# LiLFeS—Practical Unification-Based Programming System for Typed Feature Structures

**MAKINO Takaki, TORISAWA Kentaro, TSUJII Jun'ichi**
Department of Information Science, the University of Tokyo

## Abstract

This paper describes LiLFeS, an integrated unification-based programming system for linguistic formalisms based on typed feature structures, such as HPSG. The core engine of LiLFeS is an abstract machine developed for efficient handling of typed feature structures. Its basic design and optimization techniques are described. Performance comparisons between LiLFeS and other systems for typed feature structures show that LiLFeS is more than 50 times faster than ALE, and competitive to ProFIT.

## 1 Introduction

This paper describes LiLFeS, an integrated and efficient unification-based programming system for typed feature structures (TFSs), which is to be used as a core system for developing practical natural language processing systems. Although some systems for TFSs have already been developed, they still remain only to show computational soundness of grammar formalisms such as HPSG (Pollard and Sag, 1994), and are neither efficient nor flexible enough as a core language for practical NLP systems.

ALE (Carpenter and Penn, 1994) that was built on Prolog, for example, was not efficient enough to implement practical NLP systems, though it successfully showed computational treatment of TFSs. As theoretical soundness and computational tractability of TFSs and HPSG have been proven, more efficient systems based on TFSs, such as $\mathcal{AMALIA}$ (Wintner, 1997) and ProFIT (Erbach, 1994), have been implemented. While $\mathcal{AMALIA}$ is a parsing system that directly represents TFSs on an abstract machine, ProFIT compiles feature structures into Prolog terms and relies on a Prolog compiler for efficient execution.

However, since $\mathcal{AMALIA}$ cannot handle general definite clause programs, it can only show effectiveness of an abstract machine for TFSs but fails to be a core language for practical NLP systems, which require much richer functionalities such as those required for disambiguation, etc. While ProFIT provides the full capabilities of Prolog, it consumes a large amount of memory since it expands TFSs into Prolog terms.

In this paper, we discuss the design of an abstract machine which is an integration of the abstract machine for Prolog (Aït-Kaci, 1991) and the one for TFSs (Carpenter and Qu, 1995) and describe an actual implementation of a unification-based programming system based on TFSs, which we call LiLFeS.

LiLFeS has functionalities enough for building practical NLP systems. For example, LiLFeS incorporates variables, types, feature value restrictions and equations at the term level. Its syntax also allows disjunction, negation and cut, following the ordinary Prolog syntax. Built-in predicates such as copy of feature structures, assertion, lazy evaluation, multi-dimensional arrays, and `findall` predicates are provided. Definite clause programs are compiled into abstract machine instructions, which are then interpreted by an emulator written in C++. The definite clause compiler also performs last call optimization and argument indexing.

Moreover, since the abstract machine allows a user to implement new unification algorithms, one can devise, if necessary, a specialized version for his/her parsing algorithm. Actually, we showed that 1) a two-staged parsing algorithm for HPSG is much faster than conventional ones (Torisawa and Tsujii, 1996), and that 2) a new kind of unification called partial unification can be implemented efficiently on the abstract machine (Nishida, 1997).

Section 2 shows an overview of LiLFeS. In Section 3, we describe extensions and optimization

```
% 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

