Interleaving Universal Principles And Relational Constraints Over Typed Feature Logic

Thilo Götz and **Detmar Meurers**
SFB 340, Universität Tübingen, Kleine Wilhelmstraße 113,
72074 Tübingen, Germany.
`{tg,dm}@sfs.nphil.uni-tuebingen.de`

Abstract

We introduce a typed feature logic system providing both universal implicational principles as well as definite clauses over feature terms. We show that such an architecture supports a modular encoding of linguistic theories and allows for a compact representation using underspecification. The system is fully implemented and has been used as a workbench to develop and test large HPSG grammars. The techniques described in this paper are not restricted to a specific implementation, but could be added to many current feature-based grammar development systems.

Introduction

A significant part of the development of formalisms for computational linguistics has been concerned with finding the appropriate data structures to model the linguistic entities. The first order terms of Prolog and DCGs were replaced by feature structures in PATR style systems,[1] and in recent years systems using typed feature structures have been developed.

Following the tradition of DCG and PATR, these typed feature systems are generally definite clause based, e.g., CUF (Dörre and Dorna 1993), or phrase structure based, e.g., ALE (Carpenter and Penn 1994). Instead of permitting the grammar writer to express universal well-formedness constraints directly, the systems require the grammar writer to express relational constraints and attach them locally at the appropriate places in the grammar.[2]

We believe there are several reasons why the advances in the linguistic data structures should entail the development of systems offering more expressive means for designing grammars. Using universal implicative constraints, or universal principles as they are usually called in the linguistic literature, grammatical generalisations can be expressed in a more compact and modular way . Another advantage of an architecture including principles is that it computationally realizes the architecture assumed in Pollard and Sag (1994) for HPSG. It thus becomes possible to develop and test HPSG grammars in a computational system without having to recode them as phrase structure or definite clause grammars. The architecture can also serve as extended architecture for principle based parsing (e.g., Stabler and Johnson 1993) since it facilitates the implementation of GB-style universal principles. Offering both more perspicuous grammar code and closeness to linguistic theory, it seems well motivated to explore an architecture which allows both relational constraints and universal restrictions to be expressed.

Our implementation is based on the idea to compile implicational constraints into a relational representation (Götz and Meurers 1995) where calls to the constraint solver are made explicit. This allows for an integration of implicational and relational constraints and a uniform evaluation strategy. Efficient processing is achieved through user-supplied delay patterns that work on both relations and implicational constraints, as well as preferred execution of deterministic goals at run-time.

The paper is organised as follows. We will start out by illustrating our architecture with an example. We then go on to describe the key aspects of the implementation. We compare our work to other approaches before presenting some conclusions and open issues in the last section.

Motivating the architecture

Consider the Head Feature Principle (HFP) of Pollard and Sag (1994) as an introductory example for a grammatical principle. The HFP requires that in a headed construction the head features of the mother are identified with the head features of the head daughter. In a

[1] Cf. Shieber (1986) for a discussion of these formalisms.

[2] ALE has a restricted form of universal constraints, see the comparison section.

typed feature logic[3] this may be expressed by the principle shown in Fig. 1.

$$\begin{aligned} & \mathit{phrase} \wedge \mathit{dtrs} : \mathit{headed\text{-}struc} \rightarrow \\ & \quad \mathit{synsem} : \mathit{loc} : \mathit{cat} : \mathit{head} : X \wedge \\ & \quad \mathit{dtrs} : \mathit{head\text{-}dtr} : \mathit{synsem} : \mathit{loc} : \mathit{cat} : \mathit{head} : X \end{aligned}$$

Figure 1: A Head-Feature Principle

In CUF, we can encode the HFP as a clause defining a unary relation *hfp* as shown in Fig. 2.[4]

$$\begin{aligned} \mathit{hfp} \;:=\; & \mathit{synsem} : \mathit{loc} : \mathit{cat} : \mathit{head} : X \wedge \\ & \mathit{dtrs} : \mathit{head\text{-}dtr} : \mathit{synsem} : \mathit{loc} : \mathit{cat} : \mathit{head} : X \end{aligned}$$

Figure 2: A relation encoding the HFP

For the relation *hfp* to take effect, calls to it need to be attached at the appropriate places. Expressing grammatical constraints in such a way is both time consuming and error prone.

