A NATIVE-CODE COMPILER FOR A UNIFICATION-BASED PROGRAMMING LANGUAGE WITH TYPED FEATURE STRUCTURES

型付き素性構造を使用する単一化ベースのプログラミング言語のためのネイティブコードコンパイラ

by

MAKINO Takaki

牧野 貴樹

A Master Thesis

修士論文

Submitted to

the Department of Information Science

the Faculty of Science

the University of Tokyo

on February 10, 1999

in Partial Fulfillment of the Requirements
for the Degree of Master of Science

Thesis Supervisor: TSUJII Jun'ichi　**辻井 潤一**
Professor of Information Science

ABSTRACT

This study discusses design and implementation of a native-code compiler for LiLFeS, a unification-based programming language with typed feature structures (TFSs). Although LiLFeS has been designed as a platform for natural language processing of TFS-based formalisms based on an Abstract Machine for Attribute-Value Logics (AMAVL) proposed by Carpenter et al., people require more efficiency in order to implement advanced natural language processing applications, such as for statistical learning from large corpora. A large number of studies have been made on efficient implementations of Prolog, such as for the native-code compiler Aquarius Prolog, and those studies are partially applicable to the LiLFeS implementation. However, in order to increase efficiency of TFS unification, optimization using type information is necessary and the compiler should directly generate unification code, including type manipulation. A simple extension of current Prolog or AMAVL implementations cannot achieve that. This thesis describes a design for a LiLFeS compiler, which is an extension of the Berkeley Abstract Machine, the implementation framework of Aquarius Prolog. The thesis focuses on an implementation of TFS unification on an abstract machine, including (1) construction and analysis of unification algorithm for direct generation of TFS unification code, (2) design of abstract machine instructions for expressing the algorithm described in (1), and (3) static analysis techniques for type hierarchies and definite clause programs for providing context information used in code optimization. Performance comparisons of the LiLFeS compiler to other LiLFeS and Prolog systems show the high efficiency of the implementation detailed in this thesis.

論文要旨

本論文は、型付き素性構造 (TFS) を使用する単一化ベースのプログラミング言語 LiLFeS のためのネイティブコードコンパイラの設計と実装について述べる。素性構造を使用するフォーマリズム上で自然言語処理を行うための基幹言語として、 Carpenter らの提案する属性値論理抽象機械 (AMAVL) を元に設計された LiLFeS であるが、大量のコーパスからの統計学習などを含めた高度な自然言語処理を実現するには処理能力の点で不十分であった。LiLFeS の実装に応用ができる高効率な Prolog の実装方式としては、 Prolog のネイティブコードコンパイラ Aquarius Prolog などさまざまな研究が行われてきている。しかし、素性構造の単一化効率を高めるには、型情報を利用した最適化を実現することが必要で、そのためには型操作を含む単一化コードをコンパイラが直接出力しなければならない。これは、既存の Prolog 実装の単純な拡張や、 AMAVL との統合だけでは実現することができなかった。本論文では、 Aquarius Prolog の実装方式である Berkeley Abstract Machine を拡張した、 LiLFeS ネイティブコードコンパイラの設計について述べる。具体的には、抽象機械上での型付き素性構造の単一化の実現手法に関して、 (1) 型操作を含むコードを出力に含めるための単一化アルゴリズムの構築と分析、 (2) それらのアルゴリズムを記述するための抽象機械命令の設計、 (3) その上でコード最適化のための文脈情報を提供する型階層および確定節プログラムの静的解析技術、の 3 点の提案について詳述する。また、他の LiLFeS および Prolog 処理系との性能比較によって、本論文で提案する実装が高効率であることを示す。

Acknowledgements

First, I would like to thank Professor Tsujii Jun'ichi for his helpful suggestions and comments. And I would like to thank Research Associate Torisawa Kentaro for his informative advice and encouragement. This work could not be completed without contribution of Mr. Yoshida Minoru, Mr. Sese Jun, Mr. Watanabe Hiroshi, Mr. Ueno Kazunori, Mr. Sugihara Tomomichi and Mr. Ishibashi Taketo. I am indebted to Dr. Tateishi Yuka, Mr. Mitsuisi Yutaka, Mr. Miyao Yusuke, Mr. Kanayama Hiroshi, and Mr. Kimura Tomohiko for using the LiLFeS in practice and helping me find its concealed bugs. My special thanks are due to Mr. Ninomiya Takashi for reading the draft with infinite patience and making a lot of helpful suggestions. I am thankful to Dr. Taura Kenjiro for his generous advice and comments. I also wish to express my gratitude to thank all the other members of Tsujii laboratory, especially Ms. Uozumi Akiyo, Ms. Sumi Naoko and Ms. Tanaka Yoriko, and all the graduates of this department in the same year for encouraging me all the time.

Contents

1 Introduction **5**

1.1 Existing Systems for Typed Feature Structures (TFSs) 6

1.2 Approach to an Efficient TFS System 7

1.3 Structure of This Paper 8

2 Related Work **9**

2.1 Prolog Systems 9

2.2 TFS Systems 10

2.3 TFS Abstract Machines 10

2.4 LiLFeS 12

3 LiLFeS, a Programming Language **14**

3.1 Overview of a Typed Feature Structure 14

3.1.1 Types and Features 14

3.1.2 Type Hierarchy 16

3.1.3 Unification 17

3.1.4 Appropriateness Condition 19

3.2 LiLFeS, a programming language for TFSs 20

4 Architecture of the LiLFeS Native-code Compiler **23**

4.1 Overall Process Flow 23

4.2 NEW-LiLFeS Abstract Machine 24
4.2.1 Data Areas and Registers 25
4.2.2 Representation of TFS 27
4.2.3 Instruction set 29
4.3 Compilation Steps 29

5 Unification Algorithm **34**
5.1 Overview of Unification Code Generation 34
5.2 Compilation of Types – General Unification Code 35
5.2.1 Algorithm for General Unification 37
5.2.2 Type Compilation for General Unification 39
5.3 Compilation of Terms – Specialized Unifier Code 40
5.3.1 TFS without substructure 41
5.3.2 TFS without substructure, special type 42
5.3.3 TFS with substructure 42
5.3.4 Instances of integer, string, and float 44

6 Code Generation **45**
6.1 Extensions to the Instruction Set 45
6.1.1 Additions for Type-table Manipulation 45
6.1.2 Additions for Unification-Queue Operation 46
6.1.3 Modifications for Adapting Changed Memory Architecture . 46
6.2 Construction of the Code 48
6.2.1 TFS without substructure 48
6.2.2 TFS without substructure, special type 49
6.2.3 TFS with substructures 50

7 Optimization **52**
7.1 Optimization by Type Hierarchy Analysis 52
7.1.1 Optimization Conditions 53

7.1.2 Optimizing by Conditions 55
7.2 Optimization by Global Data-Flow Analysis 59

8 Built-in Predicates **61**
8.1 Useful Predicate for Efficient Parser 61
8.2 TFS Normalization 63

9 Evaluation **65**
9.1 Comparison to Other Prolog Systems 66
9.2 Comparison to Other TFS-based Systems 66
9.3 Optimization 68
9.4 Discussion 69

10 Conclusion and Future Work **70**

Chapter 1

Introduction

This thesis discusses a native-code compiler for LiLFeS (NEW-LiLFeS), a logic programming language with TFSs, in order to provide a basis for practical Natural Language Processing (NLP) applications based on HPSG[1]. The design of a TFS unification compiler and optimizer is the main topic of the development of NEW-LiLFeS. The basic principle of optimization of TFS unification is to reduce the amount of redundant operations in the unification code.

The final goal of this thesis is to provide an efficient platform for HPSG-based applications. HPSG is a sophisticated grammar formalism that uses Typed Feature Structures (TFSs), which are defined in a mathematically sound way. The HPSG formalism has the advantages of both linguistically justifiable grammar formalisms such as GB and computational grammar formalisms such as CFG. That is, it features high expressive power for natural language description and extensibility with statistical methods. Currently, an HPSG-based Japanese grammar shows both high precision and wide coverage among discrete grammar formalisms [2], and further improvement of that system using statistical method is under research [3]. Nevertheless, HPSG was widely believed as *being too slow for practical use* and there have been no serious attempts at implementation. Our claim is that HPSG itself may actually not be so inefficient; it is just that an efficient implementation of

HPSG has not been seriously pursued till now. We set a goal for the performance of a *practical* HPSG system: 100 milliseconds of average parsing time per sentence in real-world corpora. If our HPSG parser accomplishes this goal, it would be capable to parse about 1,000,000 sentences in a day, and could be used for applications such as knowledge acquisition from corpora.

1.1 Existing Systems for Typed Feature Structures (TFSs)

Since Typed Feature Structures (TFSs) [4] are the basic data structure in HPSG, the efficiency of handling TFSs has been considered the key to improving the efficiency of HPSG parsers. There exist some systems that handle TFSs, such as ALE [5], a TFS interpreter written in Prolog, and ProFIT [6], a TFS-to-Prolog-term compiler. However, as the comparison of these systems with our system (Chapter 9) shows, none of the existing systems can achieve the required efficiency we established as our goal in our research.

My Senior Thesis described my earlier LiLFeS system based on emulation of an abstract machine (OLD-LiLFeS) [7] as a basis for HPSG-based systems. OLD-LiLFeS provides efficient TFS unification embedded in a Prolog-like programming language, and produces an efficient, and easy-to-use environment for HPSG systems. English and Japanese HPSG grammars (XHPSG [8] and SLUNG [2]), an HPSG-based parser based on an advanced parsing algorithm (TNT-Parser [9]), a disjunction packing algorithm [10], and a parallel programming environment for TFSs (PSTFS [11]) have all been developed on the OLD-LiLFeS system. However, it is still impossible to achieve our goal of 100ms parsing time per a sentence using OLD-LiLFeS alone on a single processor. A more efficient TFS system is therefore required.

1.2 Approach to an Efficient TFS System

Unification of TFSs is an important operation in parsing based on HPSG. In order to make this operation as efficient as possible, we developed a native code compiler in addition to a sophisticated unification optimizer. Optimization on TFS unification is done by reducing redundant operations in the unification code. Depending on context, certain parts of the operations combined in the unification code may be redundant. For example, if the type of one of the TFSs to be unified has no subtypes, the result of unification can only be this same type again if unification succeeds. General unification code includes a table access to determine the unified type; such operation is redundant in this context.

However, such optimization is impossible in the design of the existing LiLFeS system (OLD-LiLFeS). The reasons are that (1) since OLD-LiLFeS compiles a program incrementally, such context information is hard to provide, and (2) even if the context information is available, the instruction set is too coarse-grained to remove redundant operations from the unification code.

Thus, in NEW-LiLFeS, we redesigned the TFS unification compiler from scratch, paying attention to the following points:

Global data-flow analysis and type hierarchy analysis on whole programs. Global data-flow analysis and type hierarchy analysis provides much information that is useful for redundant code removal. We designed NEW-LiLFeS to first read in the program completely, and perform these analyses.

Fine-grained instruction set. The instruction set should be fine-grained enough to allow effective reduction of redundant operations. We designed the instruction set for NEW-LiLFeS so that instructions represent operations on as small a scale as possible, such as a single register move or table access.

Compositional unification algorithm. From a naive description of the unification algorithm we cannot determine in which situation a part of algorithm becomes redundant. In this thesis we constructed a unification algorithm in

a compositional way so that we succeeded to extract a useful set of context conditions.

The NEW-LiLFeS system's design is based on the architecture of the Prolog native-code compiler Aquarius Prolog [12], except for TFS unification, which we redesigned as already mentioned. The design of the unification compiler focuses on these four parts: (1) classification and analysis of unification operations for direct generation of TFS unification code, (2) design of abstract machine instructions for expressing the operations described in (1), (3) static analysis techniques for type hierarchies and definite clause programs for providing the context information used in code optimization, and (4) efficient built-in predicates for operations necessary for parsing.

1.3 Structure of This Paper

Chapter 2 describes related work. Chapter 3 describes the target language LiLFeS, and typed feature structures, that are the basis of LiLFeS. Thereafter, the architecture of the NEW-LiLFeS compiler and the abstract machine is described in Chapter 4. Chapter 5 details the algorithm of TFS unification. Chapter 6 shows the construction of unification code. And Chapter 7 describes optimization methods on TFS unification. Built-in predicates useful for parsing are illustrated in Chapter 8. Thereafter the whole system is evaluated in Chapter 9. Chapter 10 concludes the paper and shows future work.

Chapter 2

Related Work

2.1 Prolog Systems

Many studies have been done for efficient unification of Prolog terms. Warren's Abstract Machine [13] is an important breakthrough for Prolog implementation. WAM proposed a method to compile a Prolog term into a sequence of abstract machine instructions that, when executed, modifies another Prolog term on a heap into the result of unification. Such a compiled unifier (i.e. a sequence of instructions) can perform unification more efficient than the general unifier does, since the information of one structure is incorporated in the code and the unifier need not traverse both Prolog terms in run-time.