```
        |~nelist                          ~|
        |  FIRST:foo                       |
        |        |~nelist              ~|  |
RESULT: |        |  FIRST:bar            |  |
        |  REST:|        |~nelist     ~|  |  |
        |       |  REST:|  FIRST:zot  |  |  |
        |_      |_      |_ REST:elist_|_|_|
```

(b) Result of (a)

Figure 1: LiLFeS Program Sample

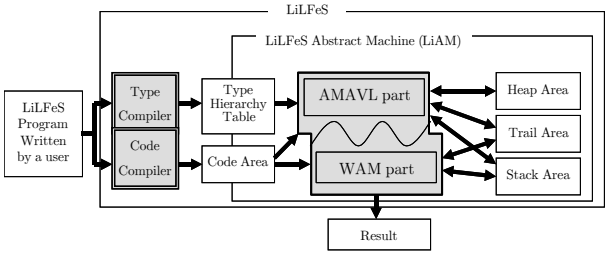



Figure 2: LiLFeS Framework Overview

techniques we incorporated in LiLFeS. Section 4 shows performance comparisons with other systems. Finally, we conclude the paper and describe further research directions.

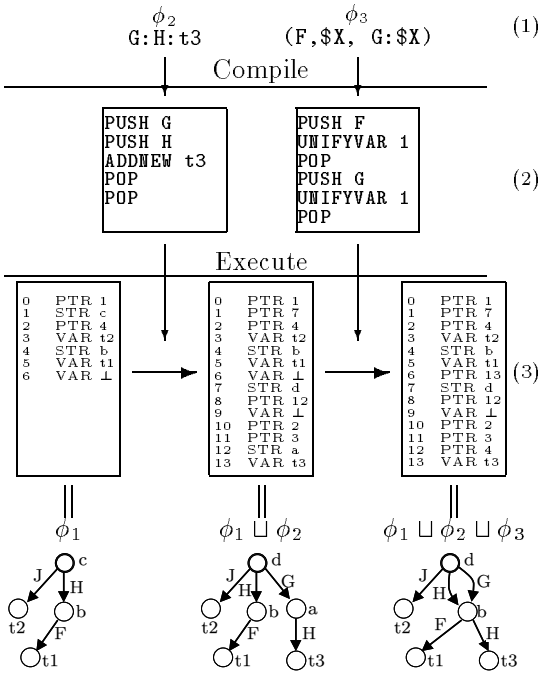
## 2 Overview

LiLFeS is a unification-based programming language where the terms are typed feature structures (TFSs). This section provides an overview of the functionality of LiLFeS.

### 2.1 *LiLFeS, a unification-based programming language*

LiLFeS is a language that we developed for Typed Feature Structures (TFSs). A program of LiLFeS consists of a set of definite clauses (which is similar to Prolog) and type definitions. Figure 1 is an example LiLFeS program. For details, refer to (Makino, 1997).

The structure of the LiLFeS system is illustrated by Figure 2. Type definitions of a LiLFeS program are compiled into a type hierarchy table,




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

Figure 3: Overview of Unification Operation

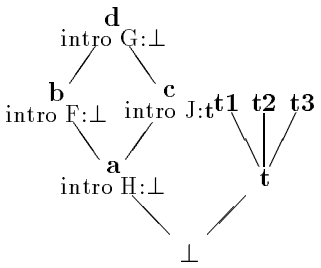
and definite clauses are compiled into a sequence of instruction codes. Then *LiAM*, the LiLFeS Abstract Machine, executes the compiled codes and returns the results.

Basically, LiAM consists of two parts: the TFS unification engine and the definite-clause inference engine. We used the framework of *AMAVL* (abstract machine for attribute-value logics) for the TFS engine, and the framework of *WAM* (Warren's Abstract Machine) for the inference engine, and integrated them into LiAM. We describe the framework of AMAVL in the next section, and describe the integration in Section 3.

### 2.2 *Abstract Machine for Attribute-Value Logics*

An abstract machine for attribute-value logics (AMAVL) (Carpenter and Qu, 1995), archetype of LiAM, is a framework that performs inference on logical formulae in attribute value logics, a logical framework describing TFSs. Logical formulae are called *descriptions* and they may include disjunctions, type constraints, and so on. However, for easier understanding, we give here a simplified explanation of AMAVL, where we focus on an aspect of AMAVL as the *direct* TFS manipulation engine. For the complete details of AMAVL, refer to (Carpenter and Qu, 1995).

Figure 3 illustrates the unification operation in AMAVL. AMAVL has two types of internal representation for TFSs: heap representation and code

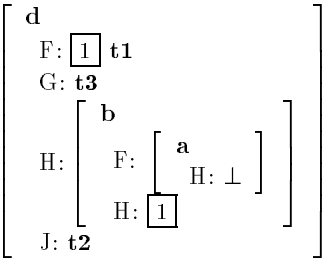



(The notation "intro $f$:$\tau$" under the type $\sigma$ describes that Approp($\sigma, f$) = $\tau$)

Figure 4: Sample type hierarchy

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

(a) Heap representation




(b) AVM representation

Figure 5: Heap representation of a TFS