Suppose we want to define the unary relation *wf-phrase* to hold of all grammatical phrases. In case all grammatical phrases are constrained by a term ϕ and some relation P, we can define the relation *wf-phrase* shown in Fig. 3.

$$\mathit{wf\text{-}phrase} := \mathit{phrase} \wedge \phi \wedge P$$

Figure 3: Defining the relation *wf-phrase*

To specify that $\phi \wedge P$ holds for all phrases while the HFP only holds for headed phrases, we now have to manually split the definition into two clauses, the subcase we want to attach the HFP to and the other one.

This is both inelegant and, barring a clever indexing scheme, inefficient. Using universal principles on the other hand, the global grammar organisation does not need to account for every possible distinction. The organisation of the data structure as typed feature structures already provides the necessary structure and the grammatical constraints only need to enforce additional constraints on the relevant subsets. Thus, the implica-

$$\begin{aligned} \mathit{wf\text{-}phrase} \;:=\; & \mathit{phrase} \wedge \mathit{dtrs} : \mathit{headed\text{-}struc} \\ & \wedge\, \mathit{hfp} \wedge\, \phi \wedge P \\ \mathit{wf\text{-}phrase} \;:=\; & \mathit{phrase} \wedge \mathit{dtrs} : \neg \mathit{headed\text{-}struc} \\ & \wedge\, \phi \wedge P \end{aligned}$$

Figure 4: Splitting up the wf-phrase relation to accommodate the HFP call

tional constraint encoding the HFP shown in Fig. 1 constrains only the headed phrases, and the non-headed ones do not need to be considered.

Finally, a system providing both universal principles and relational constraints at the same level offers a large degree of flexibility. While in HPSG theories the principles usually form the main part of the grammar and relations such as *append* are used as auxiliary constraints called by the principles, a more traditional kind of grammar for which one prefers a relational organisation can also be expressed more compactly by adding some universal principles constraining the arguments of the relations to the relational core. Both kinds of interaction are possible in the non-layered architecture we propose.

With respect to our example, the first kind of interaction would be obtained by also expressing the general restriction on *phrase* as a universal constraint as shown in Fig. 5, while the more traditional kind of grammar

$$\mathit{phrase} \;\rightarrow\; \phi \wedge P$$

Figure 5: A universal constraint on phrases

would keep the relation defining well-formed phrases shown in Fig. 3 and combine it with the universal constraint of Fig. 1 in order to avoid splitting up the relation as was shown in Fig. 4. The proper interaction of relational and universal constraints then needs to be taken care of by the system.

An example grammar

To further motivate our approach, we now show how to code a simple principle based grammar in our framework. Figure 6 shows the inheritance hierarchy of types which we will use for our example with the appropriateness conditions attached.

Our example grammar consists of some universal principles, phrase structure rules and a lexicon. The lexicon and phrase structure rules are encoded in the *wfs* (well-formed sign) relation shown in Fig. 7 and the implicational principles in Fig. 8. The *wfs* predicate takes three arguments: a difference list pair threading the string through the tree in the style of DCGs, and

[3]Following King (1989) and Carpenter (1992), we use a typed feature logic with appropriateness restrictions for the domains and ranges of features. For space reasons we cannot provide a formal definition of the logic here, but the interested reader is referred to Carpenter (1992) for an exposition.

[4]Throughout the paper and as syntax for the system discussed here we use the *functional style* notation of CUF (Dörre and Dorna 1993) for our relations, where a designated result argument is made explicit. The denotation of a relation thus is a set of objects just like the denotation of any other feature term.