Aquarius Prolog [14] is one of the legitimate successors of WAM, which raised the concept of WAM to the level of an art. It performs global data-flow analysis to extract variable type information from the program. Moreover, to optimize the output code to the limit, it adopts Berkeley Abstract Machine, which uses a finer-grained instruction set that is close to RISC instructions. Aquarius Prolog runs 2 to 5 times faster than WAM-based Prolog systems. It is the basis of the design framework of NEW-LiLFeS.

2.2 TFS Systems

LIFE [15] is one of the most famous systems that handle feature structures. The system does not require any type definition, and allows arbitrary types and features (the term in LIFE is called ψ-term). Although LIFE can handle TFS, the operation of LIFE is relatively slow because it does not restrict feature structures to be well-typed.

ALE [5], a system that handles TFSs based on Carpenter's feature structure formalism [4], is a TFS interpreter on existing Prolog systems. ALE requires complete type definition, and restricts TFS to be totally well-typed. The first version of ALE was very slow at its processing. Later ALE is improved up to the practical speed, but it is still slower than other TFS systems.

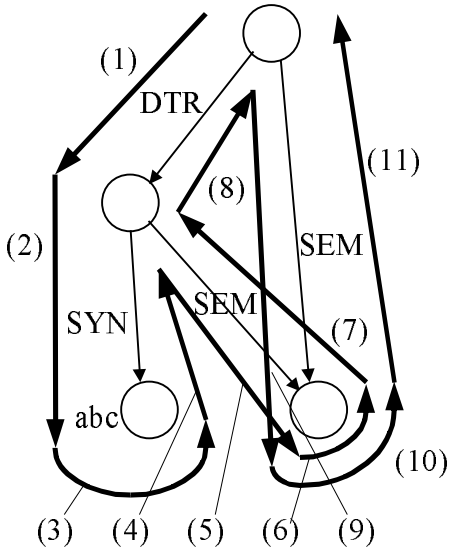
ProFIT [6] is a system that compiles TFS into Prolog terms. ProFIT relies TFS unification speed on the efficiency of latest Prolog systems. ProFIT also requires type definition, and define a mapping from TFSs to Prolog terms according to the type definition. Although the system can handle TFS faster than LIFE and ALE, the system consumes much memory since the compiled Prolog terms tend to be relatively large. Moreover, multi-inherited types are restricted in order to guarantee correct mapping to Prolog terms.

2.3 TFS Abstract Machines

The idea of an abstract machine that directly treats TFSs appears independently at two places.

AMALIA [16] is developed for the parser of Hebrew HPSG grammar. AMALIA compiles a TFS into a sequence of instructions which is designed based on WAM's instruction set. The representation of TFS on a heap is just like fixed-arity Prolog term using the information of given type definition. AMALIA is only a unification engine and it is built into the parser written in C.

An Abstract Machine for Attribute-Value Logics (AMAVL) [17] is the other one,



(a) Feature Structure F

(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	

(b) Compiled code C

Figure 2.1: Compilation of a TFS in an AMAVL

which is developed for an improvement of ALE system. AMAVL supports the same heap representation as AMALIA does. AMAVL compiles a TFS using a set consisting of 4 simple instructions. AMAVL also use the technique of type compilation for efficient general unification. AMAVL is also a mere unification engine, and is implemented on Prolog as a unification engine of ALE.

Figure 2.1 (b) shows a compiled code of a TFS show in Figure 2.1 (a). Each instruction corresponds to an operation in unification process; `PUSH` and `POP` indicate following to/retreating from a feature, `ADDNEW` indicates type coercion, and `UNIFYVAR` indicates structure-sharing.

2.4 LiLFeS

The LiLFeS abstract machine (OLD-LiLFeS) [7] is a full language system based on TFS. OLD-LiLFeS is a combination of a TFS unification engine, which is an extension of AMAVL, and definite clause handling engine, which is designed based on WAM. OLD-LiLFeS is the first implementation of AMAVL on low-level language (C++), and as efficient as AMALIA at TFS unification. Moreover, OLD-LiLFeS produces easy-to-learn and comfortable programming environment for TFS-based application with its Prolog-based syntax and a substantial set of built-in predicates. English and Japanese HPSG grammars (XHPSG [8] and SLUNG [2]), HPSG-based parser based on advanced parsing algorithm (TNT-Parser [9]), and a parallel programming environment (PSTFS [11]) are developed on the OLD-LiLFeS system.

One of the drawbacks of OLD-LiLFeS is that treatment of built-in data values, such as integers and strings, is very inefficient. This is because these data values are not included in the original design, so they are not amalgamated to the system well. Although these types of values are not necessary for representing a "pure" TFS, they play an important role in building practical applications using the language LiLFeS.

The LiLFeS native-code compiler (NEW-LiLFeS), the topic of this thesis, is a revised version of the OLD-LiLFeS. The definite clause handling engine is adapted from Aquarius Prolog instead of WAM in OLD-LiLFeS. And its TFS unification engine is designed from scratch (See Table 2.1). It not only clears the drawback of LiLFeS but also outperforms OLD-LiLFeS in TFS unification with optimized native code generation.

System	OLD-LiLFeS	NEW-LiLFeS
Implemented as	Emulator on C++	Native-code output
TFS engine	Extension of AMAVL	Originally implemented
DCP engine	WAM	Aquarius Prolog

Table 2.1: Comparison of Implementation between OLD-LiLFeS and NEW-LiLFeS

Chapter 3

LiLFeS, a Programming Language

LiLFeS is a unification-based programming language where the terms are represented in typed feature structures (TFSs). In this chapter, the Typed Feature Structure is briefly described, and LiLFeS, the programming language for TFSs are explained.

3.1 Overview of a Typed Feature Structure

Description of feature structures includes:

1. Description of types and features
2. Description of type hierarchy
3. Description of partial information and unification

3.1.1 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

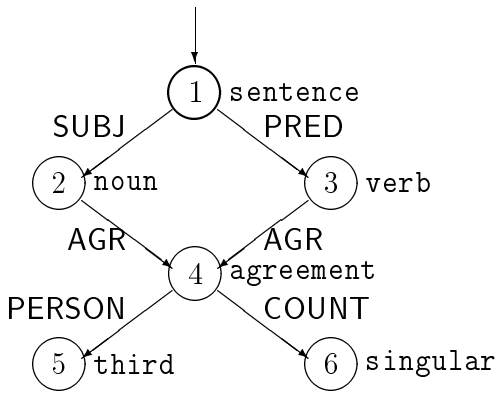


Figure 3.1: Sample Feature Structure in Graph Notation

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 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.

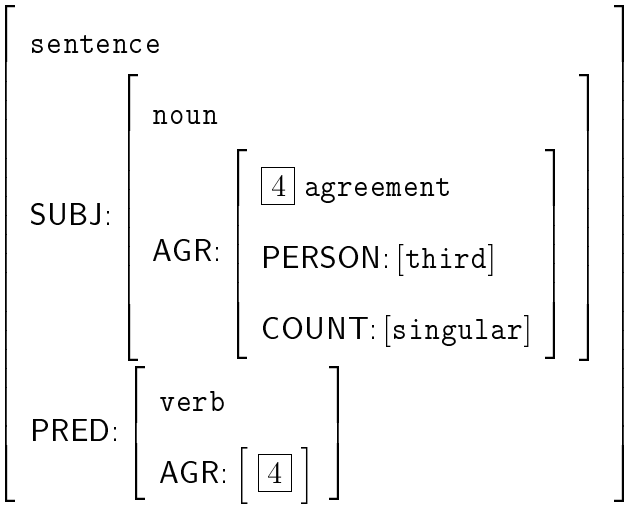


Figure 3.2: Sample Feature Structure in AVM Notation

3.1.2 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 include 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 $\bot$ (called "bottom"), which is the most general

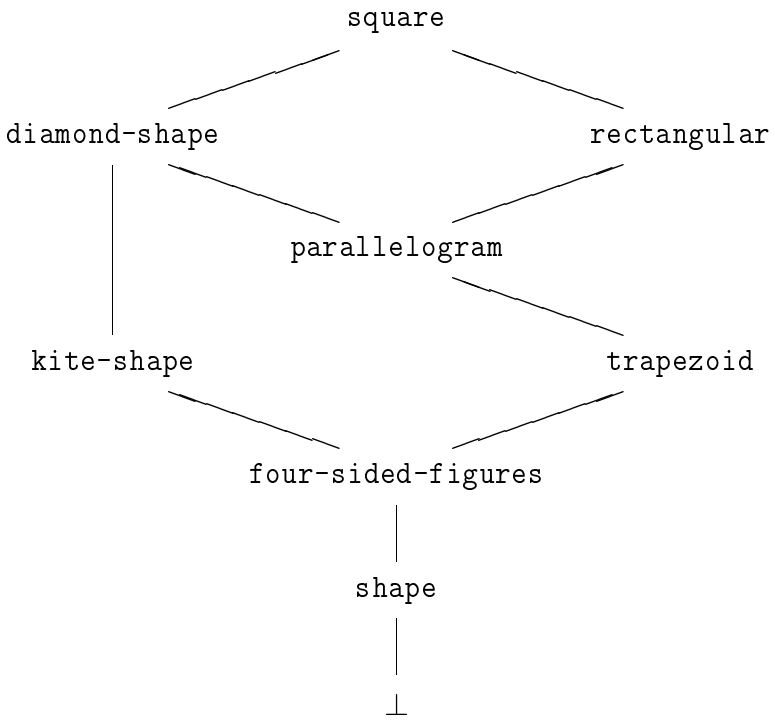


Figure 3.3: Type Hierarchy of Four-sided Objects

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

3.1.3 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*

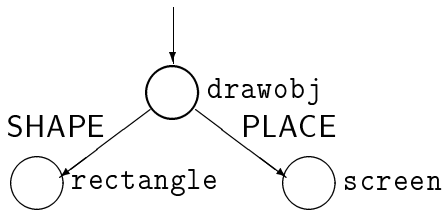


Figure 3.4: Feature Structure Representing "A rectangle is on the screen"

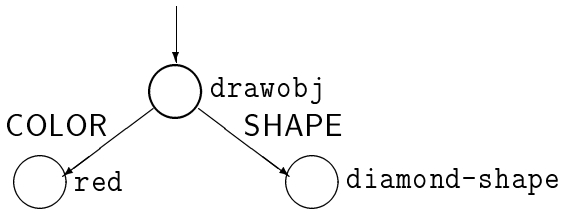


Figure 3.5: Feature Structure "It is also a red diamond-shape"

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 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 this 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`.

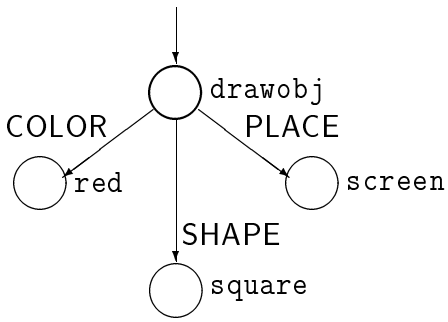


Figure 3.6: Feature Structure Unified

3.1.4 Appropriateness Condition

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 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 $\bot$ 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 *appropriateness* condition. Appropriateness is declared with a partial function Approp mapping features and types to types, where $\mathrm{Approp}(f, t)$ specifies the most general type of value that the feature f can have for a structure of type t.

Appropriateness should satisfy these two conditions, feature introduction (that is, for every feature f, there is a most general type $\mathrm{Intro}(f)$ s. t. $\mathrm{Approp}(f, \mathrm{Intro}(f))$

[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.

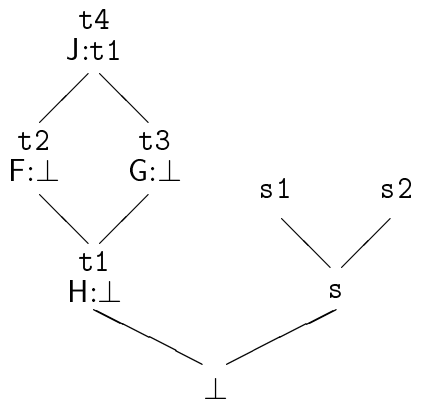


Figure 3.7: An Example Type Hierarchy

is defined) and right monotonicity (if Approp(f,t) is defined and $t \sqsubseteq t'$, then *approp*$(f,t) \sqsubseteq$ Approp(f,t')). Under these appropriateness conditions, TFS that LiLFeS handles can be restricted to *totally-well-typed* feature structure, that is, any structure of type t has all features f that Approp(f,t) exist and none of the other features.

Figure 3.7 is an example of the application of the rule. The notation H:⊥ below `t1` means that the feature H is appreciate for the type `t1`, and the feature H can take ⊥ 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.

