming language for ψ-term, which is very similar to feature structures. However, LIFE does not have a feature appropriateness rule, so any type can have any feature. LIFE and its major interpreter Wild LIFE are developed by Hassan Aït-Kaci et al.

SICStus is not a feature structure based system but a pure Prolog compiler. Its high-efficiency and high-reliability have pushed it up to one of the most widely used Prolog systems. SICStus cannot handle feature structures although it is based on the same WAM architecture as the LSFSAM is. Therefore, although comparing with SICStus is not fair, the comparison will show the target performance accomplishable from the architecture.

5.2 Experiments

We performed three types of experiments:

- **Append test**: This program measures the efficiency of the handling of simple feature structures.

- **Incremental Unification (IU) test**: This program measures the efficiency of the one-by-one (static) unification.

```
p1 <- bot.
p2 <- bot.
p3 <- bot.

append([], X, X).
append([A|X], Y, [A|Z]) :- append(X, Y, Z).

atest([], []).
atest([A|X], Y) :- atest(X, Y1),
                   append(Y1, [A], Y2),
                   append(Y2, Y1, Y).

test1 :-
    atest([p1,p2,p3,p1,p2,p3,p1,p2,p3,p1,p2,p3,p1,p2,p3,p1], Z).
```

Figure 5.1: The LiLFeS Program of Append Test

- **Dynamic Unification (DU) test**: This program measures the efficiency of the one-time (dynamic) unification.

5.2.1 Append Test

Append test is a benchmark program that generates a long list. Figure 5.1 shows the program code of the test in LiLFeS. Simply to say, the test performs the following:

$$\mathrm{atest}(l) = \begin{cases} \text{an empty list} & \text{if } l \text{ is an empty list} \\ \text{appended list } \langle \mathrm{atest}(\mathrm{cdr}\ l), \mathrm{car}\ l, \mathrm{atest}(\mathrm{cdr}\ l) \rangle & \text{otherwise} \end{cases}$$

The predicate `atest` is a predicate which gets a list of length N and returns a list of length $2^N - 1$. In this experiment we used 16-length list as an input of `atest`, so the length of the result is 65535. The predicate `append` is a self-recursive predicate, and the number of calls is the same as the length of its first argument. The number of total calls for `append` in this experiment is 131070.

A Prolog system can parse the LiLFeS program without change, since this experiment does not rely on features.

5.2.2 Incremental Unification (IU) Test

IU test is a benchmark program that measures the handling of somewhat complex feature structures and incremental unifications (that is, the unification which is performed by static unification instructions in LSFSAM). Figure 5.2 shows the program code of the test in LiLFeS.

The predicate `utest` is a predicate which generate a Fibonucci-like complex tree. In this experiment we provide 22-length list as an input of `utest`, so the number of nodes in the result is Fib(23) = 28657. The second call for `utest` in `test2` specifies the same variable as the first call, thus the necessary unification is performed one by one (in another words, incrementally).

Since this test is based on feature structure framework, some changes are required to perform the test in a pure Prolog system. In this case we map a feature structure to a functor, whose arguments are alphabetically-ordered features of all the features appearing in the original program. Figure 5.3 shows the Prolog ver-

```
w <- bot + A, F, K.
  w1 <- w  + C, G.
  w2 <- w  + D, H.
  w3 <- w  + E, J.
    w12 <- w1, w2.
    w13 <- w1, w3.
    w23 <- w2, w3.
      wx <- w12, w13, w23.

p1 <- bot.
p2 <- bot.
p3 <- bot.

utest([], w).
utest([p1], w1).
utest([p2], w2).
utest([p3], w3).
utest([p1,A|X], (C:Y, G:Z, K:Y, A:Z))
        :- utest([A|X], Y), utest(X, Z).
utest([p2,A|X], (D:Y, H:Z, K:Y, A:Z))
        :- utest([A|X], Y), utest(X, Z).
utest([p3,A|X], (E:Y, J:Z, K:Y, A:Z))
        :- utest([A|X], Y), utest(X, Z).

test2 :-
  utest([p1,p2,p3,p1,p2,p3,p1,p2,p3,p1,p2,p3,p1,p2,p3,
             p1,p2,p3,p1,p2,p3,p1], X),
  utest([p3,p3,p1,p1,p2,p2,p3,p3,p1,p1,p2,p2,p3,p3,p1,
             p1,p2,p2,p3,p3,p1,p1], X).
```