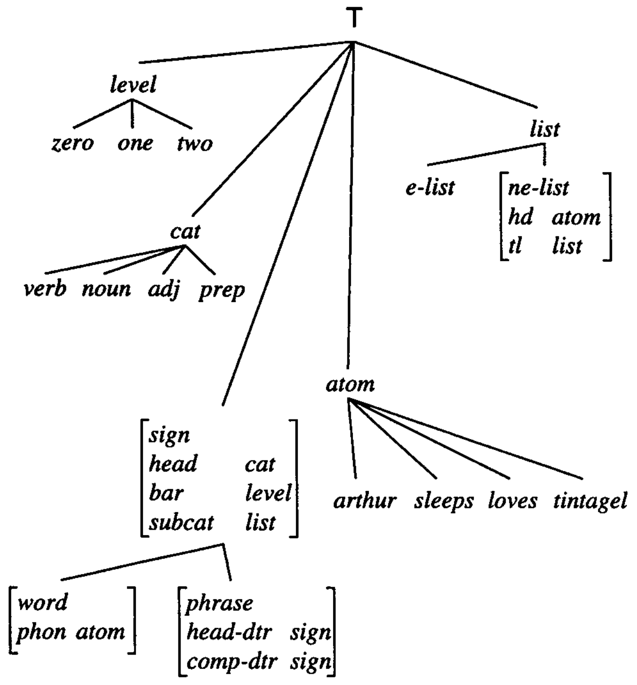


Figure 6: A type hierarchy

the syntactic category (*sign*) as functional style result argument.[5] The analysis tree is encoded in two daughters features as part of the syntactic categories in the style of HPSG. Clause 1 of *wfs* combines a verbal projection with its subject, and clause 2 with its complements. The lexical entries for the verbs "loves" and "sleeps" are specified in clauses 3 and 4, respectively. Finally, clause 5 defines lexical entries for the proper names "Arthur" and "Tintagel".

Now consider the principles defined in Fig. 8. Constraints 1-4 encode a simple version of X-bar theory, constraint 5 ensures the propagation of categorial information along the head path, and constraint 6 ensures that complements obey the subcategorization requirements of the heads.

We may now ask the system for a solution to queries like *wfs*([*arthur*, *sleeps*], []). The solution in this case is the AVM in Fig. 9.

We can also query for a term like *word* ∧ *subcat* : *ne-list* and check it against the implications alone, as it contains no relational goals. The result in Fig. 10 shows that our X-bar principles have applied: *bar* level *two* requires that the *subcat* list must be empty, and *bar* level *one* can only appear on phrases. The system thus correctly infers that only *bar* level *zero* is possible.

[5]We use standard abbreviatory bracket notation for lists.

1. *wfs*(*P*0,*P*) :=
   phrase ∧ *head* : *verb* ∧ *subcat* : [ ] ∧
   bar : *two* ∧ *comp-dtr* : *wfs*(*P*0, *P*1) ∧
   head-dtr : *wfs*(*P*1, *P*)
2. *wfs*(*P*0, *P*) :=
   phrase ∧ *head* : *verb* ∧ *subcat* : *ne-list* ∧
   head-dtr : *wfs*(*P*0, *P*1) ∧
   comp-dtr : *wfs*(*P*1, *P*)
3. *wfs*([*X*|*Y*], *Y*) :=
   word ∧ *head* : *verb* ∧ *bar* : *zero* ∧
   subcat : [*head* : *noun*, *head* : *noun*] ∧
   phon : (*loves* ∧ *X*)
4. *wfs*([*X*|*Y*], *Y*) :=
   word ∧ *head* : *verb* ∧ *bar* : *zero* ∧
   subcat : [*head* : *noun*] ∧ *phon* : (*sleeps* ∧ *X*)
5. *wfs*([*X*|*Y*], *Y*) :=
   word ∧ *head* : *noun* ∧
   bar : *two* ∧ *subcat* : [ ] ∧
   phon : ((*arthur* ∨ *tintagel*) ∧ *X*)

Figure 7: Phrase structure rules and the lexicon

1.	*bar* : *zero*	→	*word*
2.	*bar* : *one*	→	*head-dtr* : *bar* : (¬*two*)
3.	*bar* : *two*	→	*subcat* : [ ]
4.	*phrase*	→	*comp-dtr* : *bar* : *two*
5.	*phrase*	→	*head* : *X* ∧ *head-dtr* : *head* : *X*
6.	*phrase*	→	*comp-dtr* : *X* ∧ *subcat* : *Y* ∧ *head-dtr* : *subcat* : [*X*|*Y*]

Figure 8: X-bar theory, head feature principle and subcat principle

The advantages of such a modular encoding of grammatical principles are obvious. The intuition behind the constraints is clear, and new rules are easily added, since the principles apply to any rule. On the other hand, one can experiment with individual principles without having to change the other principles or rules. Finally, the option of encoding grammatical constraints as either implicational constraints or relations opens the possibility to chose the encoding most naturally suited to the specific problem. We feel that this improves on earlier, purely definite-clause-based approaches.