In the rest of this thesis, the number of features that a TFS of a type t has appears in many places. Thus we define NF(t) as the number of features, e.g. NF$(t) = |\{f|\text{Approp}(f,t) \text{ is defined}\}|$.

3.2 LiLFeS, a programming language for TFSs

A program of LiLFeS consists of a set of definite clauses (which is similar to Prolog) and type definitions. Figure 3.8 is an example LiLFeS program. For

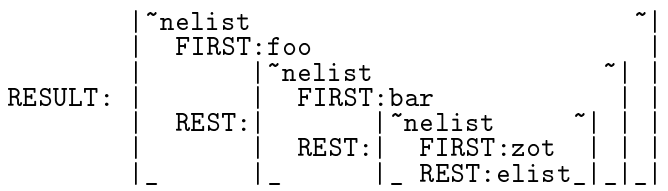
```
% Type definition.
data <- bot.
foo <- data.
bar <- data.
zot <- data.
alist <- bot.
elist <- alist.
nelist <- alist + FIRST:data, REST:alist.

% Predicate definition.
append <- pred.

% Clause declaration.
append( elist, X, X ).
append( (FIRST:A, REST:X), Y, (FIRST:A, REST:Z) ) :-
    append( X, Y, Z ).

% Sample Query.
?- append( (FIRST:foo, REST:(FIRST:bar, REST:elist)),
           (FIRST:zot, REST:elist),
           RESULT ).
```

(a) Sample LiLFeS program



(b) Result of (a)

Figure 3.8: LiLFeS Program Sample

details, refer to [18].

An example of a LiLFeS program is shown in Figure 3.8. The type definition says that the type `data` is subsumed by the type `bot` ($=\bot$), and so on. The type `nelist` is defined to introduce two features; the feature `FIRST` takes a node whose type is `data` or its subtype, and the feature `REST` `alist` or its subtype. LiLFeS

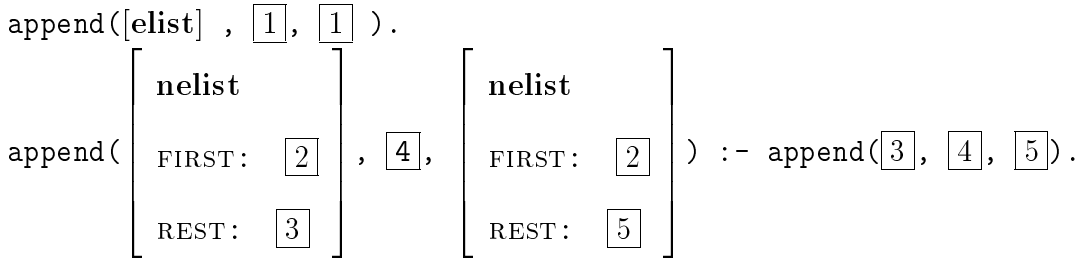


Figure 3.9: AVM Notation of the `append` clauses

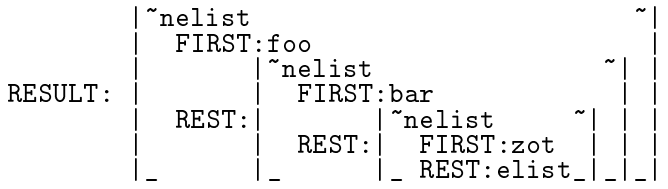


Figure 3.10: Result of the Example

requires all feature structures are totally well-typed [2].

The syntax of facts, clauses, and queries in LiLFeS conforms to those in Prolog. The declaration of `append` is the same for that in Prolog, except the notation of the list is written in TFSs. Variables in the program denote structure-sharings. The form of structure sharing is shown in Figure 3.9, the (informal) AVM notation of the `append` clauses.

The execution of a program is performed by SLDNF-resolution-like algorithm, that is, the same way as that in Prolog. The query in the sample program performs append of 2-entry list and 1-entry list. When the sample program is executed, LiLFeS performs inference in `append` predicate on the TFS and display the result of the query as in Figure 3.10.

[2] A feature structure is *totally well-typed* iff for all node q, the feature f from q exists for all features f that is appropriate to the type of q, and does not exist for all feature f that is not defined as appropriate to the type of q, and the content of any feature has an appropriate type. For details, see [4].

Chapter 4

Architecture of the LiLFeS Native-code Compiler

This chapter describes the process that a LiLFeS program is compiled into an executable file. After a brief explanation of its overall flow chart, the two most important parts, the design of NEW-LiLFeS Abstract Machine and the compilation steps of the abstract-machine-code generator are detailed.

4.1 Overall Process Flow

Figure 4.1 shows a brief flow chart of compiling process. The NEW-LiLFeS Code Generator takes a LiLFeS program as input, and generates a sequence of instructions for an imaginary processor named NEW-LiLFeS Abstract Machine. Then, machine-dependent Code Converter converts the instruction sequence into assembly code output. The output is then linked with a support library for NEW-LiLFeS Abstract Machine and built-in predicates, and the final executable is produced.

This thesis does not discuss the machine-dependent part of the compiler, because the design of NEW-LiLFeS Abstract Machine is intended that conversion to the machine-dependent assembly can be done by a simple mapping of instructions.

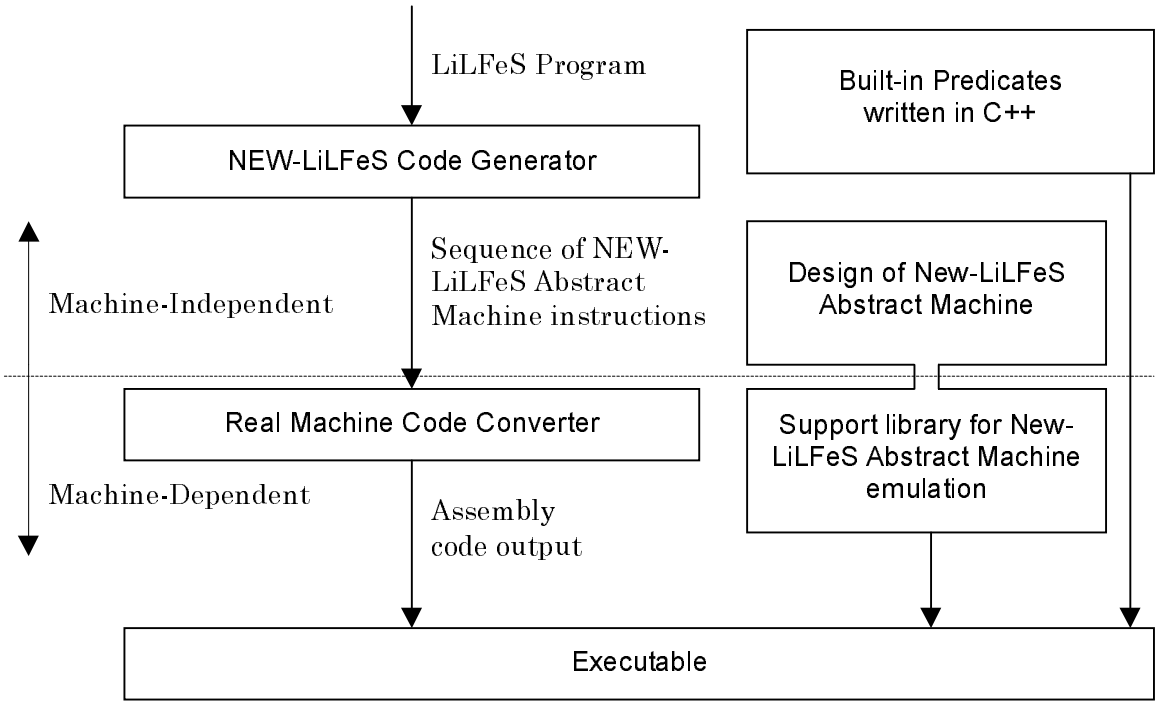


Figure 4.1: Overall Flow Chart of LiLFeS Native-code Compiler

Moreover, optimization on assembly code is almost unnecessary if NEW-LiLFeS Abstract Machine code is optimized properly, because many Abstract Machine instructions can be mapped to a single instruction of a real processor.

4.2 NEW-LiLFeS Abstract Machine

The NEW-LiLFeS Abstract Machine is designed based on Berkeley Abstract Machine, an execution model used in Aquarius Prolog [12]. Except extension for TFS manipulation, NEW-LiLFeS Abstract Machine inherits the same design policy as Berkeley Abstract Machine, that is, reduced granularity of instructions while retaining good features of Warren's Abstract Machine. A fine-grained instruction set allows extensive optimization on generated code by the LiLFeS compiler.

This section consists of three parts. First, the data areas and internal registers

of NEW-LiLFeS Abstract Machine are described. Then the way to represent TFS on the heap is explained. Finally, the instruction set of the abstract machine is introduced.

4.2.1 Data Areas and Registers

Figure 4.2 shows the data areas in the NEW-LiLFeS Abstract Machine. Dynamic data areas hold execution stack while static data areas holds compiled information, such as compiled code and type information.

- Dynamic data areas.

  Global Stack (Heap) An area that holds TFSs. This stack is also known as the *heap* although it follows a stack discipline.

  Environment Stack An area that holds variables local to a clause. For proper operation of backtracking, this area does not follow a strict stack discipline.

  Choice-point Stack A stack that holds choice points (execution state information necessary for backtracking).

  Trail Stack A stack that keeps records of changes on the heap. This record is used to unwind the state on the heap when backtracking occurs.

- Static data areas.

  Code area An area that holds compiled code. Compiled clauses as well as code generated by type compilation is stored in this area.

  Type Unification Table A table that holds information for unification of types. Indexed by a pair of types t_1 and t_2, each entry holds three values: the result of type unification, the pointer for type coercion code that changes a TFS with the type t_1 into a coerced structure with the type t_2, and the pointer for unification code that performs general unification between a TFS with the type t_1 and another TFS with the type t_2.

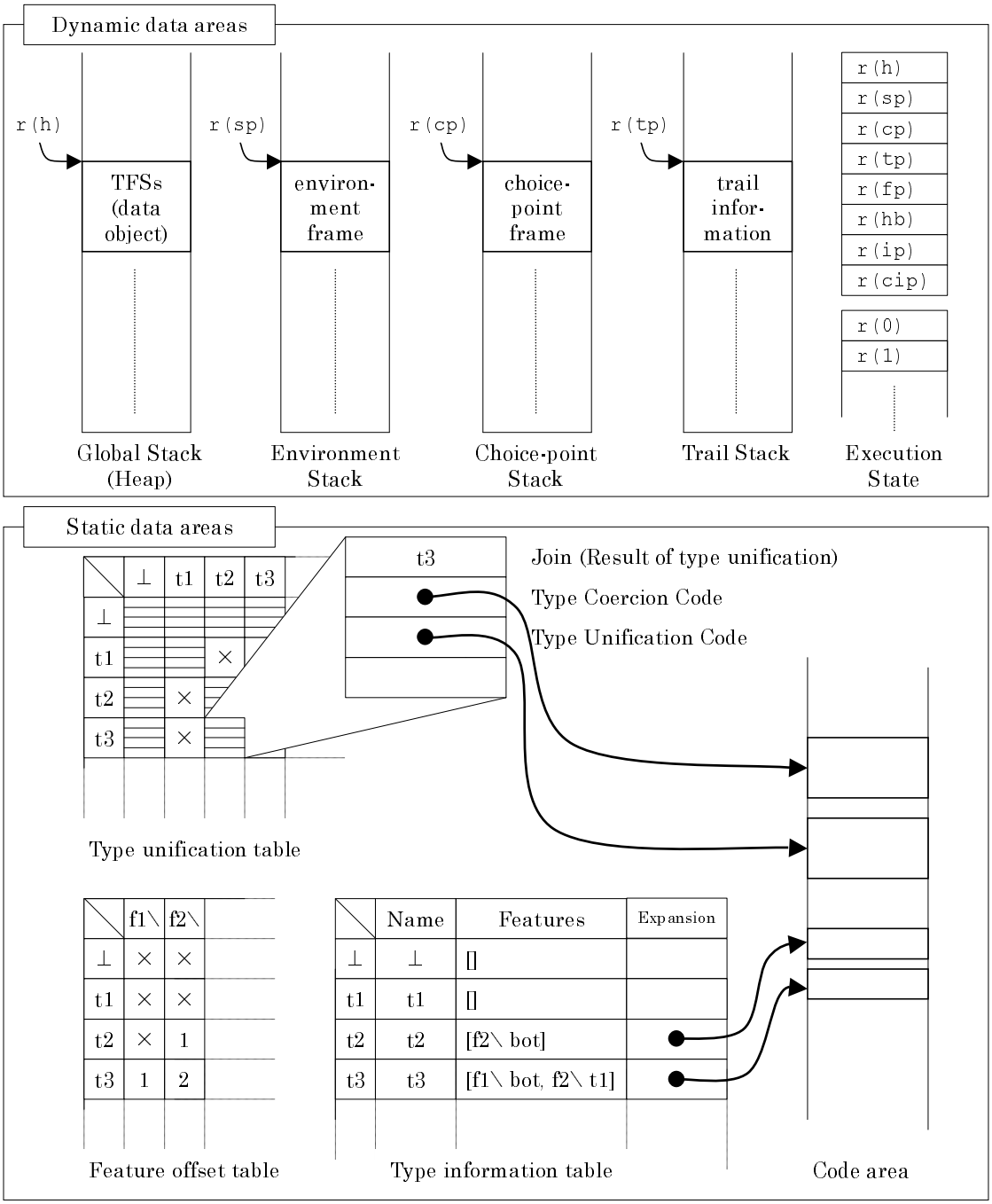


Figure 4.2: Data Area Configuration in the NEW-LiLFeS Abstract Machine

Feature Offset Table A table that holds offset information of features for types. This table regulates the order of features in TFS descriptions on the heap.

