Extension of a Feature Structure Abstract Machine for Partial Unification

NISHIDA Kenji*, MAKINO Takaki*
TORISAWA Kentaro, TATEISI Yuka, TSUJII Jun'ichi
Department of Information Science, University of Tokyo
Hongo 7-3-1, Tokyo 113, Japan
{nishiken,mak,torisawa,yucca,tsujii}@is.s.u-tokyo.ac.jp

Abstract

This paper describes an extension of a feature structure abstract machine for supporting partial unification, which is used in an efficient parsing algorithm for HPSG proposed by Torisawa et al. An abstract machine for attribute-value logics, a framework for processing typed feature structures proposed by Carpenter and Qu, is not only efficient but also easily extendable. We extended the abstract machine for partial unification by only adding a set of instructions to the machine. By combining this technique with the pre-computation of possible feature structures, efficiency of HPSG parsing is improved. We also show the feasibility of our implementation by a series of experiments.

1 Introduction

This paper describes an extension of an abstract machine for attribute-value logics[1] for supporting partial unification, which is used in a parsing algorithm for HPSG [2] proposed by Torisawa et al. [3]. In Torisawa's algorithm, HPSG is split into a CFG and the constraints which cannot be covered by the CFG. Parsing is done by a CFG parser and a specialized unification routine called *partial unification*. Partial unification is an operation that generates only the feature structure nodes that would be modified by ordinary unification, and ignores the nodes that are the same in an input feature structure and the result of unification.

An abstract machine for attribute-value logics (AMAVL), an efficient typed feature structure processing system proposed by Carpenter and Qu [1], is suitable for a use in such an HPSG parser because an AMAVL is easily extendable for supporting optimized routines such as partial unification. We demonstrate that partial unification is easily supported on an AMAVL by adding a new instruction set to it.

We first describe the overview of an AMAVL and partial unification in the next section. Section 3 describes our extension to an AMAVL for partial unification. Section 4 demonstrates its efficiency in two types of experiments. And Section 5 concludes this paper.

*The contributions of the first two authors are the same.

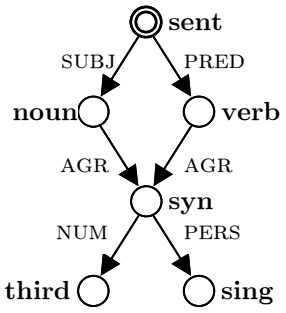
2 Background

This section describes the background of our work. First, an AMAVL, a framework for feature-structure processing systems, is described. Next, Torisawa's HPSG parsing algorithm and partial unification are described.

2.1 Typed Feature Structures

Typed feature structures (TFSs) provide a record-like data structure for representing partial information that can be expressed in terms of features and their values. Since their basic operation is unification, they are used for constraint-based formalisms such as HPSG.

A TFS is a rooted directed graph where all nodes and edges are labeled. A label on a node is called a *type*, and a label on an edge is called a *feature*. A *value* of a feature is a TFS whose root is the node the feature points to. A node that has more than two incoming edges is said to be *structure-shared*. Figure 1 (a) shows a sample of a TFS. SUBJ, PRED, and AGR are features, and **sent**, **noun**, and **verb** are types. The node labeled **syn** is structure-shared because there are two incoming AGR edges. A TFS can



(a) Graph notation

sent
SUBJ: noun
AGR: 1 syn
PER: [third]
NUM: [sing]
PRED: verb
AGR: 1

(b) AVM notation

Figure 1: Sample of a Typed Feature Structure (TFS)

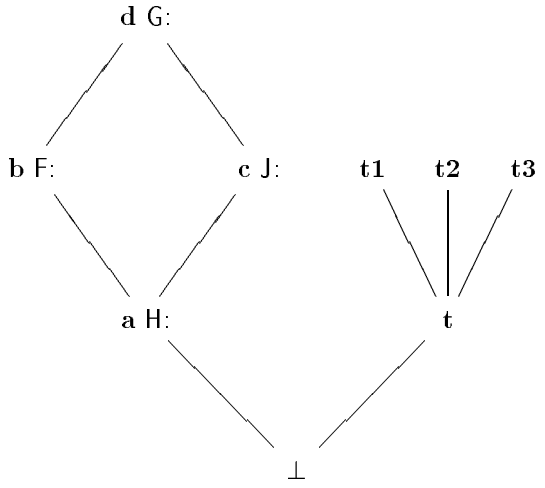


Figure 2: Sample type hierarchy

c H: 1 J: 1 t2 ⊔ b F: 2 H: 2 = d F: 3 G: ⊥ H: 3 J: 3 t2

Figure 3: A sample of Unification between TFSs

also be described in Attribute-Value Matrix (AVM) notation as in Figure 1 (b).

The types have a partial-order relation called *subsumption*. A subsumes B iff A is *more general* than B (or B is *more specific* than A). The set of relation forms a finite meet semilattice called *type hierarchy*. Figure 2 shows a sample hierarchy. The type also specifies the features the node labeled with it has. Two types connected by line shows that the lower one subsumes the upper one. We always assume the type $\bot$, called *bottom*, exists as the most general type of the type hierarchy.

Unification between two TFSs is defined. Intuitively, it calculates the new TFS which represents combination of information in the sense that it produces least upper bounds in the information ordering. Figure 3 shows a sample of unification operation. For the details of unification, see [4].

2.2 Abstract Machine for Attribute-Value Logics

This section describes an abstract machine for attribute-value logics (AMAVL), originally proposed by Carpenter and Qu [1], which is a framework that performs inference on logical formulae in attribute value logics, a logical framework describing TFSs. The logical formulae are called *descriptions* and they may include disjunctions, type constraints, and so on. However, for making the discussion simple, we here give a simplified description about AMAVL, where we focus on an aspect of AMAVL as the TFS inference engine. For the complete detail of AMAVL, refer to [1].

