Balancing Clarity and Efficiency in Typed Feature Logic through Delaying

Gerald Penn
University of Toronto
10 King's College Rd.
Toronto M5S 3G4
Canada
gpenn@cs.toronto.edu

Abstract

The purpose of this paper is to re-examine the balance between clarity and efficiency in HPSG design, with particular reference to the design decisions made in the English Resource Grammar (LinGO, 1999, ERG). It is argued that a simple generalization of the conventional delay statements used in logic programming is sufficient to restore much of the functionality and concomitant benefit that the ERG elected to forego, with an acceptable although still perceptible computational cost.

1 Motivation

By convention, current HPSGs consist, at the very least, of a deductive backbone of extended phrase structure rules, in which each category is a description of a typed feature structure (TFS), augmented with constraints that enforce the principles of grammar. These principles typically take the form of statements, "for all TFSs, ψ holds," where ψ is usually an implication. Historically, HPSG used a much richer set of formal descriptive devices, however, mostly on analogy to developments in the use of types and description logics in programming language theory (Aït-Kaći, 1984), which had served as the impetus for HPSG's invention (Pollard, 1998). This included logic-programming-style relations (Höhfeld and Smolka, 1988), a powerful description language in which expressions could denote *sets* of TFSs through the use of an explicit disjunction operator, and the full expressive power of implications, in which antecedents of the above-mentioned ψ principles could be arbitrarily complex.

Early HPSG-based natural language processing systems faithfully supported large chunks of this richer functionality, in spite of their inability to handle it efficiently — so much so that when the designers of the ERG set out to select formal descriptive devices for their implementation with the aim of "balancing clarity and efficiency," (Flickinger, 2000), they chose to include none of these amenities. The ERG uses only phrase-structure rules and type-antecedent constraints, pushing all would-be description-level disjunctions into its type system or rules. In one respect, this choice was successful, because it did at least achieve a respectable level of efficiency. But the ERG's selection of functionality has acquired an almost liturgical status within the HPSG community in the intervening seven years. Keeping this particular faith, moreover, comes at a considerable cost in clarity, as will be argued below.

This paper identifies what it is precisely about this extra functionality that we miss (modularity, Section 2), determines what it would take at a minimum computationally to get it back (delaying, Section 3), and attempts to measure exactly how much that minimal computational overhead would cost (about 2.5 μs per delay, Section 4). This study has not been undertaken before; the ERG designers' decision was based on largely anecdotal accounts of performance relative to then-current implementations that had not been designed with the intention of minimizing this extra cost (indeed, the ERG baseline had not yet been devised).

2 Modularity: the cost in clarity

Semantic types and inheritance serve to organize the constraints and overall structure of an HPSG grammar. This is certainly a familiar, albeit vague justification from programming languages research, but the comparison between HPSG and modern programming languages essentially ends with this statement.

Programming languages with inclusional polymorphism (subtyping) invariably provide functions or relations and allow these to be reified as *methods* within user-defined subclasses/subtypes. In HPSG, however, values of features must necessarily be TFSs themselves, and the only method (implicitly) provided by the type signature to act on these values is unification. In the absence of other methods and in the absence of an explicit disjunction operator, the type signature itself has the responsibility of not only declaring definitional sub-

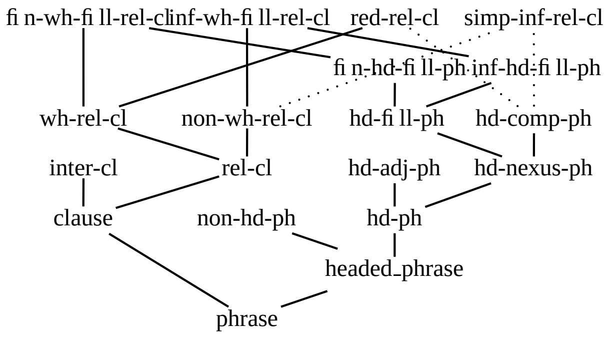


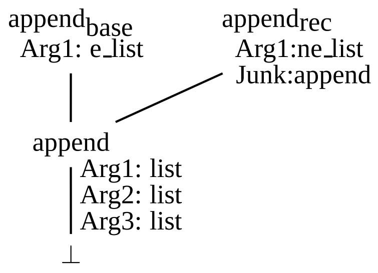
Figure 1: Relative clauses in the ERG (partial).

class relationships, but expressing all other non-definitional disjunctions in the grammar (as subtyping relationships). It must also encode the necessary accoutrements for implementing all other necessary means of combination as unification, such as difference lists for appending lists, or the so-called *qeq* constraints of Minimal Recursion Semantics (Copestake et al., 2003) to encode semantic embedding constraints.