Type Information Table A table that holds information for each type. Entries accessible from this table are a type name, a list of features and its appropriateness information, and a pointer for expansion code that converts VAR-represented structure into the equivalent STR-represented structure.

4.2.2 Representation of TFS

A cell, the data unit to represent TFS, is a pair of a tag and a value. NEW-LiLFeS adopts basically the same TFS representation as OLD-LiLFeS uses, which is originally proposed by AMAVL [17]. Figure 4.3 shows the representation of various TFSs on NEW-LiLFeS Abstract Machine. In the thesis, a cell with a tag `TAG` and a value `VALUE` is denoted as `TAG^VALUE`.

A structure 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 contents of a feature, or the value of a feature, if it is represented in a single cell. The order of the features are stored in Feature Offset Table and Type Information Table[1]. If a structure has no features, or all substructures are the default type specified by the feature appropriateness, 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 `tptr` tag is called a *pointer* and used to refer a TFS on another location. Pointers can be chained until it reaches a cell with other than `tptr` tag; self-referencing pointer is not used as in WAM and Aquarius Prolog. A structure-sharing is represented by referring the same location.

[1]Both tables contain the relation of offset and feature. Feature Offset table gives a mapping from Feature to Offset, and Type Information Table gives a mapping from Offset to Feature.

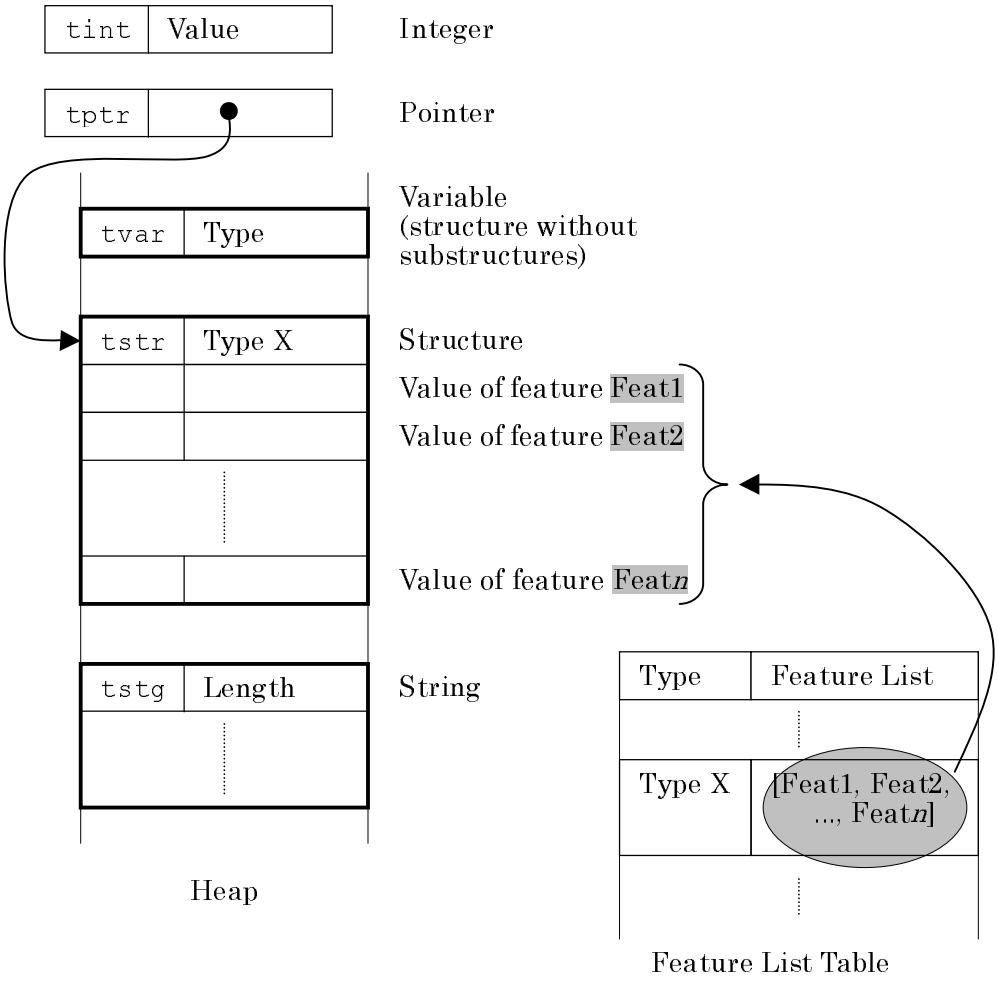


Figure 4.3: Heap Representation

Three tags, `tint`, `tstg`, and `tflo`, are used to represent an integer, string, and float value, respectively. Although these types of values are not necessary for representing a "pure" TFS, they play an important role in building practical applications using the language LiLFeS.

4.2.3 Instruction set

Table 4.1 and Table 4.2 show the instruction set and a brief description of the instructions of the NEW-LiLFeS Abstract Machine.

Many instructions are imported from Berkeley Abstract Machine of Aquarius Prolog. Table 4.1 shows these imported instructions. For detail of these instructions, refer [12].

We added several instructions to deal with TFSs. These instructions are displayed in Table 4.2. They are detailed in Section 6.1 of this thesis.

4.3 Compilation Steps

This section illustrates compilation steps in NEW-LiLFeS Compiler. Figure 4.4 shows the overview of the compilation steps.

Kernel Converter Kernel Converter takes the definite clause part of a LiLFeS program as input and removes syntax sugars such as If-Then-Else, arithmetic operations in `is` statement, and so on. The converted code is called Kernel LiLFeS.

Type Analyzer Type Analyzer takes the type definition part of a LiLFeS program, and analyzes the form of type hierarchy. In addition to subsumption relation, Type Analyzer checks various type conditions that allows deep optimization of unification code.

Type Compiler Type Compiler produces code that is used as a set of subroutine in unification operation. The detail is described in Section 5.2.2.

Dataflow Analyzer Dataflow Analyzer analyzes a Kernel LiLFeS program and derives lower-bound-value information for variables appeared in the program. Unification Compiler uses this information for further optimization of the generating code. NEW-LiLFeS uses Dataflow Analyzer for LiLFeS developed by Yoshida [19].

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\slash N,RegList,L)`	Create a choice point containing the registers listed in `RegList` and set the retry address to `L`.
`choice(I\slash N,RegList,L)` ($1 < I < N$)	Restore the argument registers listed in `RegList` from the current choice point, and modify the retry address to `L`.
`choice(N\slash 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 4.1: The Instruction Set in NEW-LiLFeS (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.
`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`.
`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 4.2: The Instruction Set in NEW-LiLFeS (2) Extended Instructions

Code Generator Code Generator compiles Kernel LiLFeS into a sequence of instructions of NEW-LiLFeS Abstract Machine. It consists of the following subroutines:

Disjunction Compiler Generates code for disjunction structure (backtracking and clause indexing).

Clause Compiler Generates code for sequence of goals keeping information of initialized variables.

Goal Compiler Generates code for each goal, which is either a unification, a predicate call, or inlined function (arithmetic operation etc.)

Unification Compiler Generates code for unification goal $V = T$, where V is a variable is T is any term of TFS.

Register Allocator Allocate abstract machine registers and local variables for generated code.

Peephole Optimizer Peephole Optimizer takes the generated instruction sequence and performs local optimization such as reduction of unnecessary assignment and jump chains.

This thesis focuses on the design of Type Analyzer, Type Compiler and Unification Compiler because the these parts are the most important extension to the Aquarius Prolog. For the detailed design of other parts, please refer to the design of Aquarius Prolog [12].

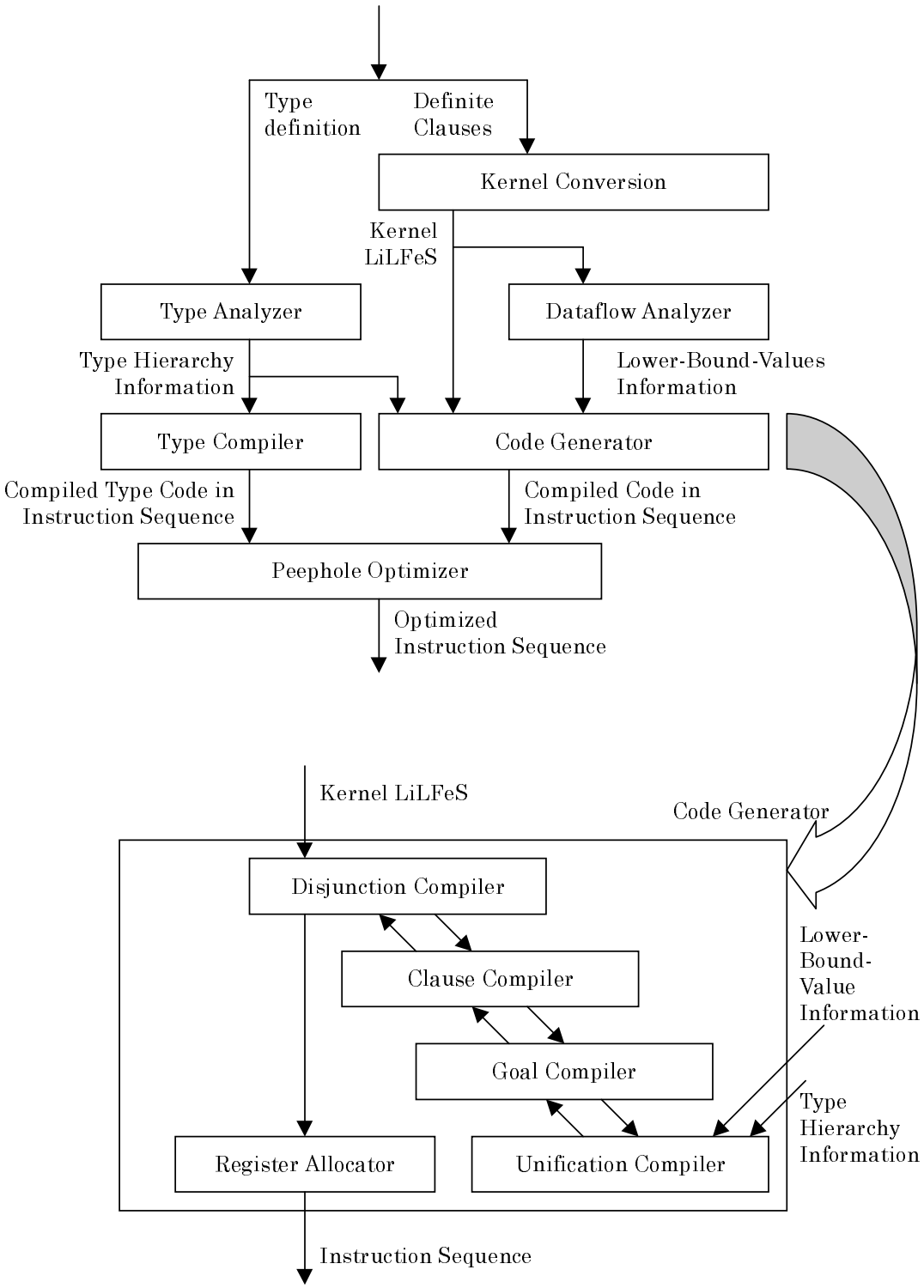


Figure 4.4: The Flow of Compilation

Chapter 5

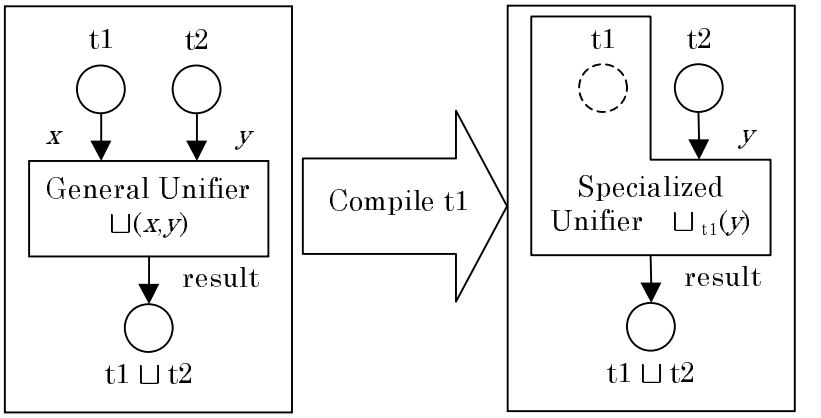
Unification Algorithm

5.1 Overview of Unification Code Generation

The basic idea of optimizing unification is to produce a unifier as specialized as possible. As described in Chapter 3, unification is an operation that takes two feature structures, and produces one (resulting) feature structure. On execution, we prepare a general unification routine that can take two arbitrary feature structures (See Figure 5.1 (a)). In the case that we have no information about V_1 and V_2 in the compiling unification goal $V_1 = V_2$, the unification must be compiled into the general unifier.