Figure 4 illustrates the unification operation on AMAVL. AMAVL has two types of internal representation for TFSs, namely, heap representation and code representation. Heap representation consists of data cells, and each cell corresponds to a node or an edge of the graph notation of a TFS. On the other hand, code representation consists of a sequence of abstract machine instructions, and each instruction corresponds to the constituent element of the TFS. Each time a code-represented TFS is executed, another TFS on the heap is modified so that the heap representation becomes the result of the unification of the old heap-represented TFS and the code-represented TFS. Unification between two heap-represented TFSs is also possible, but it is more efficient to perform unification using a code-representation, because the code-representation expresses operation in an unification directly[1].

In this section, we first describe the heap representation of TFSs, and after that, describe the code representation of TFSs and its execution process.

[1]Note that all unification cannot be performed by code-represented TFS. Structure-sharing included in a code-represented TFS requires unification between two heap-represented TFSs.

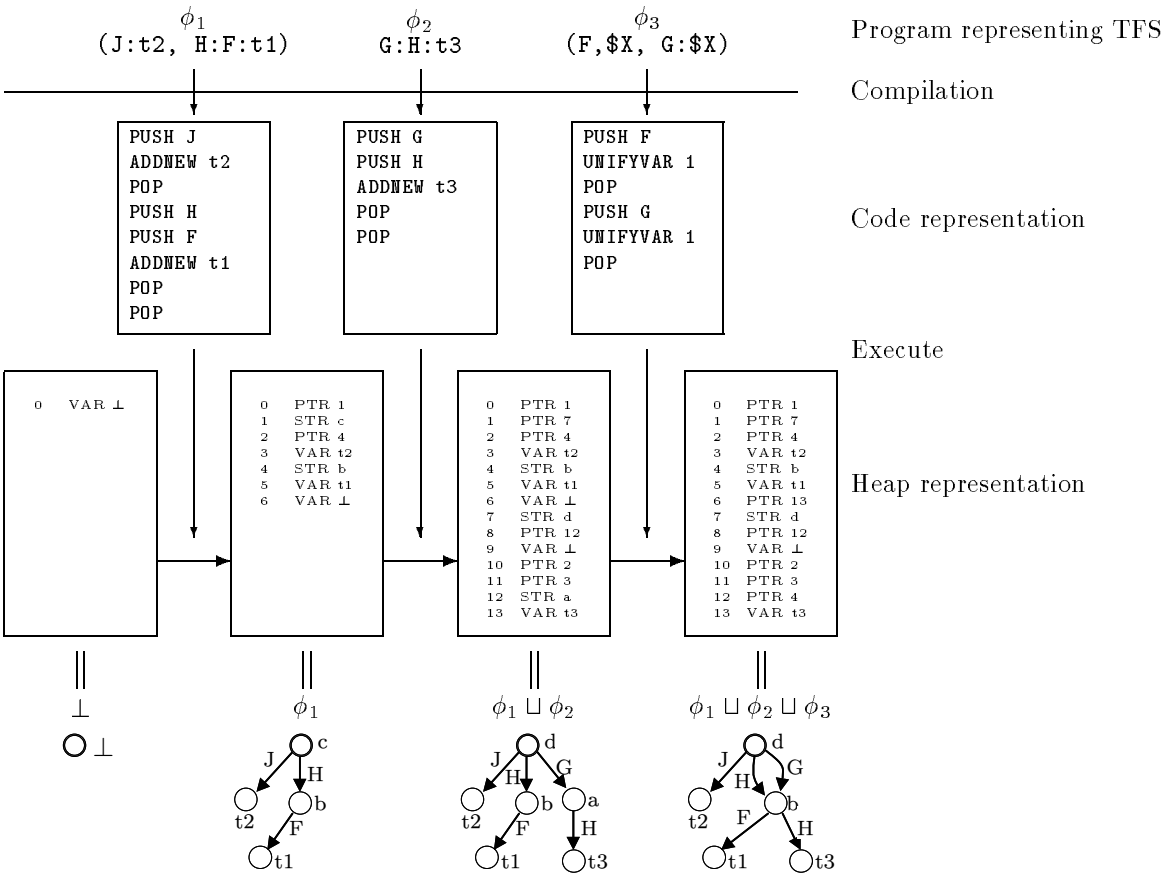


Figure 4: Overview of Data Flow in Unification Operation

2.3 Heap Representation of Typed Feature Structures

An AMAVL uses an array of data cells, called HEAP, to store TFSs. Each cell consists of a data value and a tag. There are three kinds of tags: An STR tag represents a node in the graph notation of a TFS and contains the type of the node; A PTR tag represents an edge in a graph and contains the address of the target node. The third tag, VAR, will be described later.

Since we require the feature structures to be totally well-typed[2], the number of features going out from a node can be considered constant for all feature structures with the same type. Thus, we store on the HEAP all the outgoing edges from a node in consecutive memory cells directly after the STR cell representing the type of the node. Given a type and a feature name, we can determine the offset of the edge corresponding to the feature at compile time. As a convention, we decided to sort the features within a type in alphabetical order[3].

Figure 5 shows an example of a representation on the HEAP with the type hierarchy shown in Figure 2. Its root node is at heap address 1 and has a type **d**. From the type hierarchy, the type **d** have 4 features, F, G, H, and J, corresponding to four cell at heap address 2, 3, 4, and 5, respectively. The value of the feature H starts at heap address 8, and the value of the feature H followed by H is at heap address 6, which is structure-shared to the value of the feature F.