Implementation

Compilation Building on the compilation method described in Götz and Meurers (1995), our compiler

$$\left[\begin{array}{ll}
\textit{phrase} & \\
\text{BAR} & \textit{two} \\
\text{HEAD} & \boxed{2}\ \textit{verb} \\
\text{SUBCAT} & \boxed{3} \\
\text{COMP-DTR} & \boxed{1} \left[\begin{array}{ll} \textit{word} & \\ \text{PHON} & \textit{arthur} \\ \text{BAR} & \textit{two} \\ \text{HEAD} & \textit{noun} \\ \text{SUBCAT} & \textit{e-list} \end{array}\right] \\
\text{HEAD-DTR} & \left[\begin{array}{ll} \textit{word} & \\ \text{PHON} & \textit{sleeps} \\ \text{BAR} & \textit{zero} \\ \text{HEAD} & \boxed{2} \\ \text{SUBCAT} & \left[\begin{array}{ll} \textit{ne-list} & \\ \text{HD} & \boxed{1} \\ \text{TL} & \boxed{3}\ \textit{e-list} \end{array}\right] \end{array}\right]
\end{array}\right]$$

Figure 9: Solution to the query *wfs*([*arthur*, *sleeps*], [])

$$\left[\begin{array}{ll}
\textit{word} & \\
\text{BAR} & \textit{zero} \\
\text{SUBCAT} & \textit{ne-list}
\end{array}\right]$$

Figure 10: Solution for the query *word* $\wedge$ *subcat* : *ne-list*

collects the types for which principles are formulated, defines a relational encoding of the principles, and attaches calls to the relations at the places in the grammar where a constrained type can occur. We assume that the grammar writer guarantees that each type in the grammar is consistent (for a grammar G and every type t there is a model of G that satisfies t). One therefore does not need to attach calls to each possible occurrence of a constrained type, but only to those occurrences where the grammar contains additional specifications which might lead to an inconsistency (Götz and Meurers 1996). The interpretation of the resulting program is lazy in the sense that we do not enumerate fully specific solutions but compute more general answers for which a grammatical instantiation is guaranteed to exist. A good example for this behaviour was shown in Fig. 10: the system does not instantiate the PHON and the HEAD values of the solution, since the existence of grammatical values for these attributes is independent of the query.

A way in which we deviate from the compilation method of Götz and Meurers (1995) is that our system performs all constraint inheritance at compile-time. While inheriting all principles to the most specific types and transforming the resulting constraints to a disjunctive normal form can significantly slow down compile times, the advantage is that no inheritance needs to be done on-line. To influence this trade-off, the user can instruct the system to hide a disjunctive principle in an auxiliary relation in order to keep it from being multiplied out with the other constraints. Such auxiliary relations, which will be discussed further in connection with the delay mechanism, have turned out to be especially useful in conjunction with principles with complex antecedents. The reason is that our compiler transforms an implication with complex antecedents to an implication with a type antecedent. The negation of the complex antecedent is added to the consequent, which can result in highly disjunctive specifications.

Interpretation As a guiding principle, the interpreter follows the ideas of the Andorra Model[6] in that it always executes deterministic goals before non-deterministic ones. We consider determinacy only with respect to head unification: a goal is recognised to be determinate if there is at most one clause head that unifies with it. This evaluation strategy has two advantages: it reduces the number of choice points to a minimum, and it leads to early failure detection. In our implementation, the overhead of determining which goals are determinate has turned out to be by far outweighed by the reduction in search space for our linguistic applications. An additional speed-up can be expected from applying known pre-processing techniques (Santos Costa, Warren, and Yang 1991) to automatically extract so-called determinacy code.