However, in many cases, we have information about the unifying structures that can be used to provide a more specialized unifier. For example, if we know one of the unified structure V_1 at compile-time, we can compile the goal $V_1 = V_2$ and produce a "specialized" unification routine using the known structure of V_1 (See Figure 5.1 (b)). This specialized unifier can perform unification more efficient than general unifier, since the knowledge of V_1 is 'hard-coded' in the specialized unifier, while the general unifier must collect information on both V_1 and V_2 at run-time.

The efficiency of the specialized unifier depends on how detailed context information we can extract from the source code as well as on how deeply the compiler



(a) General Unification (a) Specialized Unification

Figure 5.1: General Unification and Specialized Unification

applies the context information in producing unifier code. In the NEW-LiLFeS Native-code Compiler, dataflow analysis is performed in order to extract maximum amount of context information from the source code. The rest of this chapter describes the way to apply the extracted context information in the code generation.

5.2 Compilation of Types – General Unification Code

The general unifier code is used when no information is available on both of the structures being unified (for example, `p(V1,V2) :- V1 = V2`, and `p(V1,V2)` can be called from anywhere). However, since we require type definitions, we can assume that V_1 is the structure of one of defined types, and so as V_2. So, we can provide unifier code for any pair of structures that have known types when the type definitions are given. This is called *type compilation*.

The notion of type compilation for TFSs is proposed by AMAVL [17], and used by OLD-LiLFeS abstract machine.

The type compilation in the NEW-LiLFeS is an extension of AMAVL's in the

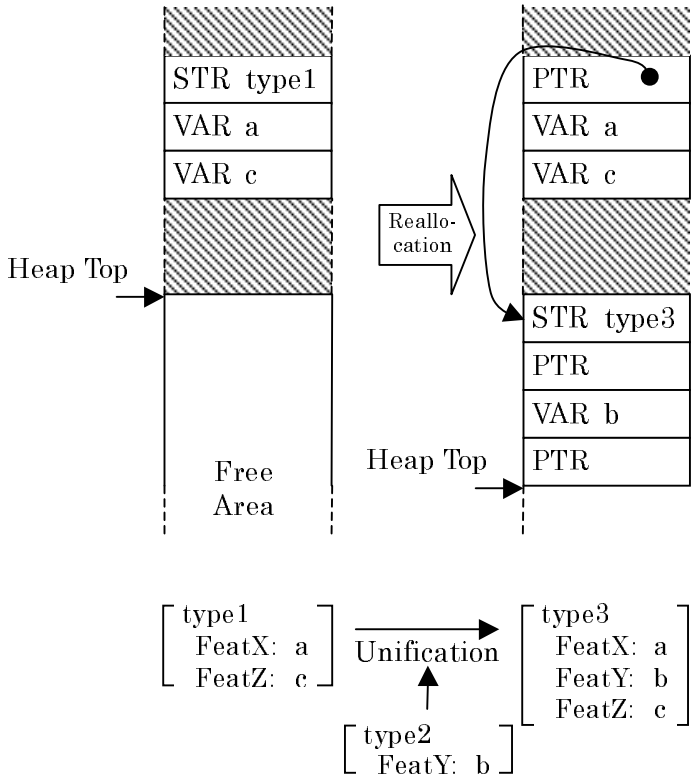


Figure 5.2: Reallocation of TFS in type coercion

following points:

- The code generated by type compilation now contains instructions for writing unified types and deciding binding directions. The code in AMAVL consists of only instructions for feature operations, and the other necessary operations should be done by general codes.
- AMAVL cannot handle the change of appropriateness, e.g. $\mathrm{Approp}(f,t) \neq \mathrm{Approp}(f,t')$ when $t \sqsubseteq t'$ and $\mathrm{Approp}(f,t) \neq \top$, while NEW-LiLFeS can.

5.2.1 Algorithm for General Unification

The requirement for *reallocation* of a structure is one of the notable differences between unification of TFS and that of Prolog first-order term. Figure 5.2 illustrates the process of reallocation. When the number of features of a TFS is increased from n to n' by unification, we need $n' + 1$ contiguous cells to represent the result. The original data area has only $n + 1$ cells, and the result feature structure cannot be fit. Thus we must allocate new $n' + 1$ cells from the heap, and put pointers in an appropriate way so that references to the old location are properly redirected to the new location.

The following is the core of the algorithm that performs general unification. Only the part of the algorithm that concerns `tvar` and `tstr` tags are shown. In this algorithm, a data area named Unification Queue is used to store pairs of TFSs to be unified. In NEW-LiLFeS Abstract Machine we have no data area named Unification Queue. However, since no choice point is created in this algorithm, the top of the Choice Point Stack is used as the Unification Queue.

1. While the Unification Queue has a value, Get a pair of TFSs (V_1, V_2) from the queue.

2. Dereference V_1 and V_2. If V_1 and V_2 appear to be token identical, no operation is necessary and go to the next entry of the Unification Queue.

3. Let t_1 and t_2 the type of V_1 and V_2, respectively.

4. Check the tags of cells that represent the top nodes of V_1 and V_2.

   (a) If both V_1 and V_2 are represented by `tvar` tags,
   Calculate $t = t_1 \sqcup t_2$.

      i. If $t = \top$, the unification fails.

      ii. Otherwise, set `tvar^`t as the result.

   (b) If V_1 is represented by `tvar` tag and V_2 by `tstr` tag,
   Calculate $t = t_1 \sqcup t_2$.

i. If $t = \top$, the unification fails.

ii. If $t = t_2$ (t_2 is subsumed by t_1, or $t_1 = t_2$), set V_1 as the pointer to V_2.

iii. if $t \neq t_2$, set V_1 as the pointer to V_2, and do the following.

A. If $\mathrm{NF}(t_2) < \mathrm{NF}(t)$ (the number of features increases by the change from t_2 to t), allocate $\mathrm{NF}(t) + 1$ cells from the heap and build a `tstr^`t node. Let V and V's features as the pointer to the newly allocated node.

B. If $\exists f(\mathrm{Approp}(f, t_2) \sqsubset \mathrm{Approp}(f, t))$ (any of the features of t has more specific feature restriction than that of t_2), put a pair $(f(V_2), \mathrm{Approp}(f, t))$ into the Unification Queue for each f so that all necessary type coercion to the substructures is applied later.

C. If either is not necessary, update V_2 to `tstr^`t.

(c) If V_1 is represented by `tstr` tag and V_2 by `tvar` tag,
Exchange V_1 and V_2, and do 4b.

(d) If both V_1 and V_2 are represented by `tstr` tags,
Calculate $t = t_1 \sqcup t_2$.

i. If $t = \top$, the unification fails.

ii. If $\mathrm{NF}(t) = \mathrm{NF}(t_1)$,
for each feature f that is appropriate for t_2, Put a pair $(f(V_1), f(V_2))$ into Unification Queue. set V_2 as the pointer to V_1.

iii. If $\mathrm{NF}(t) = \mathrm{NF}(t_2)$,
for each feature f that is appropriate for t_1, Put a pair $(f(V_1), f(V_2))$ into Unification Queue. set V_1 as the pointer to V_2.

iv. Otherwise,
allocate $\mathrm{NF}(t) + 1$ new cells from the heap, and store `tstr^`t at the top of the allocated cells (called V hereafter). For each feature f

that is appropriate for t,

A. If f is appropriate for both t_1 and t_2, bind $f(V)$ to $f(V_1)$, and put a pair $(f(V_1), f(V_2))$ into the Unification Queue.

B. If f is appropriate for only t_1, bind $f(V)$ to $f(V_1)$.

C. If f is appropriate for only t_2, bind $f(V)$ to $f(V_2)$.

D. If f is appropriate for neither t_1 nor t_2, set `tvar^`$\mathrm{Approp}(f, t)$ to $f(V)$.

If $\exists f((\mathrm{Approp}(f, t_1) \sqcup \mathrm{Approp}(f, t_2)) \sqsubset \mathrm{Approp}(f, t))$ (any of the features of t has more specific feature restriction than that of both t_1 and t_2), put a pair $(f(V), \mathrm{Approp}(f, t))$ into Unification Queue for each f so that all necessary type coercion to the substructures is applied later.

(e) (Other tags require another operations; omitted here)

This algorithm is guaranteed to terminate even if the TFSs contains cyclic terms and structure-shared nodes. This is because one loop of the algorithm decreases the number of token-independent nodes by 1, and if the number reaches one (that is, all pointers refers to the same node after dereferencing), all remaining entries in the Unification Queue is ignored by step 2. of the algorithm.

5.2.2 Type Compilation for General Unification

Since a big portion of the general unification algorithm is determined by the type of two unifying values, the code can be generated for each possible pair of types. Using such code, we can make general unification more efficient. This code generation is called *Type Compilation.*

Two parts of the algorithm is compiled. The compiled code for the step 4.(b) of the above algorithm is called *Coercion code,* because the code coerces a structure to a more specific type. On the other hand, The compiled code for the step 4.(d)

is called *Unification code,* because the code unifies two structures with recursive operations.

5.3 Compilation of Terms – Specialized Unifier Code

We use the name "specialized code" for the compiled code of the unification goal $V = T$, where V is a variable and T is any term of a typed feature structure. [1] In the following, the compilation of the goal $V = T$ is called as *the compilation of the term* T. Dataflow Analysis compilation step may provide some information about the variable V.

WAM [13] is the first proposal of the idea of term compilation for Prolog. In Prolog, the unification goal $V = T$ is executed as one of the two different actions according to the value of V:

1. When V is an unbound variable, the action is to construct T on the fly and bound to V. This is called *write mode.*

2. When V is a nonvariable term, the action is to check that the form of V matches T, and the algorithm is invoked recursively for the term's arguments. This is called *read mode.*

Thus, for Prolog, the goal $V = T$ should be compiled into code that checks V and perform one of these two actions.

As for TFSs, the problem is harder. This is because (1) V may have different types and a different number of arguments, and (2) the classification of the action also depends on T. This classification is important for specializing the compiled code as much as possible, but till now, no one has proposed any concrete classification based on the Carpenter's TFS definition [4].

[1] The unification $T_1 = T_2$, where T_1 and T_2 are two arbitrary terms, is reduced to $V = T$ in the Kernel Conversion compilation step.

In the following, the classification is described for each of the situations of T. t_V and t_T denotes the type of V and T, respectively.

5.3.1 TFS without substructure

In the case that T is a TFS without substructure (that is, leaf node of a DAG), or all substructures of T are the default specified by the feature appropriateness, the unification action is mainly a manipulation of types, that is, updating V's type to the unified type. However, the change of the type may affect V's features; for example, if the number of V's features is increased by the change, the node must be replaced to the newly allocated area on the heap as in the general unification.

Thus, the most general classification of the case becomes as the following:

1. If V has the `tvar` tag,
   check the unifiability and calculate t (the result of t_V and t_T).
   (a) If $t = t_V$ (t_V is subsumed by t_T, or $t_V = t_T$), no operation is necessary.
   (b) if $t \neq t_V$, unification succeeds by updating `V` to `tvar^`t.
   (c) If t_V and t_T are incompatible, the unification fails.
2. If V has the `tvar` tag, check the unifiability and calculate t (the result of t_V and t_T).
   (a) If $t = t_V$ (t_V is subsumed by t_T, or $t_V = t_T$), no operation is necessary and the unification successes.
   (b) if $t \neq t_V$,
      i. If $\mathrm{NF}(t_V) < \mathrm{NF}(t)$ (the number of features increases from t_V to t), allocate $\mathrm{NF}(t) + 1$ cells from the heap and build a `tstr^`t node. Set V and V's features as the pointer to the newly allocated node.
      ii. If $\exists F(\mathrm{Approp}(t_V, F) \sqsubset \mathrm{Approp}(t, F))$ (any of the features of t has more specific feature restriction than that of t_V), apply type coercion to all the features that have more specific feature restriction.

iii. If either is not necessary, the unification succeeds by updating V to `tstr^`t.