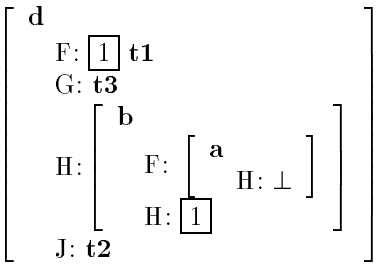
In addition to these tags, another tag named VAR is introduced. Since each feature of a feature structure has an appropriate type, the feature structures are recursively created and occupies many cells. To reduce memory consumption, we substitute a feature structure which is the most general TFS of a type t to one VAR cell, VAR t. Since, unlike an STR-represented TFS, a VAR-represented TFS occupies only one cell, VAR cells may be stored as arguments to an STR cell, in the same position where usually PTR cells are stored as representations of edges. Figure 6 shows an example of heap representation with

[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 detail, see [4].

[3] Note that this does not affect run-time efficiency, because the calculation of this part is done in type hierarchy compiler.

1	STR	**d**
2	PTR	6
3	PTR	7
4	PTR	8
5	PTR	11
6	STR	**t1**
7	STR	**t3**
8	STR	**b**
9	PTR	12
10	PTR	6
11	STR	**t2**
12	STR	**a**
13	PTR	14
14	STR	$\perp$

(a) Heap representation rooted at address 1



(b) AVM representation with the same TFS

Figure 5: Heap representation of a TFS

1	STR	**d**
2	PTR	6
3	PTR	7
4	PTR	8
4	PTR	11
6	STR	**t1**
7	STR	**t3**
8	STR	**b**
9	PTR	12
10	PTR	6
11	STR	**t2**
12	STR	**a**
13	PTR	14
14	STR	$\perp$

(a) Representation without VAR cells

1	STR	**d**
2	VAR	**t1**
3	VAR	**t3**
4	PTR	6
5	VAR	**t2**
6	STR	**b**
7	VAR	**a**
8	PTR	2

(b) Representation with VAR cells

(Correspond to the feature structure shown in Figure 5 (b))

Figure 6: Heap-representation with VAR Cells

	Program Element	Compiled code
Adding a type	t	`ADDNEW` t
Following a feature	f:P	`PUSH` f *Compiled code of* P `POP`
Structure sharing	$V	`UNIFYVAR` RegNumber($V)

(P, P_1, P_2 are some program elements of TFS)

Table 1: Basic Compilation Rule of a TFS in AMAVL

VAR cells. In addition to the substructures whose types are **t1**, **t2** and **t3**, which have no features (in other words, they are always 'most general'), the substructure of type **a** is also 'most general.' Consequently, four VAR cells are introduced, changing the TFS representation on the HEAP from Figure 6 (a) to Figure 6 (b). In this case the introduction of VAR tag reduced cell consumption from 14 cells to 6 cells.

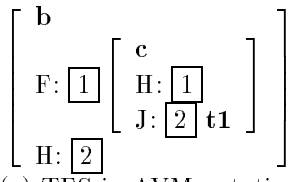
2.4 Compilation to Code Representations

Code representation of a TFS is a sequence of abstract machine instructions. Each instruction corresponds to a constituent element of a TFS. Table 1 shows the compilation rule from elements of a TFS to the compile code in an AMAVL. In this paper, we use the program notation in Figure 7(b) as a direct representation of a TFS.

In the following, the *current feature structure* CFS is a register pointing at the heap address to be operated. And an AMAVL also has a set of registers as a local memory besides the HEAP and the CFS.

Adding a type: A new type t in a TFS is compiled into an instruction `ADDNEW` t. This instruction performs a type coercion to the CFS so that its type is compatible to t. If the types are inconsistent, the instruction and the whole unification fails. If the type coercion increases the number of features of the CFS, and the CFS is represented by an STR cell, we must 1) allocate new cells to hold the result, which gets larger by the operation and cannot fit into the place of the original feature structure anymore, 2) for each feature which has corresponding old feature, store the pointer to it into the new feature, and 3) store into other features the VAR cells of the type specified by appropriateness.

Following a feature: A feature f in a TFS is compiled into a pair of instructions `PUSH` and `POP`. `PUSH` checks whether the feature structure



(a) TFS in AVM notation

```
(b,
  F: ($X, c,
       H: $X,
       J: ($Y, t1)),
  H: $Y)
```
(b) Sample Program same to (a)

Figure 7: Sample Program

pointed at by CFS contains the feature in its first level (for example, if the feature structure in Figure 7 (a) represents the structure pointed at by the CFS, the outermost occurrence of the features F and H constitute the first level of substructures). If not, the instruction tries to create the feature f by coercing the type with the base type of f, which may result in failure and backtracking. Thereafter, the value of the feature f that is now guaranteed to exist is pushed onto the top of the stack and the compiled description is performed recursively. When thereafter a `POP` instruction is encountered, the top value of the stack is deleted ('popped') and the recursion terminates, thus returning to the original CFS.

Structure sharing: A variable in a program, which works as a tag in AVM notation (See Figure 7), is compiled into a `UNIFYVAR` instruction. In an abstract machine, each variable is associated with a local register of the machine and the register's number is passed to `UNIFYVAR` as an argument. When the first `UNIFYVAR` with a certain variable is encountered, the pointer to the CFS is set to the register specified by the argument. Later encounters of a `UNIFYVAR` instruction with the same variable cause unification between the CFS and the feature structure pointed to by the register specified by the argument.

Figure 8 shows an example of code-representation of the TFS in Figure 7. Two variables, `$X` and `$Y`, are assigned to local register 1 and 2, respectively.