Figure 5.2: The LiLFeS Program of IU Test

```
utest([],       wx(_,_,_,_,_,_,_,_).
utest([p1],     wx(_,_,_,_,_,_,_,_)).
utest([p2],     wx(_,_,_,_,_,_,_,_)).
utest([p3],     wx(_,_,_,_,_,_,_,_)).
utest([p1,A|X], wx(Z,Y,_,_,_,Z,_,_,Y))
                              :- test([A|X], Y), test(X, Z).
utest([p2,A|X], wx(Z,_,Y,_,_,_,Z,_,Y))
                              :- test([A|X], Y), test(X, Z).
utest([p3,A|X], wx(Z,_,_,Y,_,_,_,Z,Y))
                              :- test([A|X], Y), test(X, Z).
```

Figure 5.3: The Prolog Program Equivalent to IU Test

```
test3 :-
  utest([p1,p2,p3,p1,p2,p3,p1,p2,p3,p1,p2,p3,p1,p2,p3,
           p1,p2,p3,p1,p2,p3,p1], X),
  utest([p3,p3,p1,p1,p2,p2,p3,p3,p1,p1,p2,p2,p3,p3,p1,
           p1,p2,p2,p3,p3,p1,p1], Y).
  X = Y.
```

Figure 5.4: The LiLFeS Program of DU Test

sion of the IU test. Each argument of `wx` shows the feature A, C, D, E, F, G, J, and K, respectively, which were appeared on Figure 5.2.

5.2.3 Dynamic Unification (DU) Test

DU test is almost the same as IU test, but it performs dynamic unification between feature structures. Figure 5.4 shows the entrance predicate of the test in LiLFeS.

While IU test creates one structure and performs unification incrementally, DU test creates two separate structures and performs unification at once. As a result, DU test reflects the time of dynamic unification.

Test\System	LSFSAM	ALE	ProFIT	Wild LIFE	SICStus
Append	1.55	28.9	0.20	14.1	0.21
IU	2.13	36.6	2.48	78.0 Out of memory	3.06
DU	3.22	36.7	6.18	Not tested	6.69

(CPU time given in seconds)

Table 5.1: Result of Speed Comparison Experiments

5.3 Results and Discussion

The experiment is performed on DEC Alpha 600/333MHz with 256M RAM. The result of the experiments are shown in Table 5.1. The DU test on Wild LIFE are not performed because they even failed in IU test, which is less heavy than DU test.

Generally LSFSAM runs faster than the other implementations. Especially in IU test and DU test, LSFSAM is even faster than SICStus. The reason is probably that Prolog is unable to handle feature structures and is forced to keep the information "this feature is empty" all the time, thus much processing is required. The feature structures in practical application such as HPSG are much larger than those in this experiment and have a far more empty features, and consequently, the difference will be larger.

In Append test, ProFIT and SICStus are much faster than LSFSAM. One of the possible reason is that SICStus is highly optimized for list handling and predicate calling, and ProFIT is able to convert the program to Prolog and make SICStus work efficiently. Further optimizations will bring LSFSAM closer to the efficiency level of SICStus.

Chapter 6

Conclusion and Future Work

In this paper we have described a feature structure abstract machine, named LSFSAM, which is an extension of the framework AMAVL proposed by Carpenter et al. [3]. The characteristics of LSFSAM are summarized as follows:

- LSFSAM is the first realization of AMAVL framework to the best of our knowledge. To realize AMAVL some specifications had to be fixed.

- LSFSAM have a function of feature structure copying, which is not mentioned in AMAVL. It is useful for natural language processing because the same result is used in many places in such a processing.

- LSFSAM is able to handle Prolog-like language LiLFeS in addition to AMAVL framework. With LiLFeS we easily and efficiently handle the feature structures on LSFSAM.

The comparison of LSFSAM with the existent implementations shows that LSFSAM is quite competitive.

Currently planned future work are as follows:

1. Further optimization

   Many powerful optimization techniques have been proposed in Warren's Abstract Machine [4]. Integrating them into LSFSAM may improve its performance drastically. The possible techniques are:

   POP reduction optimization Carpenter et al. [3] mentioned the optimization to eliminate unneeded POPs.