Unification, furthermore, is an inherently non-modular, global operation because it can only be defined relative to the structure of the entire partial order of types (as a least upper bound). Of course, some partial orders are more modularizable than others, but legislating the global form that type signatures must take on is not an easy property to enforce without more local guidance.

The conventional wisdom in programming languages research is indeed that types are responsible for mediating the communication between modules. A simple type system such as HPSG's can thus only mediate very simple communication. Modern programming languages incorporate some degree of *parametric* polymorphism, in addition to subtyping, in order to accommodate more complex communication. To date, HPSG's use of parametric types has been rather limited, although there have been some recent attempts to apply them to the ERG (Penn and Hoetmer, 2003). Without this, one obtains type signatures such as Figure 1 (a portion of the ERG's for relative clauses), in which both the semantics of the subtyping links themselves (normally, subset inclusion) and the multi-dimensionality of the empirical domain's analysis erode into a collection of arbitrary naming conventions that are difficult to validate or modify.

A more avant-garde view of typing in programming languages research, inspired by the Curry-Howard isomorphism, is that types are equivalent to relations, which is to say that a relation can mediate communication between modules through its arguments, just as a parametric type can through its parameters. The fact that we witness some of these mediators as types and others as relations is simply an intensional reflection of how the grammar writer thinks of them. In classical HPSG, relations were generally used as goals in some proof resolution strategy (such as Prolog's SLD resolution), but even this has a parallel in the world of typing. Using the type signature and principles of Figure 2, for ex-



$$\text{append}_{\text{base}} \implies Arg2 : L \ \wedge \ Arg3 : L.$$
$$\text{append}_{\text{rec}} \implies Arg1 : [H|L1] \wedge Arg2 : L2 \ \wedge \ Arg3 : [H|L3] \wedge Junk : (append \ \wedge \ A1 : L1 \wedge A2 : L2 \ \wedge \ Arg3 : L3).$$

Figure 2: Implementing SLD resolution over the append relation as sort resolution.

ample, we can perform proof resolution by attempting to *sort resolve* every TFS to a maximally specific type. This is actually consistent with HPSG's use of feature logic, although most TFS-based NLP systems do not sort resolve because type inference under sort resolution is NP-complete (Penn, 2001).

Phrase structure rules, on the other hand, while they can be encoded inside a logic programming relation, are more naturally viewed as algebraic generators. In this respect, they are more similar to the immediate subtyping declarations that grammar writers use to specify type signatures — both chart parsing and transitive closure are instances of all-source shortest-path problems on the same kind of algebraic structure, called a *closed semi-ring*. The only notion of modularity ever proven to hold of phrase structure rule systems (Wintner, 2002), furthermore, is an algebraic one.

3 Delaying: the missing link of functionality

If relations are used in the absence of recursive data structures, a grammar could be specified using relations, and the relations could then be unfolded offline into relation-free descriptions. In this usage, relations are just macros, and not at all inefficient. Early HPSG implementations, however, used quite a lot of recursive structure where it did not need to be, and the structures they used, such as lists, buried important data deep inside substructures that made parsing much slower. Provided that grammar writers use more parsimonious structures, which is a good idea even in the absence of relations, there is nothing wrong with the speed of logic programming relations (Van Roy, 1990).

Recursive datatypes are also prone to non-termination problems, however. This can happen when partially instantiated and potentially recursive data structures are submitted to a proof resolution procedure which explores the further instantiations of these structures too aggressively. Although this problem has received significant attention over the last fifteen years in the constraint logic programming (CLP) community, no true CLP implementation yet exists for the logic of typed feature structures (Carpenter, 1992, LTFS). Some aspects of general solution strategies, including incremental entailment simplification (Aït-Kaci et al., 1992), deterministic goal expansion (Doerre, 1993), and guard statements for relations (Doerre et al., 1996) have found their way into the less restrictive sorted feature constraint systems from which LTFS descended. The CUF implementation (Doerre et al., 1996), notably, allowed for delay statements to be attached to relation definitions, which would wait until each argument was at least as specific as some variable-free, disjunction-free description before resolving.