Since instructions described above involve complex operations which refer to type hierarchy and feature appropriateness, An AMAVL also provides a type hierarchy compiler which compiles type hierarchy and feature appropriateness to the operation of `ADDNEW`, `PUSH`, and `UNIFYVAR`. For the detail of compilation of the type hierarchy, see [1].

2.5 Torisawa's Parsing Algorithm and Partial Unification

The goal of HPSG parsing is to produce feature structures corresponding to the roots of parse trees. In

```
ADDNEW    b    ; (b,
PUSH      F    ;   F:(
UNIFYVAR  1    ;      $X,
ADDNEW    c    ;         c,
PUSH      H    ;       H:
UNIFYVAR  1    ;         $X
POP            ;           ,
PUSH      J    ;       J:(
UNIFYVAR  2    ;          $Y,
ADDNEW    t1   ;             t1
POP            ;               )
POP            ;                ),
PUSH      H    ;   H:
UNIFYVAR  2    ;     $Y
POP            ;       )
```

Figure 8: Example of Compiled Code

naively implemented HPSG parsers, the production would involve generation and unification of a large amount of feature structures. Torisawa's parsing algorithm reduces the amount of unification that must be performed at parsing time by 1) precomputing skeletal part of a parse tree and a large part of feature structures before parsing (*off-line raising*) and 2) avoiding the generation and the unification of precomputed feature structure at parsing time.

Off-line raising of feature structures takes advantage of the fact that, in HPSG, major parts of possible parse trees (their form and most of their features) are determined by a lexical head. For instance, given a lexical entry, head features of its possible ancestors can be determined before parsing. This is done by simply unifying a lexical entry to the head daughter of feature structure representing a grammar rule called a principle, and recursively applying the same procedure to the obtained mother. We call the trees obtained by off-line raising *pseudo parse trees* and the obtained feature structures **core-structures**.

Figure 9 shows the pseudo parse tree obtained from a lexical entry of a word "wrote." The core structures are head-feature in S_1 and S_2 and a part of the non-head daughters N_1 and N_2 determined by structure sharings (more precisely, the property NPness of N_1 and N_2 is determined from the SUBCAT value of the lexical entry "wrote" through structure sharings tagged as 1 and 2). The other part of feature structures such as a part of the SEM value, the AGENT value and the OBJECT value in S_1 and S_2 remains *unspecified* until the lexical entries for other words in the input are given.

At parsing time, the parser performs the following two phases.

Phase 1 : determine how to combine pseudo trees i.e., how non-head daughters of pseudo trees