   Instruction join optimization Joining a frequently occurring sequence of instructions into one instruction may reduce calculation time.

   Last call optimization The last predicate call need not return back to the current predicate.

   Register initializing optimization The variable initialization method of WAM is usable when no disjunction exists.

   Register bindings optimization Some excess register bindings can be trimmed.

Environment trimming optimization The environment frame may contain some useless information in current implementation.

Stack variables optimization Some heap cells consumed for temporary redirecting can be saved by moving it to the stack.

Indexing optimization When some of the arguments are partially instantiated, we can avoid some unifications which can be determined as certain failure.

C-conversion optimization To convert well-used predicates, type coercion routines, and unification instructions into C language will improve the efficiency.

2. **Further extension for natural language processing**

   One of the main application of LSFSAM is natural language processing. Some extensions are essential to use LSFSAM in natural language processing framework such as HPSG. For example, we can implement the equivalence checking and unifiability checking functions in LSFSAM.

3. **Parallelization of the LSFSAM**

   The design of LSFSAM is intended for parallelization. All machine states are organized into one C++ object and can easily copy and send to another processor. We expect parallelized LSFSAM to work much faster and widen its application areas.

4. **Integration of probability-based search**

   For the purpose of finding the result as fast as possible, the most possible predicate should be called first. Building probability-based search into LSFSAM should bring it to the basis of development of Artificial Intelligence systems because the requirement of such a system is the ability to handle complex information efficiently. LSFSAM with probability-based search should be qualified for the requirement.

Bibliography

[1] Carl Pollard and Ivan A. Sag. *Head-Driven Phrase Structure Grammar*. University of Chicago Press and CSLI Publications, 1993.

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

[3] Bob Carpenter and Yan Qu. An abstract machine for attribute-value logics. In *IWPT 95*, pages 59–70, 1995.

[4] Hassan Aït-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction*. The MIT Press, 1991. ISBN 0-262-01123-9 (hc.) — ISBN 0-262-51058-8 (pbk.).

[5] Bob Carpenter and Gerald Penn. ALE 2.0 users' guide - the attribute-logic engine. Technical report, Computational Linguistics, Carnegie Mellon University, 1994. Also available on http://macduff.andrew.cmu.edu/ale/.

[6] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16(3&4):195–234, 1993.

[7] Gregor Erbach. ProFIT 1.54 user's guide. Available in http://coli.uni-sb.de/ erbach/formal/profit/profit.html, 1995.

[8] Swedish Institute of Computer Science. *SICStus Prolog User's Manual, Release 3 #0*, June 1995. Also available on http://www.sics.se/sicstus.html.

[9] Bob Carpenter. *The Logic of Typed Feature Structures.* Cambrige University Press, 1992. ISBN 0-521-41932-8.

[10] Yan Qu. An abstract machine for typed feature structure theories. Master's thesis, Carnegie Mellon University, 1994.

[11] Kenji Nishida. Implementation of partial unification using feature structure abstract machine. A Senior Thesis, 1997. Submitted to the Department of Information Science, the Faculty of Science, the University of Tokyo.

[12] Hassan Aït-Kaci et al. The Wild LIFE handbook (prepublication edition). Available in http://www.isg.sfu.ca/life/, March 1994.

Appendix A

LSFSAM Implementation

A.1 Memory Architecture

Following is the list of the sorts of memory areas used in LSFSAM.

Memory	Description
HEAP	The global memory area. All of the data of the feature structures are put on here.
BACKHEAP	The memory area that stores copied feature structures.
STACK	Environment frames, which holds a part of local variables, and choice point frames, which holds backtracking information of a machine state, are on the stack.
TRAIL	The trail holds backtracking information of the heap. The pairs of overwritten heap addresses and their previous values.
PDL	A dynamic unification may include one or more dynamic unification between substructures. The push down list holds the pairs of heap addressed to be unified.

A.2 Register Set

Following is the register set in LSFSAM. Some of them have different names from those of AMAVL.