In the remainder of this section, a method is presented for reducing delays on any inequation-free description, including variables and disjunctions, to the SICStus Prolog `when/2` primitive (Sections 3.4). This method takes full advantage of the restrictions inherent to LTFS (Section 3.1) to maximize run-time efficiency. In addition, by delaying calls to subgoals individually rather than the (universally quantified) relation definitions themselves,[1] we can also use delays to postpone non-deterministic search on disjunctive descriptions (Section 3.3) and to implement complex-antecedent constraints (Section 3.2). As a result, this single method restores all of the functionality we were missing.

For simplicity, it will be assumed that the target language of our compiler is Prolog itself. This is inconsequential to the general proposal, although implementing logic programs in Prolog certainly involves less effort.

[1]Delaying relational definitions is a subcase of this functionality, which can be made more accessible through some extra syntactic sugar.

3.1 Restrictions inherent to LTFS

LTFS is distinguished by its possession of *appropriateness conditions* that mediate the occurrence of features and types in these records. Appropriateness conditions stipulate, for every type, a finite set of features that can and must have values in TFSs of that type. This effectively forces TFSs to be finite-branching terms with named attributes. Appropriateness conditions also specify a type to which the value of an appropriate feature is restricted (a *value restriction*). These conditions make LTFS very convenient for linguistic purposes because the combination of typing with named attributes allows for a very terse description language that can easily make reference to a sparse amount of information in what are usually extremely large structures/records:

Definition: *Given a finite meet semi-lattice of types,* Type, *a fixed finite set of features,* Feat, *and a countable set of variables,* Var, Φ *is the least set of descriptions that contains:*

- $v, v \in Var$,
- $\tau, \tau \in Type$,
- $\textsc{f} : \phi, \textsc{f} \in Feat, \phi \in \Phi$,
- $\phi_1 \wedge \phi_2, \phi_1, \phi_2 \in \Phi$, *and*
- $\phi_1 \vee \phi_2, \phi_1, \phi_2 \in \Phi$.

A nice property of this description language is that every non-disjunctive description with a non-empty denotation has a unique most general TFS in its denotation. This is called its *most general satisfier*.

We will assume that appropriateness guarantees that there is a unique most general type, $\texttt{Intro}(\textsc{f})$ to which a given feature, F, is appropriate. This is called *unique feature introduction*. Where unique feature introduction is not assumed, it can be added automatically in $\mathcal{O}(F \cdot T)$ time, where F is the number of features and T is the number of types (Penn, 2001). Meet semi-latticehood can also be restored automatically, although this involves adding exponentially many new types in the worst case.

3.2 Complex Antecedent Constraints

It will be assumed here that all complex-antecedent constraints are implicitly universally quantified, and are of the form:

$$\alpha \Longrightarrow (\gamma \wedge \rho)$$

where α, γ are descriptions from the core description language, Φ, and ρ is drawn from a definite clause language of relations, whose arguments are also descriptions from Φ. As mentioned above, the ERG uses the same form, but where α can only be a type description, τ, and ρ is the trivial goal, `true`.

The approach taken here is to allow for arbitrary antecedents, α, but still to interpret the implications of principles using subsumption by α, i.e., for every TFS (the implicit universal quantification is still there), either the consequent holds, or the TFS is not subsumed by the most general satisfier of α. The subsumption convention dates back to the TDL (Krieger and Schäfer, 1994) and ALE (Carpenter and Penn, 1996) systems, and has earlier antecedents in work that applied lexical rules by subsumption (Krieger and Nerbone, 1991). The ConTroll constraint solver (Goetz and Meurers, 1997) attempted to handle complex antecedents, but used a classical interpretation of implication and no deductive phrase-structure backbone, which created a very large search space with severe non-termination problems.

Within CLP more broadly, there is some related work on guarded constraints (Smolka, 1994) and on inferring guards automatically by residuation of implicational rules (Smolka, 1991), but implicit universal quantification of all constraints seems to be unique to linguistics. In most CLP, constraints on a class of terms or objects must be explicitly posted to a *store* for each member of that class. If a constraint is not posted for a particular term, then it does not apply to that term.

The subsumption-based approach is sound with respect to the classical interpretation of implication for those principles where the classical interpretation really is the correct one. For completeness, some additional resolution method (in the form of a logic program with relations) must be used. As is normally the case in CLP, deductive search is used alongside constraint resolution.