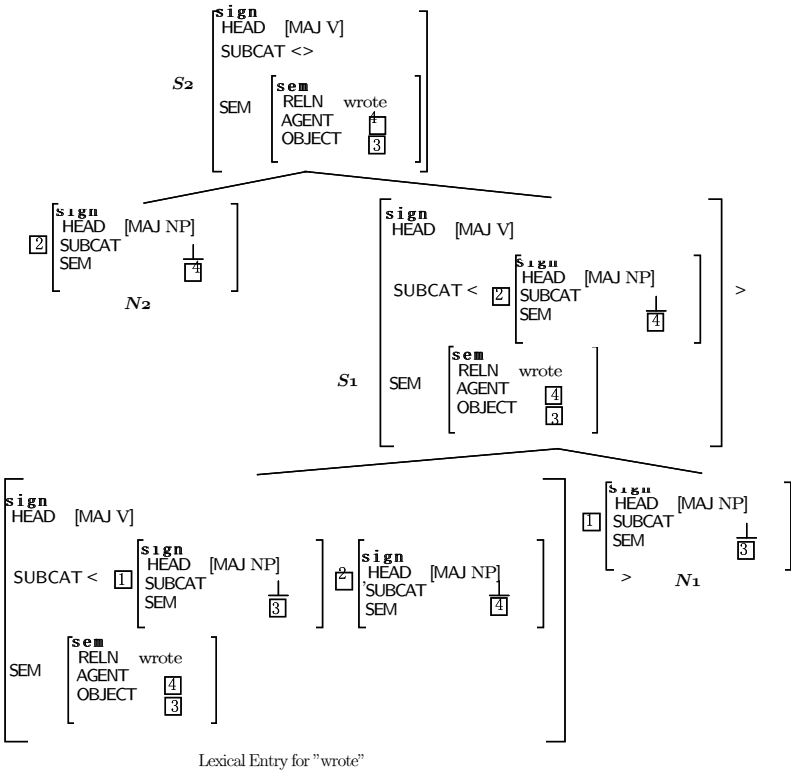


Figure 9: Example of Core Structure

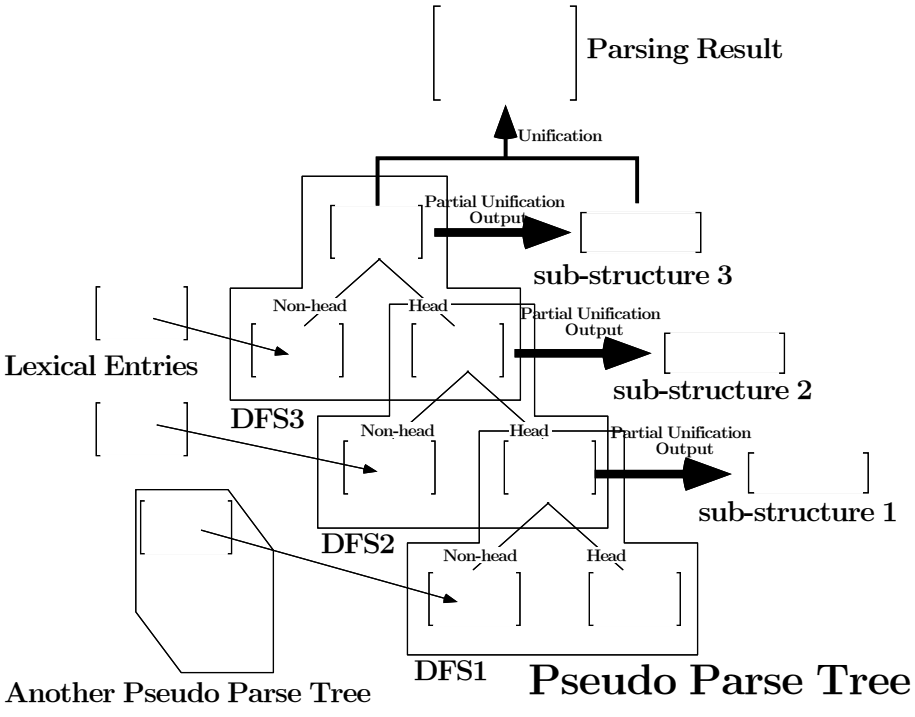


Figure 10: Rough Sketch of Phase 2 Computation

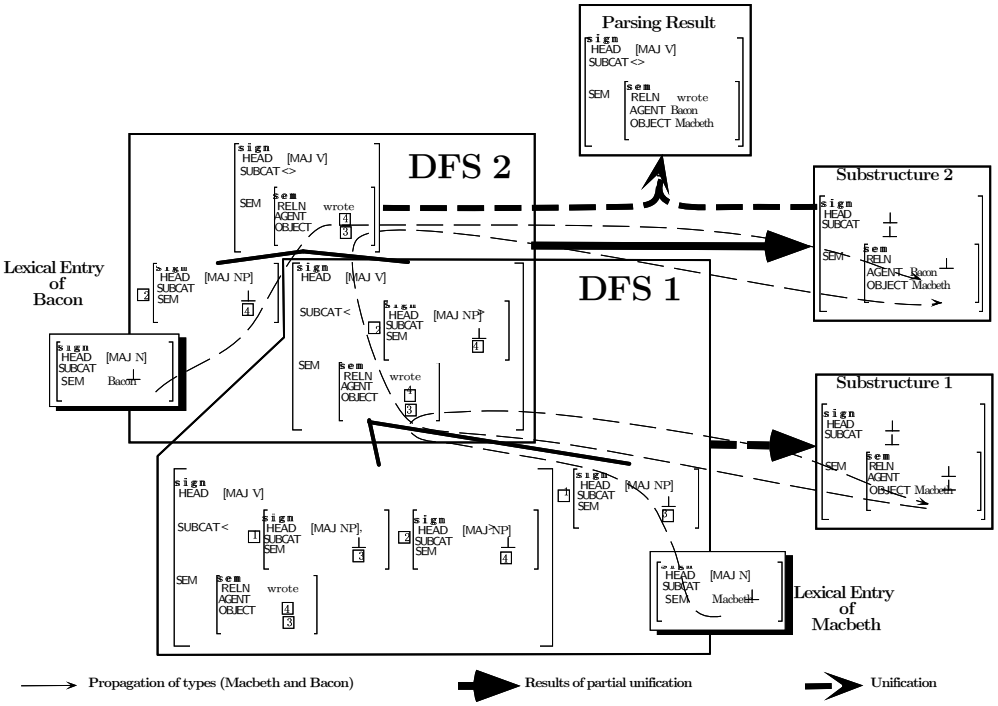


Figure 11: Examples of Partial Unification

should be unified with root nodes of other pseudo trees[4].

Phase 2 : compute *differential* feature structures which contain only the feature structure nodes modified during the combination of the trees. The modification occurs because a new information is given by another pseudo parse tree. We call this differential structures **sub-structures**.

Roughly, sub-structures correspond to the part of the feature structures that are *unspecified* in the core-structures. In the previous example of "wrote," the part of SEM values in S_1 and S_2 are in the sub-structures.

The key idea of Phase 2 is to compute only sub-structures, not the whole feature structures. At Phase 2, a parser traverses a pseudo parse tree from its lexical entry towards the root, with computing sub-structure corresponding to each node in the tree as sketched in Figure 10. When the traverse of the pseudo parse tree is finished, and a sub-structure corresponding to the root of the parse tree is computed, the core-structure of the root node and the computed sub-structure are unified. If the root of the pseudo parse tree is a root of a parse tree covering an input sentence, the unified result is a result of parsing. Otherwise, the unified result is used for computing a sub-structure for another pseudo parse tree as a non-head daughter.

Partial unification is a variant of unification used for computing sub-structures. Recall that the sub-structure consists only of the feature structure nodes in the mother of a non-head daughter that will be modified through structure sharings between the nodes when the daughter is unified with a root of another pseudo parse tree. Partial unification generates only such nodes but not others. Generating unmodified nodes in Phase 2 is redundant because they have already been computed at off-line raising. An example of computation at Phase 2 is given in Figure 11.

3 Extension of the Abstract Machine

We extended an AMAVL for partial unification. We first explain the requirements for the implementation of partial unification, and after that, describe the implementation.

[4] In [3], this operation was performed by *unifiability checking* which does not consume storage as much as unification. It is also possible to perform this by CFG compiled from pseudo parse trees.

3.1 Behavior of Partial Unification

The partial unification takes three inputs. The first is a triple of TFSs called a Dependency Feature Structures(DFS)[5]. It is a triple $\langle M, H, N \rangle$ where,

- M : a core-structure corresponding to a node in a pseudo parse tree,
- H : a core-structure corresponding to a head daughter of M.
- N : a core-structure corresponding to a non-head daughter of N.