The execution order of non-determinate goals can be influenced by the user with wait declarations (Naish 1985). The execution of some goal is postponed until the call is more specific than a user-specified term. Speculative computation may thus be reduced to a necessary minimum. For our previous example, we might define the delay statements in Fig. 11. The first state-

```
delay(wfs,arg1:list)
delay(phrase,subcat:list)
delay_deterministic(sign)
```

Figure 11: Control statement examples

ment says that calls to *wfs* must be delayed until the first argument is instantiated to some list value. Similarly, the second statement delays the principles on *phrase* until the subcat information is known. The third statement is of a slightly different form, based on the preferred treatment of determinate goals described above. Instead of specifying the instantiation state required for execution, the delay_deterministic statement

[6]Cf. Haridi and Janson (1990) and references cited therein.

specifies that the universal principles about signs can only be executed in case they are determinate.

The delay mechanism for relational goals is very close to the one used in CUF. We extended this mechanism to the universal principles: the constraints on a certain type were only checked, once certain attributes were sufficiently instantiated (w.r.t. the delay statement). Our experience has shown, however, that delaying universal principles in such a way turns out to be too weak. Instead of delaying all constraints on a type until some condition is met, one wants to be able to postpone the application of some particular universal principle. A subcategorization principle applying to phrases, for example, should be delayed until the valence requirements of the mother or the daughters are known. We therefore allow the user to name a principle and supply it with a specific delay. Internally, this corresponds to introducing an auxiliary relation under the name supplied by the user and delaying it accordingly so that the choice points introduced by the principle are hidden.

Let us illustrate the problem and its solution with a schematic example. Suppose the grammar writer writes a principle $\phi \rightarrow \psi$. Our compiler will generate from this a constraint $t \rightarrow (\neg\phi) \vee (\phi \wedge \psi)$, for some appropriate type t. If ϕ is a complex conjunctive description, then the result of normalising $\neg\phi$ might be highly disjunctive. This has two undesirable consequences. Firstly, if there is another constraint $t \rightarrow \xi$ with disjunctive ξ, then the compiler will need to normalise the expression $((\neg\phi)\vee(\phi\wedge\psi))\wedge\xi$. This is the appropriate thing to do in those cases where many of the generated disjuncts are inconsistent and the resulting disjunction thus turns out to be small. If, however, these constraints talk about different parts of t's structure, then the resulting disjunction will be big and the expansion at compile-time should be avoided.

The other problem is that we can only specify delays on *all* constraints on t at once, and cannot delay individual principles. In other words, the control for the execution of principles is not fine-grained enough.

We solved these problems by offering the user the possibility to name constraints, e.g., *principle1* : $\phi \rightarrow \psi$. This prohibits the compile-time cross-multiplication described above, and it allows the user to specify delays for such a principle, e.g. `delay(principle1, ...)` or even `delay_deterministic(principle1)`, if that is appropriate.

Debugging Having addressed the key issues behind compilation and interpretation, we now turn to a practical problem which quickly arises once one tries to implement larger grammars. On the one hand, the complex data structures of such grammars contain an overwhelming number of specifications which are difficult to present to the user. On the other hand, the interaction of universal principles and relations tends to get very complex for realistic linguistic theories. While a powerful graphical user interface[7] solves the presentation problem, a sophisticated tracing and debugging tool was developed to allow stepwise inspection of the complex constraint resolution process. The debugger displays the feature structure(s) to be checked for grammaticality and marks the nodes on which constraints still have to be checked. As a result of the determinacy check, each such node can also be marked as failed, delayed or deterministic. Similar to standard Prolog debuggers, the user can step, skip, or fail a constraint on a node, or request all deterministic processing to be undertaken. An interesting additional possibility for non-deterministic goals is that the user can inspect the matching defining clauses and chose which one the system should try. Figure 12 below shows a screen shot of the debugger.

The debugger has turned out to be an indispensable tool for grammar development. As grammar size increases, it becomes very difficult to track down bugs or termination problems without it, since these problems are often the result of some global interaction and thus cannot be reduced to a manageable sub-part of the grammar.