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 new 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 would also be possible, but it is more efficient to perform unification using code representation, because the code-representation expresses the inherent operation of an unification directly.

In the following, we describe both of heap representation and code representation, respectively.

### *2.3 Heap Representation of Typed Feature Structures*

AMAVL uses an array of data cells, called a heap, to store typed feature structures. Each cell consists of a data value and a tag. There are three kinds of tags: A PTR tag represents an edge in the graph notation and contains the address of the target node; An STR tag represents a node in the graph of a TFS and contains the type of the node; A VAR tag represents a leaf node in a graph[1].

Since we require feature structures to be totally well-typed[2], the number of features going out from a node is considered constant for all feature structures of the same type. Thus, as in WAM, we store on a heap all the outgoing edges from a node in consecutive memory cells directly after the STR cell representing 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 sort the features within a type in alphabetical order[3].

Figure 5 shows an example of a representation on a heap with the type hierarchy shown in Figure 4. Its root node is at heap address 1 and has type **d**. From the type hierarchy, it can be seen that type **d** has 4 features: F, G, H, and J. Thus they correspond to four cells at heap addresses 2, 3, 4 and 5, respectively. The entry for the feature H starts at heap address 6, and the entry for the path H|H is at heap address 2, which is structure-shared with the value of the feature F.

### *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[4].

In the following, a global register CFS is used to point to the *current feature structure*.

**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 fails and triggers backtrack. If the type coercion increases the number of features of the CFS, and the CFS is not a leaf node, new cells are allocated to hold the result and old cells are redirected to the new cells.

**Following a feature** A feature $f$ in a TFS is compiled into a pair of instructions `PUSH` and `POP`. `PUSH` checks whether the node of the feature structure pointed at by CFS contains the feature. If

[1] An STR tag can also represent a leaf node, but this is inefficient, because they may contain unnecessary features

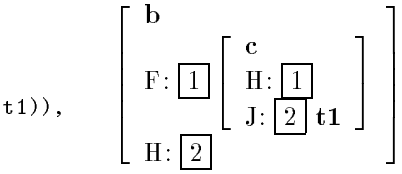
[2] A feature structure is *totally well-typed* iff for all node $q$, the feature $f$ from $q$ exists for all features $f$ that is appropriate to the type of $q$, and does not exist for all feature $f$ that is not defined as appropriate to the type of $q$, and the content of any feature has an appropriate type. For details, see (Carpenter, 1992).

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

[4] As described in Section 2.2, we treat in this paper AMAVL code as a direct representation of TFSs which does not include disjunction. However, according to (Carpenter and Qu, 1995), a TFS description part in AMAVL can contain logical devices, such as disjunctions, directly in TFS description inside a clause without representing them by relying on definite clause.