Though they are treated as if they were *independent* feature structures, we assume that structure sharings between them are kept, in other words, M,H,N can contain identical feature structure nodes[6]. The second input, denoted by H', is a sub-structure corresponding to a head daughter H in the pseudo parse tree. When H is a lexical entry, H' is $\bot$. Otherwise H' is a result of partial unification in the previous steps. The third input, denoted by N', is a TFS representing a root of another pseudo parse tree. N' is obtained by unification of the core-structure representing the root and the sub-structures obtained at Phase 2 if the root is not a lexical entry. Otherwise, it is a lexical entry.

The goal of partial unification is to propagate only the modifications of TFSs, which are caused by H' and N', from the daughters H and N, to the mother M in the DFS. A partial unification routine compares H with H', and N with N' respectively and finds a node n with a type t in H or N satisfying one of the following conditions.

Cond 1. Where p is a path leading to n in H or N,

- n also appears in M,
- there exists a node n' such that it is reachable by p in H' or N' and its type is t', and
- $t \sqcup t' \neq t$.

Cond 2. Where P is a set of all the paths leading to n in H and N,

- n appears in M,
- there exists a node n' such that P' is a set of all the paths leading to n' in H' and N' and $P \cap P' \neq \emptyset$, and
- there exists a path $p_{\text{diff}} \in P'$ such that it leads to $n_{\text{diff}} (\neq n)$ in H or N and n_{diff} appears in M.

Cond 1 corresponds to modification of types. **Cond 2** corresponds to modification in configurations of structure sharings.

For the nodes satisfying the above condition in M, partial unification produces the node unified with its counterparts, which are all possible n' and n_{diff} in the above conditions. For the other nodes in M, partial unification generates a node and its features if and only if they have nodes satisfying the above condition as their descendants.

3.2 Implementing Partial Unification

We show our implementation of partial unification on an AMAVL in this section. Since the behavior of partial unification is quite different from that of ordinary unification, an AMAVL needs new instruction sets. Suppose that we performs partial unification of a DFS $\langle M, H, N \rangle$, a substructure of a head-daughter H', and a non-head-daughter N'. The DFS $\langle M, H, N \rangle$ is compiled to a sequence of the instructions newly added to the AMAVL. The added instructions are divided into two categories.

- Instructions corresponding to H and N (Daughter-Part): These instructions find nodes which satisfy **Cond 1** or **Cond 2**.
- Instructions corresponding to M (Mother-Part): These instructions produce the nodes found in Daughter-Part as a sub-structure of M.

In the following sections, we describe instruction sets for Daughter-Part and Mother-Part, respectively.

3.2.1 Compiling Daughter-Part

The basic behavior of the instructions for Daughter-Part is the same as that of ordinary unification; they traverse TFSs by PUSHs and POPs, and handle structure-sharings by UNIFYVARs. However, since the aim of Daughter-Part is to find nodes satisfying **Cond 1** or **Cond 2**, the instructions for Daughter-Part should avoid making unnecessary features in order to obtain the maximum efficiency.

To perform the task described above, some special fields were added to registers and HEAP cells in an original AMAVL for storing extra information. To a register, we added *propagation flag* for indicating if the node pointed by the register satisfies either of the conditions. We also need to assign *tag markers* to a node on the HEAP in order to identify n_{diff} in **Cond 2**. Tag markers are integers and can be seen as the tags representing structure sharings in AVM notations. For storing the tag markers, a special fields were added to STR cells and VAR cells.

The behavior of the new instructions are summarized as follows.

[5] To be exact, in [3], a DFS is a result of Dependency Analysis on the triple, and unnecessary TFSs are eliminated from the DFS. The difference does not make change on the other descriptions on this paper.

[6] In real implementation, the triple $\langle M, H, N \rangle$ is combined into one TFS as follows: DFS = $M \sqcup \begin{bmatrix} \text{H-DTR}:H \\ \text{NH-DTR}:N \end{bmatrix}$

- A `DPUSH` instruction works as a `PUSH` instruction does, except that when the TFS pointed by the register CFS does not have a feature specified by the argument of `DPUSH`, the instruction skips instructions until corresponding `DPOP` instruction appears, instead of creating the feature and continuing execution.

- A `DPOP` instruction works exactly as a `POP` instruction does.

- A `DUNIFYVAR` instruction corresponding to `UNIFYVAR` in normal unification is used at a node which has a possibility of satisfying **Cond 2**.

  - Set a tag specified by the argument of `DUNIFYVAR` to tag marker field s of all the nodes followable from the node pointed by the CFS.

  - If a node pointed by CFS has a tag marker, then the node can be n' of **Cond 2**. In this case, set a propagation flag to both the register associated with the tag specified as the argument of `UNIFYVAR` (corresponding to n) and the register associated with the tag in the tag marker field of the node pointed by the CFS.(corresponding to n_{diff}).

- A `DADDNEW` instruction is used at a node which has a possibility of satisfying the condition for n' in **Cond 1**. The node pointed by CFS can be n' in **Cond 1** If $t \sqcup t' \neq t$, where t is a type specified in the argument of `DADDNEW` and t' is a type of the node pointed by CFS. In this case, set a propagation flag to the register associated with the tag recorded in the tag marker field of the node pointed by CFS has. Note that `DADDNEW` is applied only to the nodes which are structure shared by several different paths and their descendants. The other nodes in DFSs cannot satisfy **Cond 1**, and normal `ADDNEW` is used in such a case for simply checking if types are consistent.