(c) If t_V and t_T are incompatible, the unification fails.

3. If V has other tags, the unification fails.

5.3.2 TFS without substructure, special type

The algorithm is slightly different if T can be unified to non-TFS instance, such as integer and string values. If T is `bot`, no operation is necessary for the unification. Otherwise, T is one of the *special types*, that is, `integer`, `string`, or `float`. For these types the following algorithm is applied:

1. If V has the `tvar` tag, check t_V, the type of V.

   (a) If t_V is the same as the type of T, the unification succeeds without any operation.

   (b) If t_V is `bot`, the unification succeeds by updating V to the type of T

   (c) Otherwise, the unification fails.

2. if V has the appropriate tag that is represented by T's type (e.g. `tint` for `integer`, `tstg` for `string`, and `tflo` for `float`), the unification succeeds without any operation.

3. Otherwise, the unification fails.

5.3.3 TFS with substructure

When T has one or more substructures that is more specific than the default value of a feature, the algorithm gets more complex. 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 are added according to the updated type.

1. If V has the `tvar` tag,
   check the unifiability and calculate t (the result of unification between t_V and t_T).
   (a) If $t = t_T$, construct T on the fly and bind V to the constructed T.
   (b) If $t \neq t_T$, check the consequence of the change from t_T to t.
      i. If $\mathrm{NF}(t) > \mathrm{NF}(t_T)$, construct T while putting added features between existing features in T.
      ii. If $\exists F(\mathrm{Approp}(t_T, F) \sqsubset \mathrm{Approp}(t, F))$, construct T while applying type coercion to all the features that have more specific feature restriction.
      iii. If either is not necessary, construct T on the fly except the type is changed from t_T to t.
   (c) If t_V and t_T are incompatible, the unification fails.
2. If V has the `tstr` tag,
   (a) If $t = t_V$, the unification succeeds by only applying the algorithm for V's substrctures and T's substructures recursively.
   (b) If $t \neq t_V$, check the consequence of the change from t_V to t and apply a type coercion.
      i. If $\mathrm{NF}(t) > \mathrm{NF}(t_V)$, allocate $\mathrm{NF}(t) + 1$ cells from the heap and build a `tstr`$\hat{}t_3$ node. Set V and V's features as the pointer to the newly allocated node.
      ii. If $\exists F(\mathrm{Approp}(t_V, F) \sqsubset \mathrm{Approp}(t, F))$, apply type coercion to all the features that have more specific feature restriction.
      iii. If either is not necessary, update V_1 to `tstr`$\hat{}t_3$.

After the type coercion, apply the algorithm for V's substrctures and T's substructures recursively.

(c) If t_V and t_T are incompatible, the unification fails.

3. If V has other tags, the unification fails.

5.3.4 Instances of integer, string, and float

An instance of integer can be unified to the same integer value, or the TFS of type `integer` or type `bot`. This is the same as string and float values. Thus the following algorithm is applied to the compilation of all of these instances:

1. If V has the `tvar` tag, check the t_V, the type of V.

   (a) If t_V is `bot` or the type that represents T (e.g. `integer` for integer value), the unification succeeds by binding V to T.

   (b) Otherwise, the unification fails.

2. If V has the appropriate tag that is represented by T (e.g. `tint` for integer, `tstg` for string, and `tflo` for float value), the unification succeeds by confirming V is equal to T.

Chapter 6

Code Generation

This chapter describes the code generation based on the algorithm discussed in Chapter 5.

6.1 Extensions to the Instruction Set

The instruction set used in NEW-LiLFeS is based on that of the Berkeley Abstract Machine, but some instructions are added or modified in order to manipulate TFS. Table 4.1 and Table 4.2 are a complete instruction set of NEW-LiLFeS. This thesis only describes the changed instruction is described in detail. Refer [12] to the detailed description of the other instructions.

The changes are roughly divided into three class: Additional instructions for type-table manipulation, Additional instructions for unification-queue operation and modification of a instruction for changed memory architecture.

6.1.1 Additions for Type-table Manipulation

Instructions that access type information tables are added to the instruction set of NEW-LiLFeS. Type information required by unification is calculated by Type Compilation prior to the unification and stored to tables. (see 5.2 for the detail of

Type Compilation.) In unification, an access to the tables always accompanies one or more checks of the retrieved value and conditional branches, and the sequence will not be separated by optimization. Thus I included accompanying checks and branches with these instructions. Some useful variant of the instructions are also provided.

Eight instructions are added to the instruction set: `exttype`, `coerce`, `coerchk`, `jccifa`, `jcc`, `jecifa`, `jec`, and `offset`. For brief description see Table 4.2.

6.1.2 Additions for Unification-Queue Operation

Two instructions, `unify_queue` and `proceed`, are added for unification-queue operation. In LiLFeS Type Compilation generates code for every pair of types for efficient operation on general unification. In the code, substructures that requires recursive operation are stored in Unification Queue (which is temporarily allocated on the top of Choice Point Stack) and the operation continues until all entries in the queue is processed. A `unify_queue` instruction puts a pair of values to be unified into the unification queue. A `proceed` instruction retrieves a pair of values from the queue and perform unification on them, or return if the queue is empty.

6.1.3 Modifications for Adapting Changed Memory Architecture

The `deref` instruction, which resolves the chain of the pointer references, is changed from BAM's 2-argument style to 3-argument style due to the difference of memory architecture in WAM and LiLFeS. Figure 6.1 and Figure 6.2 shows the difference. In WAM, the cell that marks the end of the dereference chain (the circled cells in the figures) is all of the information required; On the contrary, in LiLFeS, not only the end-of-the-chain cell but also the cell before the end-of-the-chain cell is required.

This is because we need to keep the address of the end-of-the-chain cell in the following operations of the `deref` instruction. In WAM-style memory architecture, the address is not necessary; both self-referencing `ptr` cells and `str` cells contain

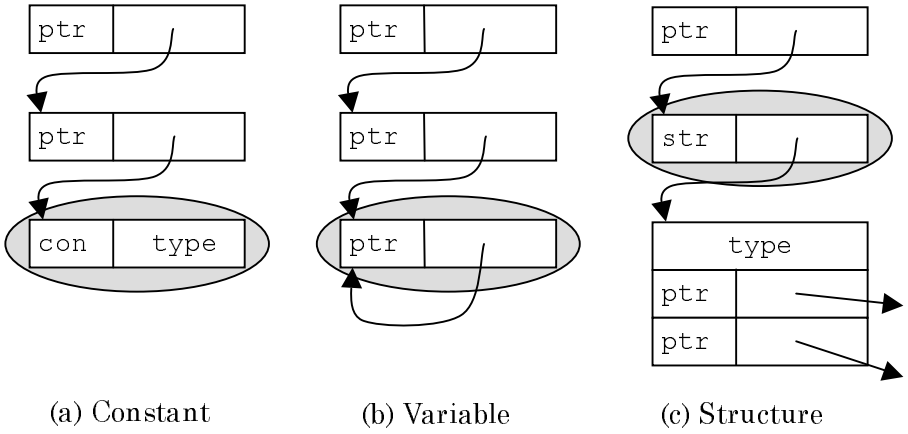


Figure 6.1: Deref operation in the WAM

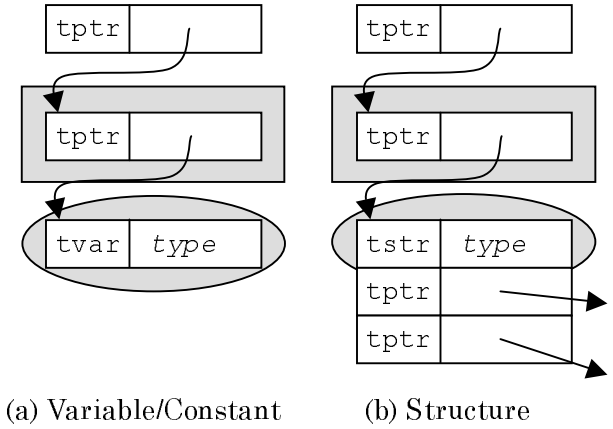


Figure 6.2: Deref operation in LiLFeS

sufficient address information, and `con` cells that represent constants can be copied without any problem. However, in LiLFeS-style memory architecture, the address is required to overwrite the dereferenced cell and to make structure-sharings using the reference to the dereferenced cell.

Note that WAM-style memory architecture cannot be used to represent TFSs. A self-referencing cell cannot represent a variable, since a variable has a type.

Moreover, `str`-tagged pointer cannot indicate the last reference to the structure since the structure may be "reallocated" any number of times. (See Section 5.2 for reallocation of TFS.)

One extra argument that holds the address of the dereferenced cell is added to the `deref` instruction. `deref V,W,X` means that X is a dereferenced cell of V and W is the address of X.

6.2 Construction of the Code

This section describes the construction of the compiled unification code using algorithm described in Chapter 5 and instructions described in Section 6.1. The code described in this section is the most general code, that is, the code which can process any possible situation. Optimization techniques for the code is discussed in Chapter 7.

6.2.1 TFS without substructure

The following is the code conforming to the algorithm described in Section 5.3.1, page 41. General unification routine is used for the part 2.(b)i./ii. of the algorithm, which is different for all possible result type t, and cannot be totally hard-coded in general.

Instruction	Description
`deref V, W, X`	Dereference the pointer stored in the register `V`
`switch tstr, X, L1, L2, fail`	Branch according to the tag : corresponding to the case-branch of 1., 2., and 3.
`label L1`	Case of 1.

continued on next page

Instruction		Description
exttype	X, Y	Extract V's type to Y
coerce	Y, type, Z, fail	Access to the table and get the unified type (fail if incompatible)
jump	eq, Y, Z, L3	In case of 1.(a), jump to the exit
trail	W	1.(b); Save the original value for backtracking
move	tvar^Z, [W]	1.(b); Overwrite new value (tvar^t')
jump	L3	Jump to the exit
label L2		Case of 2.
exttype	X, Y'	Extract V's type to Y
coerce	Y', type, Z', fail	Access to the table and get the unified type (fail if incompatible)
jump	eq, Y', Z', L3	In case of 2.(a), jump to end
jccifa	Y', Z', W, L3	Jump to general unification routine if 2.(b)i./ii.
trail	W	Case of 2.(b)iii. ; Save the original value for backtracking
move	tstr^Z', [W]	Overwrite new value (tstr^t')
jump	L3	Jump to the exit
label L3		

6.2.2 TFS without substructure, special type

The following code is the generated code the algorithm described in Section 5.3.2, page 42 when t_T is integer.

Instruction		Description
deref	V, W, X	
switch	tint, X, L2, L1, fail	
label L1		
jump	eq, tvar^integer, X, L2	
jump	ne, tvar^bot, X, fail	
trail	W	
move	tvar^integer, [W]	

continued on next page

Instruction		Description
jump	L2	
label L2	End	

6.2.3 TFS with substructures

The algorithm described in Section 5.3.3, page 42 can be represented in the following code. Vertical dots are the placeholder where generated code for substructures should be stored.

General unification routine is used for two parts of the algorithm, 1.(b)i./ii. and 2.(b)i./ii., which deeply depend on the result type.

Instruction		Description
deref	V, W, X	Dereference the pointer stored in the register V
switch	tstr, X, L1, L2, fail	Branch according to the tag : corresponding to the case-branch of 1., 2., and 3.
label L2		Case of 2.
exttype	X, Y	Extract V's type to Y
coerce	Y, *type*, Z, fail	Access to the table and get the unified type (fail if incompatible)
jump	eq, Y, Z, L6	In case of 2.(a), go to recursive code directly
jccifa	Y, Z, W, L7	Call general unification in the case 2.(b)i./ii.
trail	W	2.(b)iii.; Save the original value for backtracking
move	tstr^Z, [W]	Overwrite new value (tstr^t')
jump	L6	Jump to recursive part
label L7		Return from general unification
deref	W, W	Extra dereference may be necessary
label L6		Start of recursive part (read-mode)
offset	Z, *feat1* “*verb+* “+, O1	Read offset of feature *feat1* “*verb+* “+ from the table

continued on next page

	Instruction	Description
	`adda`	`W, O1, V1`	Calculate the address of substructure in *feat1* "*verb+* "+
	⋮		Recursive unification code for substructure in *feat1* "*verb+* "+ is put here
	`offset`	`Z,` *feat2* "*verb+* "`+, O2`	Read offset of feature *feat2* "*verb+* "+ from the table
	`adda`	`W, O2, V2`	Calculate the address of substructure in *feat2* "*verb+* "+
	⋮		Recursive unification code for substructure in *feat1* "*verb+* "+ is put here
	⋮		Repeat until all necessary substructures are processed
	`jump`	`L3`	Jump to exit
`label L1`			Case of 1.
	`exttype`	`X, Y'`	Extract `V`'s type to `Y'`
	`coerce`	`Y',` *type*`, Z, fail`	Access to the table and 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 for 1.(a) and 1.(b)
	`push`	`tstr^`*type*`, r(h), 1`	If 1.(a), store the top of the structure
	`jump`	`L5`	Jump to recursive part
`label L4`			
	`jecifa`	`type, Z, W, L7`	Call general unification in the case 1.(b)i./ii.
	`push`	`tstr^Z, r(h), 1`	Case 1.(b)iii. Store the top of the structure
	`jump`	`L5`	Jump to recursive part
`label L5`			Start of recursive part (write-mode)
	`push`	`...., r(h), 1`	Put the value of feature *feat1* "*verb+* "+
	`push`	`...., r(h), 1`	Put the value of feature *feat2* "*verb+* "+
	⋮		Build all substructures in the same way
	`jump`	`L3`	
`label L3`			End of the unification

Chapter 7

Optimization

This chapter describes the optimization of the unification code. The code described in Chapter 6 is the most general code, which can process any possible situation. However, in many situations not all of the code is necessary, and removing the redundant part of the code makes the code more efficient. In this chapter, two kinds of useful situation information and the way to use the information for the context is described. One is information of types of feature structures being compiled, and the other is variable information derived from the context information.