```
(b,
  F: ($X, c,
      H: $X,
      J: ($Y, t1)),
  H: $Y)
```

(a) Sample Program




(b) AVM Notation of (a)

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

(c) Compiled codes

Figure 6: Sample Compilation of AMAVL

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 address of the CFS is pushed onto the stack and the CFS is set to the value of feature $f$ that is now guaranteed to exist. The compiled description is performed recursively, and when `POP` is encountered, the stack top is popped and the original CFS is recovered.

**Structure sharing** A variable in a program, which represents a structure sharing denoted by a number like 1, is compiled into a `UNIFYVAR` instruction. In the abstract machine, each variable is associated with a local register of the machine. When the first `UNIFYVAR` with a certain variable is encountered, the address of the CFS is stored in the register associated with the variable. Later encounters of `UNIFYVAR` with the same variable cause unification between the CFS and the feature structure addressed by the register associated with the variable.

An example of code representation of a TFS is in Figure 6. Two variables, `$X` and `$Y`, are assigned to local register 1 and 2, respectively.

Since instructions described above involve complex operations that refer to the type hierarchy and feature appropriateness, AMAVL also provides a type hierarchy compiler which compiles type hierarchy and feature appropriateness into the operation of `ADDNEW`, `PUSH`, and `UNIFYVAR`. For details of compilation of the type hierarchy, see (Carpenter and Qu, 1995).

## 3 Integrated Features in LiLFeS

We have integrated AMAVL and WAM into LiAM, the LiLFeS abstract machine, and added various extensions and optimization techniques to improve efficiency. AMAVL is a powerful framework for unification of typed feature structures. Practical NLP systems, however, requires more complex operations than simple unifications. For that purpose, we provide the ability to handle Prolog-like clause by integrating WAM's clause control and AMAVL's TFS handling.

In this section we also explain the built-in predicates introduced in LiLFeS. These predicates are useful to construct a practical parsing system.

### *3.1 Integration with WAM*

Warren's Abstract Machine (Aït-Kaci, 1991) is a well-known framework for processing Prolog terms and Horn-clauses. AMAVL defines processing of terms (TFSs) only, and it doesn't define processing of clauses. We have integrated AMAVL's term processing and WAM's clause processing into a new abstract machine LiAM[5].

Table 1 is a list of design features which are ported from WAM to LiAM. With these design features LiLFeS is able to handle definite clauses, backtracking and cuts.

| |
|---|
| **Calling convention of clauses** |
| Instructions for clause control (`call` etc.) |
| Argument passing conventions |
| Global register for continuation pointer |
| **Environment stack frame** |
| Global register for environment frames |
| Allocation of environment frames (`allocate` etc.) |
| **Backtracking of clauses** |
| Global registers for backtracking |
| Instructions for backtracking control (`try` etc.) |
| Instructions for cut operations |
| **Optimization techniques** |
| Last call optimization |
| Argument indexing |

Table 1: Integrated features from WAM to LiAM

To make these features of WAM consistent with AMAVL, we changed some instructions of AMAVL. AMAVL's instructions as described in Section 2.2 assume that variables are properly initialized, while instructions in WAM involve initializations in its unification code. Thus we changed AMAVL instructions to involve initialization. Firstly, we divided the `UNIFYVAR` instruction into two instructions. `LINKVAR` is used for the

[5] Due to lack of space we omit detailing WAM here. For more details, refer to (Aït-Kaci, 1991).