The reader interested in further practical aspects of our system is referred to (Götz and Meurers 1997)

Comparison with previous work

There are quite a number of typed feature systems available today, among them ALE (Carpenter and Penn 1994), CUF (Dörre and Dorna 1993) and TFS (Emele and Zajac 1990; Emele 1994).

TFS also offered type constraints and relations and to our knowledge was the first working typed feature systems. However, it had some serious drawbacks. TFS did not allow universal principles with complex antecedents, but only type constraints. And the system did not include a delay mechanism, so that it was often impossible to ensure termination or efficient processing. The addition of a delay mechanism as described in this paper would certainly increase the efficiency of TFS.

ALE provides relations and type constraints (i.e., only types as antecedents), but their unfolding is neither lazy, nor can it be controlled by the user in any way.

[7]To view grammars and computations our system uses a GUI which allows the user to interactively view (parts of) AVMs, compare and search AVMs, etc. The GUI comes with a clean backend interface and has already been used as front-end for other natural language applications, e.g., in VERBMOBIL. The GUI was developed by Carsten Hess.

This can lead to severe termination problems with recursive constraints. The ALE type constraints were designed to enhance the typing system, and not for recursive computation. This should be done in the phrase structure or procedural attachment part. However, we believe that the addition of delaying and an interpretation strategy as described in this paper would add to the attractiveness of ALE as a constraint-based grammar development platform.

The definite clause part of our system is very similar to the one of CUF: both use delay statements and preferred execution of deterministic goals. Although CUF does not offer universal principles, their addition should be relatively simple. Given that CUF already offers the control strategies required by our scheme, the changes to the run-time system would be minimal.

Conclusion and future research

We have presented an architecture that integrates relational and implicational constraints over typed feature logic. We showed how such an architecture facilitates the modular and compact encoding of principle based grammars.

Our implementation has been tested with several smaller and one large (> 5000 lines) grammar, a linearisation-based grammar of a sizeable fragment of German (Hinrichs et al. 1997). As the grammar constraints combine sub-strings in a non-concatenative fashion, we use a preprocessor that "chunks" the input string into linearisation domains, which are then fed to the constraint solver. With our Prolog based interpreter, parse times are around 1-5 sec. for 5 word sentences and 10-60 sec. for 12 word sentences. It should be pointed out that parsing with such a grammar would be difficult with any system, as it does neither have nor allow the addition of a context-free backbone.

We are currently experimenting with a C based compiler (Zahnert 1997) using an abstract machine with a specialised set of instructions based on the WAM (Warren 1983; Aï-Kaci 1991). This compiler is still under development, but it is reasonable to expect speed improvements of at least an order of magnitude. Abstract-machine-based compilation of typed feature logic languages has recently received much attention (Carpenter and Qu 1995, Wintner 1997, Penn in prep.). True compilation is the logical development in a maturing field that has hitherto relied on interpreters in high-level programming languages such as Prolog and Lisp.

We also plan to investigate a specialised constraint language for linearisation grammars, to be able to optimise the processing of freer word order languages such as German.

References

Aï-Kaci, H. (1991). *Warren's Abstract Machine.* MIT Press.

Carpenter, B. (1992). *The logic of typed feature structures,* Volume 32 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press.

Carpenter, B. and G. Penn (1994). ALE – The Attribute Logic Engine, User's Guide, Version 2.0.1, December 1994. Technical report, Carnegie Mellon University.

Carpenter, B. and Y. Qu (1995). An abstract machine for attribute-value logics. In *Proceedings of the Fourth International Workshop on Parsing Technology.* Prague.

Dörre, J. and M. Dorna (1993, August). CUF - a formalism for linguistic knowledge representation. In J. Dörre (Ed.), *Computational aspects of constraint based linguistic descriptions I,* pp. 1–22. Universität Stuttgart: DYANA-2 Deliverable R1.2.A.

Emele, M. C. (1994). The typed feature structure representation formalism. In *Proceedings of the International Workshop on Sharable Natural Language Resources,* Ikoma, Nara, Japan.