7.1 Optimization by Type Hierarchy Analysis

Some of the 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 code that deals with features and substructures is not necessary. In this case, the lighter, specialized unification code without treatment of features and substructures can be used instead of full and slower unification code.

In this section, we pursue conditions of types that can be unified with specialized unification code, and methods for extracting the conditions from type hierarchy and specializing unification code using the conditions.

7.1.1 Optimization Conditions

Useful condition of a type for optimization is what determines the conditional branch in the unification algorithm in compilation. The followings are the conditional branches used in the algorithm in Chapter 5:

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

I derived the following conditions useful for optimization according to the type t so that I can optimize the unification.

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

Condition *Never_Feature* 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* Any type that have features does not change by the unification with t. In this case, the equality check always succeeds if V has `tstr` tag.

Condition *Fixed_NF* For any t's subtype t', $\mathrm{NF}(t) = \mathrm{NF}(T') > 0$. In other words, any subtype of the type t have no additional feature. Condition *Final* implies this condition.

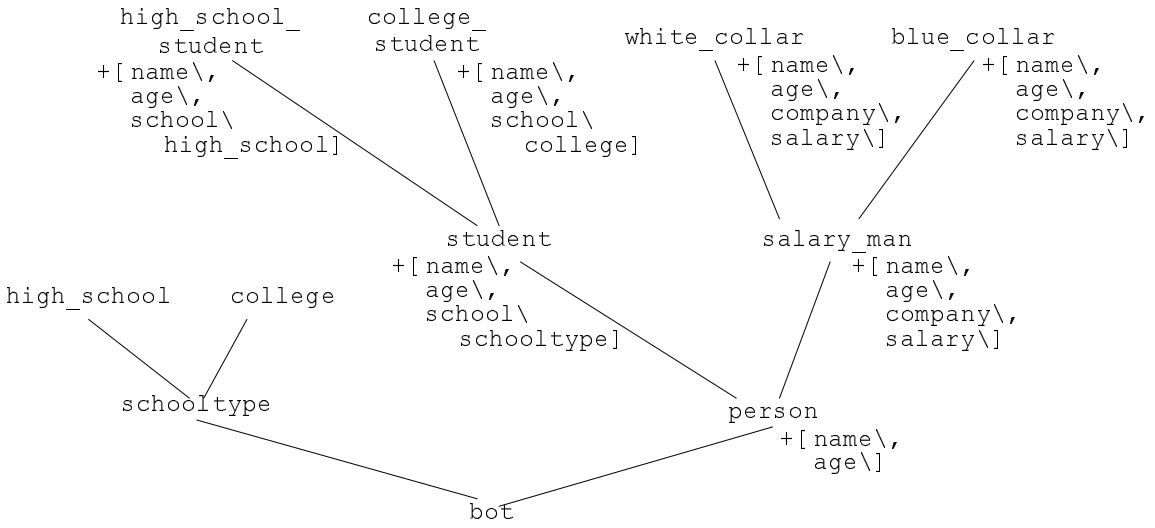


Figure 7.1: Sample Type Hierarchy

Condition *Preserve_NF* For any type t', if t and t' are compatible, $\mathrm{NF}(t \sqcup t') = \mathrm{NF}(t')$. In other words, 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* For any t's subtype t', $\forall f \mathrm{Approp}(f, t) = \mathrm{Approp}(f, t')$. In other words, any subtype of t has the same appropriateness condition as t has. Condition *Final* implies this condition.

Condition *Preserve_Approp* For any type t', if t and t' are compatible, $\forall f \mathrm{Approp}(f, t \sqcup t') = \mathrm{Approp}(f, t')$. In other words, a TFS of any type suffers no change on its appropriate conditions from successful unification with the type t. Condition *Preserve_StrType* implies this condition.

Suggest a type hierarchy is given as in Figure 7.1. Optimization conditions of the types on the type hierarchy is shown in Table 7.1.

Type	*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	
`salary_man`				X		X	
`white_collar`	X			X		X	
`blue_collar`	X			X		X	

Table 7.1: Condition Table of the Sample Type Hierarchy

7.1.2 Optimizing by Conditions

Conditions declared in Section 7.1.1 provides wide possibility of eliminating redundant codes. In the following we demonstrate the effect of elimination in three cases of conditions on the type t of the TFS being compiled.

Effect of the Condition *Never_Feature*

The condition *Never_Feature* allows a drastic optimization on the unification code of a TFS without substructure. Since t cannot be unified with any TFS with substructures, any value with `tstr` tag can be rejected with no further analysis. The

following code shows this optimization on the code for TFS without substructures.

Non-optimized	Optimized
`deref V, W, X`	`deref V, W, X`
`switch tstr, X, L1, L2, fail`	`switch tstr, X, L1, fail, fail`
`label L1`	`label L1`
`exttype X, Y`	`exttype X, Y`
`coerce Y, type, Z, fail`	`coerce Y, type, Z, fail`
`jump eq, Y, Z, L3`	`jump eq, Y, Z, L3`
`trail W`	`trail W`
`move tvar^Z, [W]`	`move tvar^Z, [W]`
`jump L3`	`jump L3`
`label L2`	
`exttype X, Y'`	
`coerce Y', type, Z', fail`	
`jump eq, Y', Z', L3`	
`jccifa Y', Z', W, L3`	
`trail W`	
`move tstr^Z, [W]`	
`jump L3`	
`label L3`	`label L3`

Effect of the Conditions *Final*, *Fixed_NF*, *Fixed_Approp*

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 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`**	*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`	

continued on next page

Original Code	Optimized Code	Condition
⋮	⋮	
`jump L3`	`jump L3`	
`label L3`	`label L3`	

Effect of the 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`	
`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`	
⋮	⋮	

continued on next page

Original Code	Optimized Code	Condition
`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`	

7.2 Optimization by Global Data-Flow Analysis

Global data-flow analysis is a method to derive information of variables from the source code. Aquarius Prolog [12] performs 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 of data-flow analysis is expected to be larger than the benefit on Prolog if global data-flow analysis gives type information.

In the NEW-LiLFeS native-code compiler, global data-flow analysis module developed by Yoshida [19] is used. The data-flow analysis module produces the lower-bound value of a variable, that is, the commonpart (meet) of all the possible value of the variable. This section describes how the lower-bound value information obtained by the data-flow analysis is used to optimize the generated code.

T Subsumes Lower-Bound Value

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

Compound Lower-Bound Value

When the lower-bound value is a compound term, it means that the variable V always holds a value with `tstr` tag. This means that tag checking and code corresponding to `tvar` are not necessary.

Lower-Bound Value Restricts Unifiable Type Set

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 `cons`. Since Type Hierarchy Analyzer determins that `cons` is *Final*, the restrict that unifiable type set to {`list`, `cons`}. Thus we can replace the `coerchk` instruction, which accesses type table to check V's type is unifiable, into a set of simple conditional jumps.

Chapter 8

Built-in Predicates

We provide a set of built-in predicates useful for parsing systems. Pure unification-based programming language has restrictions for developing efficient programs, such as difficulty to handle non-destructive operations and unrecoverable functions. Since our primary goal is to build practical NLP systems, a prime requirement for the design of LiLFeS is to be appropriate for development of efficient parsing systems, especially a fast implementation of chart parsers. In this chapter, we first discuss which predicate is useful for the development of an efficient parser, and the implementation of the built-in predicates with a notion of *TFS normalization.*

8.1 Useful Predicate for Efficient Parser

Unification is not the only operation that composes parsing process; other operations in parsing should be enhanced for an efficient parser. Figure 8.1 shows the four basic operations in a chart parser: the parser retrieves edges from an array by copying, combine them by unification, and perform factoring (checking out of duplicated edge generation) by equivalence checking, and store the result edge to the array by copying again, and this process is repeated until all edges are processed.

Copying of TFS and equivalence checking are relatively inefficient operations,

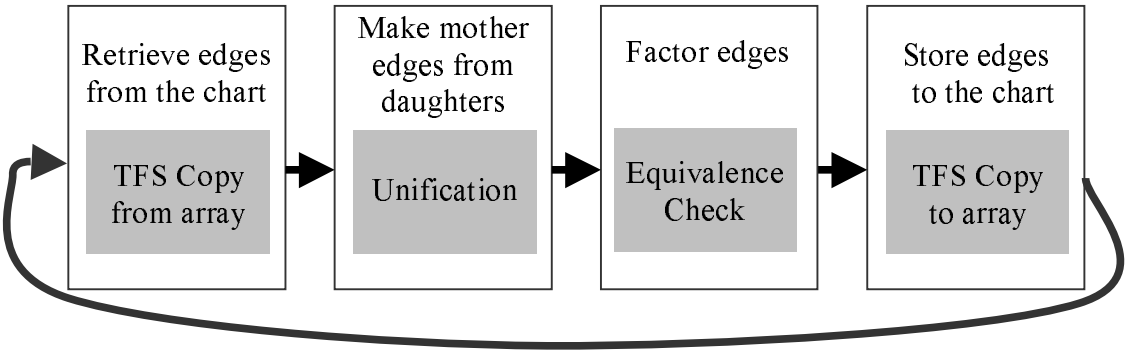


Figure 8.1: Parsing principle from a viewpoint of engineering

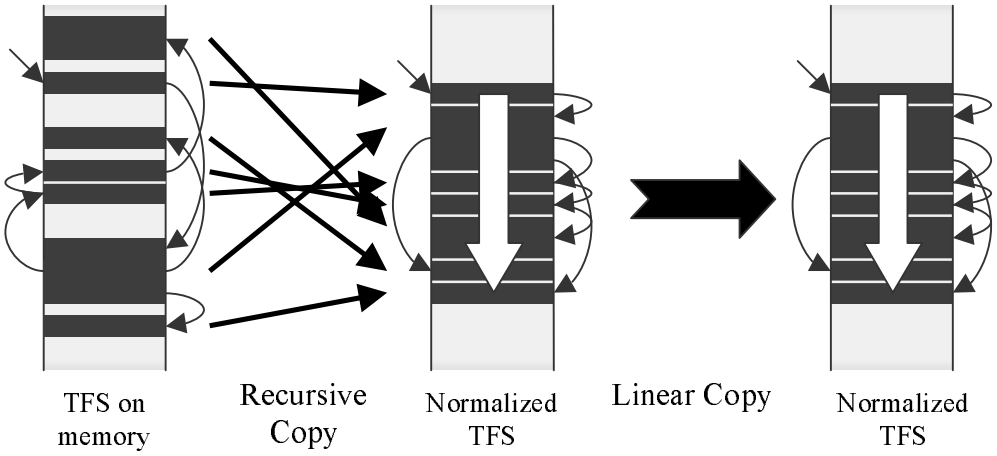


Figure 8.2: TFS Copy and Normalization

even if provided in built-in predicates. This is because recursive traversing and marking operation is required in order to preserve structure-sharings across these operations. In the next section an idea to provide efficient copying and equivalence checking of TFSs is described.