```
ALLOC      1     ; p(
UNIFY1ST   A1    ;   A,
LINKVAR    X3    ;
UNIFY1ST   A2    ;     B)
LINKVAR    Y1    ;        :-
LINK1ST    A1    ; q(
PUSH       F     ;   F:
UNIFYVAR   A3    ;     A
POP              ;      ,
FOLLOW     G     ;       G:
UNIFYVAR   Y1    ;         B
CALL       q/1   ;          ),
LINK1ST    A1    ; r(B
UNIFYVAR   Y1    ;
DEALLOC          ;
EXECUTE    r/1   ;    ).
```

Figure 7: Sample Compiled Code

first occurrence of a certain variable, and initializes the associated register. On the other hand, new `UNIFYVAR` is used for later occurrences of the same variable, and perform unification as the old one. The unification compiler chooses properly between these two instructions[6]. This separation also accelerates `UNIFYVAR` because it no longer requires the first-occur check of a variable. Secondly, we created a group of instructions that initializes the current feature structure (CFS) pointer. Four instructions refer to the CFS: `LINKVAR`, `UNIFYVAR`, `ADDNEW` and `PUSH`. We introduced corresponding instructions that regard the CFS as being used for the first time: `LINK1ST`, `UNIFY1ST`, `ADDNEW1ST`, and `PUSH1ST`, respectively.

We also optimized instruction sequences by introducing a `FOLLOW` instruction[7]. `FOLLOW` is a variant of the `PUSH` instruction that does not push CFS onto the stack. Since many `PUSH`ed pointers are disposed of without having been used, we can reduce such inefficiency by this new `FOLLOW` instruction.

Figure 7 shows an example of compiled code using this integrated instruction set. In the code, Variable `A` is allocated to a local register X3, and variable `B` is allocated to a stack variable Y1. `ALLOC`, `DEALLOC`, `CALL`, and `EXECUTE` instructions in the code work as their counterparts in WAM.

### *3.2 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. For this reason, we provided a set of built-in predicates useful for such programs:

- Copying of feature structures
- Equivalence checking
- Token Identification
- Asserting a clause
- Lazy evaluation
- Array operations

Usage of these predicates improves efficiency of NLP systems. For example, suppose that we are about to build a bottom-up chart parser. The program stores calculated edges into a two-dimension array, which is efficiently represented. Calculated edges may be spurious, that is, already calculated and stored; equivalence checks are performed to check out spurious edges. The calculated edge may be used several times, thus copy predicates are useful to avoid side effects, since backtracking is inefficient when the parsing paths become long. Although some of these predicates are theoretically 'dirty' and violate bi-directionality for instance, they are helpful in constructing practical programs.

## 4 Performance Comparisons

We made performance comparisons between LiLFeS and conventional parsers with typed feature structures to demonstrate the efficiency of LiLFeS. Tested programs are LiLFeS, ALE 2.01, and ProFIT 1.54. Table 2 shows the result of our experiments. LiLFeS is more than 50 times faster than ALE, and quite competitive to ProFIT.

LiLFeS is getting slower in complex syntax because a large number of types increases cache misses, while ProFIT was not so affected by the number of types. The reason of the slow-down is that, to resolve type inheritance information, LiLFeS regularly refers a type hierarchy table, whose size is $O(n^2)$, where $n$ is the number of types.

On the other hand, LiLFeS is getting faster with the bottom-up chart parser. This is because LiLFeS has efficient built-in predicates that are useful to the bottom-up chart parser. In some comparisons LiLFeS is even faster than ProFIT with SICStus native code compiler.

Note that LiLFeS does not achieve the final goal in this test. We are currently developing built-in predicates for an efficient CKY parser and partial

[6] (Carpenter and Qu, 1995) says that the variable first-occur check cannot be done at compile time due to disjunctions of descriptions that are realized by backtracking. But although it is easy to implement the first-occur-check embedding in the case that a TFS contains no disjunction, the first-occur-check embedding can be done even if disjunctions are allowed. In this case, the basic idea is to define three types of instructions: for the first occurrence, for later occurrences, and for undecidable cases.

[7] This optimization was suggested by Carpenter and Qu (Carpenter and Qu, 1995).

| | | Simple bottom-up | | Chart parser | | |
|---|---|---|---|---|---|---|
| | | ($\alpha$) | ($\beta$) | ($\alpha$) | ($\beta$) | ($\gamma$) |
| # sentences parsed per second* on DEC Alpha† | LiLFeS | 347.22 | 4.63 | 315.36 | 15.64 | 7.229 |
| | ProFIT[a] | 425.77 | 7.12 | 314.36 | 7.01 | 24.90 |
| | ALE[a] | 5.902 | 0.126 | Not Tested | Not Tested | Not Tested |
| # sentences parsed per second* on Sun UltraSparc‡ | LiLFeS | 217.10 | 1.61 | 91.66 | 6.42 | 1.05 |
| | ProFIT[a] | 342.86 | 7.86 | 236.59 | 5.15 | 19.79 |
| | ProFIT[b] | 677.39 | 15.39 | 290.38 | 5.69 | 16.70 |

($\alpha$) Simple HPSG grammar, which has 3 principles, 3 id-schemata, and 10 lexical entries. The grammar includes 60 types. Input is one 9-word sentence and output is 1 solution. ($\beta$) Same grammar with ($\alpha$), but the grammar is artificially changed to output 64 solutions. ($\gamma$) Rather complex HPSG grammar, which has 5 principles, 7 id-schemata, and 102 lexical entries. The grammar includes 60 types. Input is one 11-word sentence and output is 1 solution.
*a* With SICStus-vanilla (WAM interpreter) *b* With SICStus-fastcode (native code compiler), not provided for Alpha
† DEC Alpha 500/400MHz with 256M memory ‡ Sun UltraSparc 1/167MHz with 128M memory on Solaris
∗ Measured by user CPU time

Table 2: Comparison between LiLFeS and Other Systems

unification, which is necessary for Torisawa's parsing algorithm (Torisawa and Tsujii, 1996). With this implementation we expect to drastically improve parsing speed over the current parser, which is written in definite clauses.

## 5 Conclusion and Future Work

We described LiLFeS, an integrated unification-based programming system with typed feature structures. The core of the system is an abstract machine that is the product of integrating Warren's Abstract Machine and Carpenter's Abstract Machine for Attribute-Value Logics.

Future work will be enhancement of LiLFeS with other optimization techniques. Our research (Nishida, 1997) shows that a technique of partial unification (Torisawa and Tsujii, 1996) can improve parsing performance up to 250% when combined with an abstract machine and HPSG grammar formalism. Compression of a type hierarchy table will decrease cache misses and improve efficiency. Furthermore, Integration with the technique of Aquarius Prolog (Van Roy, 1990) would accelerate LiLFeS. Aquarius Prolog compiles Prolog into native machine codes, and achieves drastic speedup. The same is possible with LiLFeS, and we expect great speedup for this.

## References


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

Bob Carpenter and Gerald Penn. 1994. ALE 2.0 user's guide. Technical report, Carnegie Mellon University Laboratory for Computational Linguistics, Pittsburgh, PA.

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

Bob Carpenter. 1992. *The Logic of Typed Feature Structures.* Cambridge University Press, Cambridge, England.

Gregor Erbach, 1994. *ProFIT 1.05 user's guide.* Computerlinguistik, Universitaet des Saarlandes.

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

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

Carl Pollard and Ivan Sag. 1994. *Head-Driven Phrase Structure Grammar.* University of Chicago Press, Chicago.

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

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

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