3.2.2 Compiling Mother-Part

The codes for Mother-Part produces nodes which satisfy either **Cond 1** or **Cond 2** in Daughter-Part as a sub-structure of M. In order to dispense with the creation of nodes which don't satisfy the conditions, we changed the behavior of the instructions for Mother-Part slightly different from that of ordinary unification; new features are created when a feature is restored from the stack, not when a feature is saved on the stack. In other words, our extension builds a TFS in a bottom-up manner in Mother-Part. By this change, creation of a TFS can be postponed until it is known whether the TFS satisfies the conditions. Thus an AMAVL can dispense with the creation of unnecessary features.

- An `MPUSH` instruction takes no argument, and `MPUSH` saves the address in CFS onto the stack and set the CFS to NULL in order to indicate there is no current TFS instead of creating a feature.Since `MPUSH` instructions are used for simply fixing the depth of a TFS, a feature is not specified. A feature may be specified when an `MPOP` instruction is executed.

- An `MUNIFYVAR` instruction is like `UNIFYVAR` but works only if `DUNIFYVAR` or `DADDNEW` has set a propagation flag on the register associated to the tag specified in the argument of the `MUNIFYVAR`. When CFS is set to NULL, `MUNIFYVAR` simply sets CFS to the pointer to the value of the register.

- An `MPOP` instruction creates the feature specified in the argument of the `MPOP` from the child node to the mother node unless CFS is set to NULL, where NULL means that the conditions are not satisfied and that creation of the TFS is not necessary.

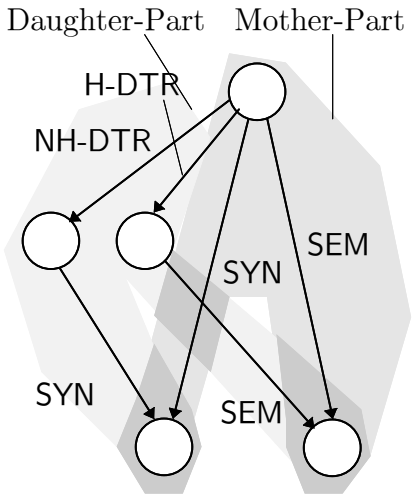
These instructions don't generate TFS nodes that don't satisfy neither of **Cond 1** and **Cond 2** in the result sub-structure of partial unification. On the other hand, if we use instructions for ordinary unifications for this purpose, the result sub-structure will include unnecessary TFSs. Because a sub-structure of partial unification will be duplicated as mother sign, the unnecessary TFS would be duplicated as many times as the height of the parse tree, thus the inefficiency would be multiplied.

3.3 An Example of Compilation

Figure 13 illustrates the execution of the instructions for partial unification in contrast to ordinary unification, when the compiled TFS Figure 12 (b) for Figure 12 (a) is executed on the TFS Figure 12 (c).

4 Experiments

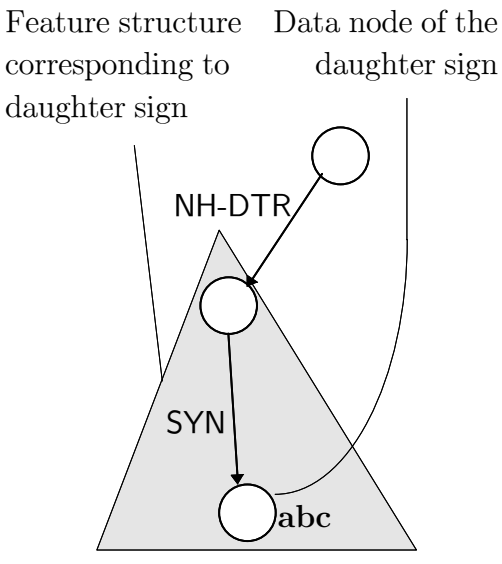
In this section, we provide a series of experiments to show the improvement from Torisawa's implementation to our implementation. It compares the processing time of the ordinary unification and that of partial unification, on an AMAVL and Torisawa's implementation. We also provide another series of experiments to show that basic performance of an AMAVL is competitive to other common feature-structure processing system.



(a) Sample DFS

(1)	DPUSH	NH-DTR
(2)	DPUSH	SYN
(3)	DUNIFYVAR	1
(4)	POP	
(5)	POP	
(6)	DPUSH	DTR
(7)	DPUSH	SEM
(8)	DUNIFYVAR	2
(9)	POP	
(10)	POP	
(11)	MPUSH	
(12)	MUNIFYVAR	1
(13)	MPOP	SYN
(14)	MPUSH	
(15)	MUNIFYVAR	2
(16)	MPOP	SEM

(b) Code for the DFS



(c) Sample daughter sign

Figure 12: Example of Dependency Feature Structure

4.1 Experiments on Partial Unification

The experiments are performed on our implementation of an AMAVL, named LiAM, and Torisawa's partial unification routine for his parser [3]. The experiments compare the time of ordinary and partial unification repeated 10000 times. The feature structure used in the experiments is a real data used in our grammar in HPSG framework, where the DFS has 50 nodes and the feature structure corresponding to daughter has 23 nodes.

Table 2 shows the results of the experiments. In these experiments, partial unification is about twice as fast as ordinary unification. These experiments also show that LiAM is more than 10 times faster than Torisawa's implementation. According to [3], Torisawa's implementation requires 3 seconds to parse a 20-word Japanese sentence. According to the speed improvement ratio from Torisawa's implementation to LiAM and the fact that an HPSG parser spends most of its unification time in unification, the HPSG parser realized on an AMAVL is expected to be efficient enough for practical use.

4.2 Experiments on Handling of Feature Structures

We measured efficiency of LiAM to estimate the basic performance of an AMAVL, using our programming language, named LiLFeS, built on LiAM. We prepared three benchmark programs. (1) Append Test: measuring speed of handling of small feature structures, (2) IU Test: measuring speed of unification of compiled feature structures, and (3) DU test: measuring speed of unification of uncompiled feature structures.