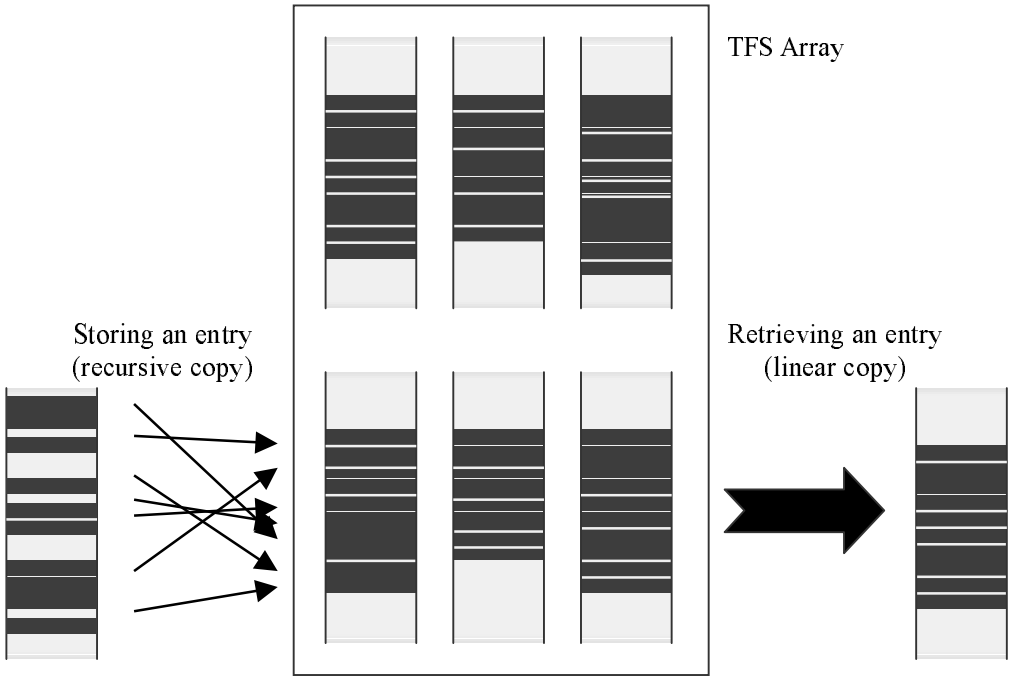


Figure 8.3: Array Operation with Normalization

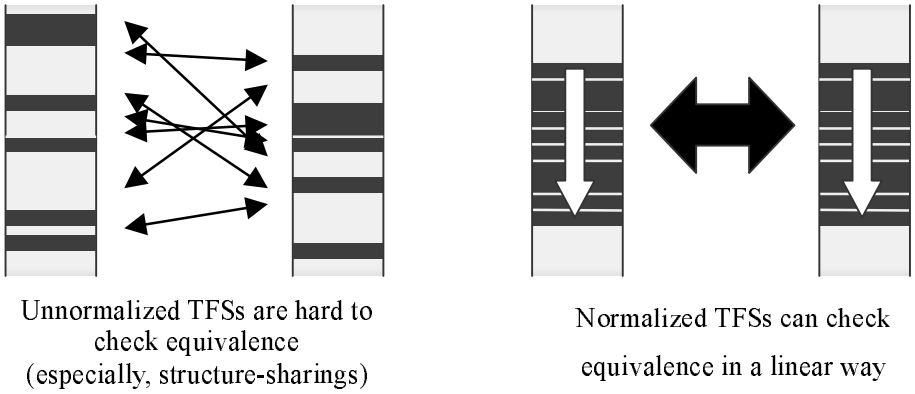


Figure 8.4: Equivalence Check with Normalization

8.2 TFS Normalization

Efficient built-in predicates of TFS copying, array operation, and equivalence check are provided based on the idea of TFS normalization. Usually, copying of

TFSs is inefficient since (1) recursive traversing is required to collect cells of TFSs scattered on the heap, and (2) marking operation is required to detect structure-sharings and cyclic terms. However, if the feature structure is stored on a continuous region on memory, the copying can be done in a linear way, using a simple loop. This is the basic idea of TFS normalization.

We define a *normalized TFS* as a TFS represented on a heap such that (1) All nodes of the TFS are stored in the depth-first traversing order into the ascending direction of the heap address, and (2) All nodes of the TFS are stored in one contiguous area of the heap. Note that the result of a normal (recursive, depth-first) TFS copy produces a normalized TFS as a result. Once copied, the TFS can be efficiently copied in a linear way (See Figure 8.2).

We applied this idea to the built-in TFS arrays. A TFS is copied in order to be stored into an array, and copied again in order to be retrieved from the array. All TFSs stored in an array is normalized, therefore the efficient linear copy routine can be used for the retrieval as in Figure 8.3. Since a stored TFS value is usually retrieved several times, efficient linear copy is performed several times as frequent as normal copy.

The same idea can be applied to the equivalence check. Equivalence check between normalized TFSs are much easier than equivalence check between unnormalized TFSs, since structure sharing is promptly checkable. (See Figure 8.4)

Chapter 9

Evaluation

This chapter evaluates the performance of the LiLFeS native-code compiler. Evaluation is done in three aspects: comparison to other Prolog systems, comparison to other TFS systems, and evaluation of optimization techniques.

We performed experiments on Sun UltraSparc and Intel Pentium II. Using these two computing environments, we can show the maximal performance of the LiLFeS native-code compiler and give a fair comparison to other systems.

Currently, NEW-LiLFeS has two code generators: A C++-code generator and a Pentium assembly generator. The C++-code generator lets us run the compiled code on any processor running GNU C++, while the Pentium assembly generator produces more efficient code than the C++-code generator does. However, many other systems are optimized for Sparc processors. SICStus Prolog, for example, has a native-code compiler for Sparc, but none for Pentium machines. Aquarius Prolog cannot be executed on Pentium systems either. Thus performance data on Sparc processors is shown in addition to that on Pentium processors in order to provide a fair comparison with other systems. We will show the maximal performance of the NEW-LiLFeS on Pentium.

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

(Environment: Sun UltraSparc 1/167MHz with 128MB memory)

(Unit: Seconds in user time)

Table 9.1: Performance Evaluation

9.1 Comparison to Other Prolog Systems

Table 9.1 shows the performance evaluation in comparison to other Prolog systems, SICStus [20] and Aquarius Prolog [14]. As can be seen, OLD-LiLFeS is quite slower than other Prolog systems; treatment of built-in data types such as integer was one of the bottlenecks of OLD-LiLFeS. In contrast, the new LiLFeS native-code compiler is quite efficient, and its performance is approaching to that of the latest Prolog systems. Considering that LiLFeS treats more complex data structures than pure Prolog, we can regard this performance as a competitive one.

9.2 Comparison to Other TFS-based Systems

A comparison between NEW-LiLFeS and other TFS Systems is given in Table 9.2. NEW-LiLFeS shows a speedup of 3-5 times from the OLD-LiLFeS. It is

Environment	System	Simple	HPSG
Sun UltraSparc 1/250MHz	LiLFeS Native-Code Compiler	2.54	1.30
	OLD-LiLFeS	3.89	1.61
	ProFIT on SICStus Native	1.00	5.17
	ProFIT on SICStus Emulator	2.12	5.87
	ALE 3.1 on SICStus Native	24.85	3.86
	ALE 3.1 on SICStus Emulator	70.65	9.77
Intel Pentium II/400MHz	LiLFeS Native-Code Compiler	0.64	0.45
	LiLFeS Byte-Code Emulator	2.23	0.97
	ProFIT on SICStus Emulator	3.28	5.07
	ALE 3.1 on SICStus Emulator	55.7	7.72

(Simple: toy parser that parses English sentence)

(HPSG: HPSG parser and grammar distributed with ALE)

(Unit: Seconds in user time)

Table 9.2: Performance Evaluation

still slower than ProFIT + SICStus native-code compiler in 'simple' experiments, though the difference is very small. We think the reason for this is the different traversing order between ProFIT + SICStus (breadth-first) and LiLFeS native-code compiler (depth-first) .

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

Program	Execution Speed			Object file size		
	Optimized	Plain	Ratio	Optimized	Plain	Ratio
fib(30)	1.41	1.63	0.87	5592	7948	0.70
rev(1000) 10 times	1.96	2.54	0.77	11516	15812	0.73
Simple	1.07	1.23	0.87	76936	117292	0.66

(Unit: Seconds in user time)

Table 9.3: Evaluation of Optimization

that the native code compiler improves the TFS handling capability significantly.

9.3 Optimization

Table 9.3 shows the results of our experiments for evaluating the effect of the optimization measures described in Chapter 7[1]. In all tests the optimization reduces code side around 60% and produced 15–29% speedup. This means that unification codes are efficiently specialized for both built-in data types and TFSs.

The rev(1000) test benefits the most of the optimization because Global Dataflow Analysis derives the lower-bound values of many variables as `list`, and the optimization technique described in Section 7.2 reduces table accesses for almost all list operations.

[1]Experiments in this section have been carried out using a preliminary version of NEW-LiLFeS, and their results cannot be compared to the experiments in Section 9.1 and Section 9.2.

9.4 Discussion

Because of remaining bugs in the implementation of the native-code compiler, we cannot yet perform experiments on real-world text parsing. However, we can estimate the result from the results on emulator-based LiLFeS (160 milliseconds using the sophisticated algorithm [9]) and speed ratio between emulator-based LiLFeS and native-code compiler (2 to 3 times speed-up). The estimated parsing time is 60ms - 80ms per sentence, so we can say that we will be able to achieve our goal of 100ms in near future.

Chapter 10

Conclusion and Future Work

I developed a native-code compiler for LiLFeS, a logic programming language for TFSs. In the development of the compiler, I redesigned the TFS unification algorithm to allow extensive optimization of the generated code for TFS unification, and extended instruction set to express the algorithm on the NEW-LiLFeS Abstract Machine. I also designed unification optimizer that uses information given by a global dataflow analyzer and type hierarchy analyzer.

The native-code compiler is fully implemented. The compiled code is more than twice as efficient as existing systems, and it is supposed to be able to achieve our goal of 100ms average parsing time per sentence in real world corpus.

Future work on the LiLFeS native-code compiler should be enrichment of surrounding tools and documents as well as enhancement of the compiler itself. Since the LiLFeS system has been requested for distribution from several sites, it is urgent that we provide English documents and a debugger. Also, native machine support for other than Intel CPU is necessary for the compiler to be used widely. Furthermore, introduction of other optimization techniques developed on OLD-LiLFeS, such as TNT Parser [9], should be done.

References

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

[2] Yutaka Mitsuishi, Kentaro Torisawa, and Jun'ichi Tsujii. HPSG-style underspecified Japanese grammar with wide coverage. In *Proceedings of COLING–ACL '98*, pages 876–880. Association for Computational Linguistics, August 1998.

[3] 光石 豊, 鳥澤 健太郎, and 辻井 潤一. HPSG による統語解析のための統計モデル. In 言語処理学会第 *4* 回年次大会発表論文集, pages 209–212. 言語処理学会, 1998.

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

[5] Bob Carpenter and Gerald Penn. ALE 2.0 users' guide - the attribute-logic engine. Technical report, Computational Linguistics, Carnegie Mellon University, 1994.

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

[7] Takaki Makino. Implementation of an efficient feature structure abstract machine. A Senior Thesis, 1997. Submitted to the Department of Information Science, the Faculty of Science, the University of Tokyo.

[8] Yuka Tateisi, Kentaro Torisawa, Yusuke Miyao, and Jun'ichi Tsujii. Translating the XTAG English grammar to HPSG. In *Proc. TAG+4 Workshop*, pages 172–175, 1998.

[9] Kenji Nishida. An efficient HPSG parsing algorithm. Master's thesis, Department of Information Science, the Faculty of Science, the University of Tokyo, 1999.

[10] Yusuke Miyao, Kentaro Torisawa, Yuka Tateisi, and Jun'ichi Tsujii. Packing of feature structures for optimizing the HPSG-style grammar translated from TAG. In *Proc. TAG+4 Workshop*, pages 104–107, 1998.

[11] Takashi Ninomiya, Kentaro Torisawa, and Jun'ichi Tsujii. An efficient parallel substrate for typed feature structures on shared memory parallel machines. In *COLING-ACL '98*, pages 968–974, August 1998.

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

[13] Hassan Aït-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction.* The MIT Press, 1991.

[14] Ralph Clarke Haygood. Aquarius Prolog user manual. Technical Report TR-92-10, Swedish Institute of Computer Science, Box 1263, 164 28 Kista, Sweden, 1992.

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

[16] Shalom Wintner. *An Abstract Machine for Unification Grammars — with Application to an HPSG Grammar for Hebrew.* PhD thesis, the Technion – Israel Institute of Technology, 1997.

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

[18] Takaki Makino. LiLFeS user's guide. Available in `http://www.is.s.u-tokyo.ac.jp/~mak/lilfes/`, 1997. In Japanese.

[19] 吉田稔, 牧野貴樹, 鳥澤健太郎, and 辻井潤一. 素性構造処理言語 LiLFeS の最適化技術. In 言語処理学会第 *4* 回年次大会発表論文集, March 1998.

[20] Swedish Institute of Computer Science. *SICStus Prolog User's Manual, Release 3 #0*, June 1995.