Register	Description
HP	The heap pointer holds the top of the heap.
HB	The heap boundary pointer holds the boundary of the range on the heap where trailing are needed.
HA	The heap attention pointer indicates where the static unification code are pointed on.
HU1	The heap unification pointer 1 indicates the place the type coercion code is focusing on, or the dynamic unification code is focusing on as one of the two feature structures.
HU2	The heap unification pointer 2 indicates the place the dynamic unification code is focusing on as the other of the two feature structure.
SP	The stack pointer holds the top of the stack.
FP	The frame pointer holds the address of the current environment frame.
RP	The register pointer holds the top of the LOCAL registers.
TP	The trail pointer holds the top of the trail.
CP	The choice-point-frame pointer holds the address of the current choice point frame.
IP	The instruction pointer holds the next instruction address to execute.
CIP	The continuation instruction pointer holds the next instruction address when RETURN instruction is executed at the end of the clause.
UIP	The unification instruction pointer holds the next instruction address when PROCEED instruction is executed at the end of the type coercion/dynamic unification code.

A.3 Instruction Set

Following is an instruction set of LSFSAM.

Instruction	Arg	Description
Instructions used in dynamic unification code		
BIND1		Overwrite the cell at HU2 with $\langle$PTR, HU1$\rangle$. Increment HU1 and HU2.
BIND2		Overwrite the cell at HU1 with $\langle$PTR, HU2$\rangle$. Increment HU1 and HU2.
BINDEO		Performs either BIND1 or BIND2.
BINDEOT		Update the cell at HU2 with the common subtype of feature structures at HU1 and HU2, and overwrite the cell at HU2 with $\langle$PTR, HU1$\rangle$ to. Increment HU1 and HU2.
BIND1T	τ	Overwrite the cell at HU2 with $\langle$PTR, HU1$\rangle$, and set $\langle$STR, $\tau\rangle$ to the cell at HU1. Increment HU1 and HU2.
BIND2T	τ	Overwrite the cell at HU1 with $\langle$PTR, HU1$\rangle$, and set $\langle$STR, $\tau\rangle$ to the cell at HU2. Increment HU1 and HU2.
BINDNEW	τ	Allocate a new cell from heap (Address A) and set to it $\langle$STR, $\tau\rangle$. Overwrite the cells at HU1 and HU2 with $\langle$PTR, $A\rangle$. Increment HU1 and HU2.
UNIFY		Put the pair (HU1, HU2) to the unification queue. Increment HU1 and HU2.
UNIFYS		Put the pair (HU1, HU2) to the unification queue. Allocate a new cell from heap and set to it $\langle$PTR, HU1$\rangle$. Increment HU1 and HU2.
COPY1		Allocate a new cell from heap and set to it $\langle$PTR, HU1$\rangle$. Increment HU1.
COPY2		Allocate a new cell from heap and set to it $\langle$PTR, HU2$\rangle$. Increment HU2.
SKIP1		Increment HU1.
SKIP2		Increment HU2.
FRESH	τ	Allocate a new cell from heap and set to it $\langle$VAR, $\tau\rangle$.
PROCEED		If the unification queue is empty, continue execution to UIP. Otherwise, get a pair of address from the queue and jump to the unification routine.

Instructions used in static unification code		
ADDNEW	τ	Add a new type τ to the feature structure at HA.
PUSH	F	Save the current HA to the LOCAL[RP], increment RP, and follows the feature F of the feature structure at HA. Try to create the feature if not exist.
POP		Decrement RP and restore HA from the LOCAL[RP].
UNIFYVAR	n	Start a dynamic unification code between HA and LOCAL[n].
Backtracking Instructions		
TRY	L	Allocate a new choice point frame setting its next clause field to the instruction labeled L and continue execution from the following instruction.
RETRY	L	Having backtracked to the current choice point, reset all the necessary information from it, update its next clause field to the instruction labeled L, and continue execution from the following instruction.
LASTTRY		Having backtracked to the current choice point, reset all the necessary information from it, and discard it, and continue execution from the following instruction.
JUMP	L	Continue execution from the instruction labeled L.
TRYAT	L	Allocate a new choice point frame setting its next clause field to the following instruction and jump to instruction labeled L.
RETRYAT	L	Having backtracked to the current choice point, reset all the necessary information from it, update its next clause field to the following instruction, and jump to the instruction labeled L.
LASTAT	L	Having backtracked to the current choice point, reset all the necessary information from it, and discard it, and jump to the instruction labeled L.
Instructions used in type coercion code		
BIND		Allocate a new cell from heap and store to it $\langle$PTR, HU$\rangle$, and increment HU1 by 1.
FRESH		*See the same instruction above*
ADDSTR	τ	Allocate a new cell from heap and set to it $\langle$STR, $\tau\rangle$.
PROCEED		*See the same instruction above*