Emele, M. C. and R. Zajac (1990). Typed unification grammars. In *Proceedings of the 13th International Conference on Computational Linguistics.*

Götz, T. and W. D. Meurers (1995). Compiling HPSG type constraints into definite clause programs. In *Proceedings of the Thrirty-Third Annual Meeting of the ACL,* Boston. Association for Computational Linguistics.

Götz, T. and W. D. Meurers (1996). The importance of being lazy - using lazy evaluation to process queries to HPSG grammars. In P. Blache (Ed.), *Actes de la troisième conférence anuelle sur le traitment automatique du langage naturel.*

Götz, T. and W. D. Meurers (1997). The ConTroll system as large grammar development platform. In *Proceedings of the ACL/EACL post-conference workshop on Computational Environments for Grammar Development and Linguistic Engineering,* Madrid, Spain.

Haridi, S. and S. Janson (1990). Kernel Andorra Prolog and its computation model. In D. H. D. Warren and P. Szeredi (Eds.), *Proceedings of the seventh international conference on logic programming,* pp. 31–46. MIT Press.

Hinrichs, E., D. Meurers, F. Richter, M. Sailer, and H. Winhart (1997). Ein HPSG-Fragment des Deutschen, Teil 1: Theorie. Arbeitspapiere des SFB 340 Nr. 95, Universität Tübingen.

King, P. J. (1989). *A logical formalism for head-driven phrase structure grammar.* Ph. D. thesis, University of Manchester.

Naish, L. (1985). *Negation and Control in Prolog.* Springer-Verlag.

Penn, G. (in prep.). *Statistical Optimizations in a Feature Structure Abstract Machine.* Ph. D. thesis, Carnegie Mellon University.

Pollard, C. and I. A. Sag (1994). *Head-Driven Phrase Structure Grammar.* Chicago: University of Chicago Press.

Santos Costa, V., D. H. D. Warren, and R. Yang (1991). The Andorra-I preprocessor: Supporting full Prolog on the Basic Andorra model. In *Proceedings of the Eighth International Conference on Logic Programming*, pp. 443–456.

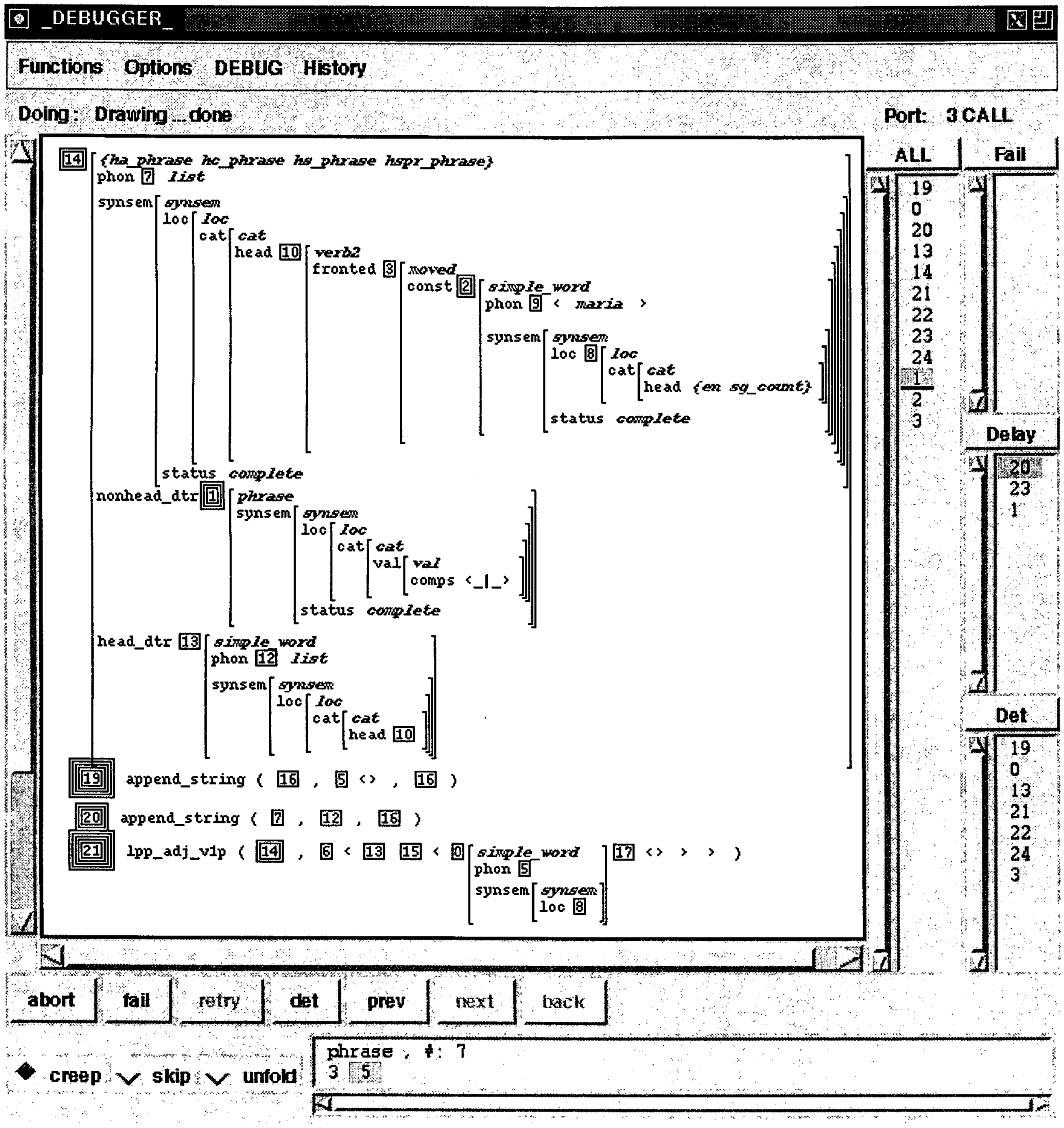
Shieber, S. M. (1986). *An Introduction to Unification-Based Approaches to Grammar.* Number 4 in CSLI Lecture Notes. Center for the Study of Language and Information.