In addition to LiLFeS, we measured the speed on two feature structure interpreter. ALE [5] and Wild LIFE [6], and one feature structure compiler, ProFIT [7]. The result of the experiments shown in Table 3 suggest that LiAM itself is more efficient than other existing feature structure processing systems.

5 Conclusion

We have implemented LiAM, an efficient framework for feature structures, for partial unification. Partial unification is a specialized variant of unification which accomplishes high-efficiency for HPSG parsing systems. From experiments it appears that partial unification is efficiently performed on the LiAM as well as the LiAM itself is efficient enough.

Estimating from the comparison between LiAM and Torisawa's system, the parsing time of HPSG parsing system based on LiAM is expected to reach the level for practical use in natural language processing.

System\Test	Partial unification	Ordinary unification
LiAM	0.280	0.551
Torisawa's	3.666	7.954
Time ratio	13.10	14.44

(DEC Alpha 500/400MHz with 256M RAM, CPU time given in seconds)

Table 2: Comparing Calculation Time between AMAVL and Torisawa's implementation

System\Test	Append Test	IU Test	DU Test
LiLFeS	0.63	1.38	2.48
ProFIT[a]	1.24	2.84	6.57
ProFIT[b]	0.20	2.48	6.18
ALE	28.9	36.6	36.7
Wild LIFE	14.1	78.0[c]	78.0[c]

(DEC Alpha 600/333MHz with 256M RAM, CPU time given in seconds)

[a] Treating lists as feature structures, as LiLFeS and ALE do.
[b] Treating lists as Prolog built-in lists, making full use of list optimization on Prolog.
[c] Wild LIFE cannot complete the tests because of Out of Memory error.

Table 3: Comparing Calculation Time of Feature Structure Handling

References

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

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

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

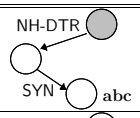
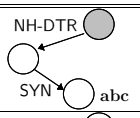
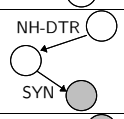
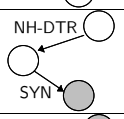
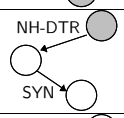
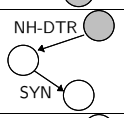
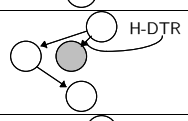
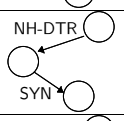
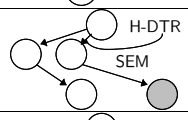
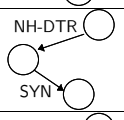
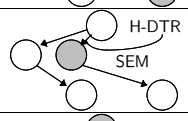
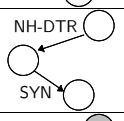
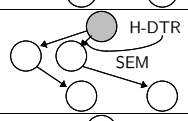
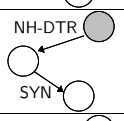
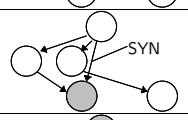
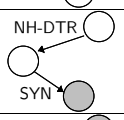
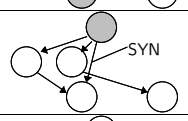
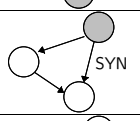
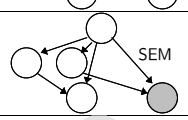
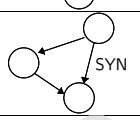
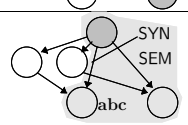
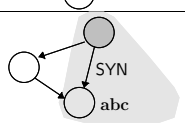
[4] Bob Carpenter. *The Logic of Typed Feature Structures*. Cambrige University Press, 1992. ISBN 0-521-41932-8.

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

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

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

[8] 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.

Normal Unification			Partial Unification		
Instructions	State of Heap	Description	Instructions	State of Heap	Description
		Initial state.			Initial state.
PUSH NH-DTR PUSH SYN		Follow features.	DPUSH NH-DTR DPUSH SYN		Follow existing features.
UNIFYVAR 1 POP POP		Save the pointer of CFS to the variable 1. And go back 2 levels.	DUNIFYVAR 1 POP POP		Save the pointer of CFS because there is a significant data **abc**. And go back 2 levels.
PUSH H-DTR		No H-DTR feature; make it and follow it	DPUSH H-DTR		No H-DTR feature; skip instructions until corresponding POP appears.
PUSH SEM UNIFYVAR 2		No SEM feature; make it and follow it Save the pointer to the variable 2.	DPUSH SEM DUNIFYVAR 2		Now skipping; Nothing done
POP		Go back one level.	POP		This POP is corresponding to PUSH SEM, not PUSH H-DTR. Continue skipping.
POP		Go back one level.	POP		Here is the corresponding POP. Go back to the last point.
PUSH SYN UNIFYVAR 1		PUSH creates SYN. UNIFYVAR associates it to previous 1.	MPUSH MUNIFYVAR 1		MPUSH clears CFS. MUNIFYVAR sets CFS to 1.
POP		The feature is already made, go back one level.	MPOP SYN		MPOP reversely makes the feature SYN while going back.
PUSH SEM UNIFYVAR 2		PUSH creates SEM. UNIFYVAR associates it to previous 2.	MPUSH MUNIFYVAR 2		MPUSH clears CFS. MUNIFYVAR do nothing because 2 is empty.
POP		Go back to top-level, complete.	MPOP SEM		This MPOP do not make the feature because CFS is empty. Go back to the previous point, complete.

(● indicates the place CFS is pointing to, ⬟ indicates the result substructure)
(some names of features and types are omitted)

Figure 13: Execution Step of Normal Unification and Partial Unification