Instructions for variable/clause control		
CLRTEMP	n	Clear LOCAL[0], LOCAL[1] $\cdots$ LOCAL[$n-1$].
START		Clear HA.
ALLOC	n_p, n_t	Allocate an environment frame with n_p permanent variables and clear LOCAL[RP], LOCAL[RP+1], $\cdots$ LOCAL[$n_t - 1$].
DEALLOC		Deallocate the current environment frame.
CALL	p	Set the following instruction address to CIP, and jump to the address of predicate p.
EXECUTE	p	Jump to the address of predicate p.
Miscellaneous Instructions		
FAIL		Always fail and trigger backtrack if possible.

Appendix B

The Syntax of LiLFeS

Following is a BNF description of the syntax of LiLFeS.

```
           ⟨top⟩ ::= ⟨typedef⟩
                   | ⟨sentence⟩

       ⟨typedef⟩ ::= ⟨typename⟩ <- ⟨supertypelist⟩ .
                 ::= ⟨typename⟩ <- ⟨supertypelist⟩ + ⟨introfeatlist⟩ .

 ⟨supertypelist⟩ ::= ⟨typename⟩
                   | ⟨typename⟩ , ⟨supertypelist⟩

 ⟨introfeatlist⟩ ::= ⟨introfeat⟩
                   | ⟨introfeat⟩ , ⟨introfeatlist⟩

     ⟨introfeat⟩ ::= ⟨featurename⟩
                   | ⟨featurename⟩ :  ⟨typename⟩

      ⟨sentence⟩ ::= ⟨proc⟩ .
                   | ⟨proc⟩ :- ⟨goallist⟩ .
                   | ?- ⟨goallist⟩ .

          ⟨proc⟩ ::= ⟨procname⟩
                   | ⟨procname⟩ ( ⟨arglist⟩ )
```

$$
\begin{array}{rcl}
\langle\text{goallist}\rangle & ::= & \langle\text{goallist}\rangle \;\; ; \;\; \langle\text{goallist}\rangle \\
& | & \langle\text{goallist}\rangle \;\; \texttt{,} \;\; \langle\text{goallist}\rangle \\
& | & \texttt{(} \;\; \langle\text{goallist}\rangle \;\; \texttt{)} \\
& | & \langle\text{goal}\rangle \\
\\
\langle\text{goal}\rangle & ::= & \langle\text{structure}\rangle \;\; \texttt{=} \;\; \langle\text{structure}\rangle \\
& | & \langle\text{proc}\rangle \\
\\
\langle\text{arglist}\rangle & ::= & \langle\text{structure}\rangle \\
& | & \langle\text{structure}\rangle \;\; \texttt{,} \;\; \langle\text{arglist}\rangle \\
\\
\langle\text{structure}\rangle & ::= & \texttt{(} \;\; \langle\text{structure}\rangle \;\; \texttt{)} \\
& | & \langle\text{typename}\rangle \\
& | & \langle\text{varname}\rangle \\
& | & \langle\text{featurename}\rangle \;\; : \;\; \langle\text{structure}\rangle \\
& | & \langle\text{structure}\rangle \;\; \texttt{,} \;\; \langle\text{structure}\rangle \\
& | & \langle\text{structure}\rangle \;\; ; \;\; \langle\text{structure}\rangle \\
& | & \texttt{[} \;\; \langle\text{list}\rangle \;\; \texttt{]} \\
\\
\langle\text{list}\rangle & ::= & \qquad\qquad\qquad\qquad\qquad\qquad ; \; (\mathit{empty}) \\
& | & \langle\text{liststructure}\rangle \\
& | & \langle\text{liststructure}\rangle \;\; \texttt{|} \;\; \langle\text{structure}\rangle \\
\\
\langle\text{liststructure}\rangle & ::= & \langle\text{structure}\rangle \\
& | & \langle\text{liststructure}\rangle \;\; \texttt{,} \;\; \langle\text{structure}\rangle
\end{array}
$$