Stabler, E. P. and M. Johnson (1993). Topics in principle based parsing. Course notes for the 1993 LSA Summer Institute.

Warren, D. H. D. (1983). An abstract Prolog instruction set. Technical note 309, SRI International.

Wintner, S. (1997). *An Abstract Machine for Unification Grammars.* Ph. D. thesis, Technion, Haifa, Israel.

Zahnert, A. (1997). fl2c – ein Compiler für CLP(TFS). Diplomarbeit, Fakultät für Informatik, Universität Tübingen.


DEBUGGER
Functions Options DEBUG History
abort fail retry det prev next back
creep skip unfold
ALL
Fail
Delay
Det

```
Doing: Drawing...done                                  Port: 3 CALL

[14] {ha_phrase hc_phrase hs_phrase hspr_phrase}
     phon [7] list
     synsem [ synsem
              loc [ loc
                    cat [ cat
                          head [10] [ verb2
                                      fronted [3] [ moved
                                                    const [2] [ simple_word
                                                                phon [9] < maria >
                                                                synsem [ synsem
                                                                         loc [8] [ loc
                                                                                   cat [ cat
                                                                                         head {en sg_count} ] ]
                                                                         status complete ] ] ] ] ] ]
              status complete ]
     nonhead_dtr [1] [ phrase
                       synsem [ synsem
                                loc [ loc
                                      cat [ cat
                                            val [ val
                                                  comps <_|_> ] ] ]
                                status complete ] ]
     head_dtr [13] [ simple_word
                     phon [12] list
                     synsem [ synsem
                              loc [ loc
                                    cat [ cat
                                          head [10] ] ] ] ]

[19] append_string ( [16] , [5] <> , [16] )
[20] append_string ( [7] , [12] , [16] )
[21] lpp_adj_v1p ( [14] , [6] < [13] [15] < [0] [ simple_word
                                                 phon [5]
                                                 synsem [ synsem
                                                          loc [8] ] ] [17] <> > > )

ALL     Fail     Delay     Det
19               20        19
0                23        0
20               1         13
13                         21
14                         22
21                         24
22                         3
23
24
1
2
3

phrase , #: 7
3 5
```

Figure 12: A screen shot of the graphical debugger