Under such assumptions, our principles can be converted to:

$$trigger(\alpha) \Longrightarrow v \wedge \texttt{whenfs}((v=\alpha),((v=\gamma)\wedge\rho))$$

Thus, with an implementation of type-antecedent constraints and an implementation of `whenfs/2` (Section 3.3), which delays the goal in its second argument until v is subsumed by (one of) the most general satisfier(s) of description α, all that remains is a method for finding the *trigger*, the most efficient type antecedent to use, i.e., the most general one that will not violate soundness. $trigger(\alpha)$ can be defined as follows:

- $trigger(v) = \bot$,
- $trigger(\tau) = \tau$,
- $trigger(\text{F} : \phi) = \mathit{Intro}(\text{F})$,
- $trigger(\phi_1 \wedge \phi_2) = trigger(\phi_1) \sqcup trigger(\phi_2)$, and
- $trigger(\phi_1 \vee \phi_2) = trigger(\phi_1) \sqcap trigger(\phi_2)$,

where $\sqcup$ and $\sqcap$ are respectively unification and generalization in the type semi-lattice.

In this and the next two subsections, we can use Figure 3 as a running example of the various stages of compilation of a typical complex-antecedent constraint, namely the Finiteness Marking Principle for German (1). This constraint is stated relative to the signature shown in Figure 4. The description to the left of the arrow in Figure 3 (1) selects TFSs whose substructure on the path SYNSEM:LOC:CAT satisfies two requirements: its HEAD value has type *verb*, and its MARKING value has type *fin*. The principle says that every TFS that satisfies that description must also have a SYNSEM: LOC: CAT: HEAD: VFORM value of type *bse*.

To find the trigger in Figure 3 (1), we can observe that the antecedent is a feature value description (F:ϕ), so the trigger is *Intro*(SYNSEM), the unique introducer of the SYNSEM feature, which happens to be the type *sign*. We can then transform this constraint as above (Figure 3 (2)). The `cons` and `goal` operators in (2)–(5) are ALE syntax, used respectively to separate the type antecedent of a constraint from the description component of the consequent (in this case, just the variable, `X`), and to separate the description component of the consequent from its relational attachment. We know that any TFS subsumed by the original antecedent will also be subsumed by the most general TFS of type *sign*, because *sign* introduces SYNSEM.

3.3 Reducing Complex Conditionals

Let us now implement our delay predicate, `whenfs(V=Desc,Goal)`. Without loss of generality, it can be assumed that the first argument is actually drawn from a more general conditional language, including those of the form $V_i = Desc_i$ closed under conjunction and disjunction. It can also be assumed that the variables of each $Desc_i$ are distinct. Such a complex conditional can easily be converted into a normal form in which each atomic conditional contains a non-disjunctive description. Conjunction and disjunction of atomic conditionals then reduce as follows (using the Prolog convention of comma for AND and semi-colon for OR):

```
whenfs((VD1,VD2),Goal) :-
   whenfs(VD1,whenfs(VD2,Goal)).
whenfs((VD1;VD2),Goal) :-
   whenfs(VD1,(Trigger = 0 -> Goal
                ; true)),
   whenfs(VD2,(Trigger = 1 -> Goal
                ; true)).
```

The binding of the variable `Trigger` is necessary to ensure that Goal is only resolved once in case the

```
(1) synsem:loc:cat:(head:verb,marking:fin) ⟹ synsem:loc:cat:head:vform:bse.
(2) sign cons X goal
      whenfs((X=synsem:loc:cat:(head:verb,marking:fin)),
            (X=synsem:loc:cat:head:vform:bse)).
(3) sign cons X goal
      whentype(sign,X,(farg(synsem,X,SynVal),
      whentype(synsem,SynVal,(farg(loc,SynVal,LocVal),
      whentype(local,LocVal,(farg(cat,LocVal,CatVal),
      whenfs((CatVal=(head:verb,marking:fin)),
      (X=synsem:loc:cat:head:vform:bse))))))).
(4) sign cons X goal
      (whentype(sign,X,(farg(synsem,X,SynVal),
      whentype(synsem,SynVal,(farg(loc,SynVal,LocVal),
      whentype(local,LocVal,(farg(cat,LocVal,CatVal),
      whentype(category,CatVal,(farg(head,CatVal,HdVal),
      whentype(verb,HdVal,
      whentype(category,CatVal,(farg(marking,CatVal,MkVal),
      whentype(fin,MkVal,
            (X=synsem:loc:cat:head:vform:bse)))))))))))))).
(5) sign cons X goal
      (farg(synsem,X,SynVal),
      farg(loc,SynVal,LocVal),
      farg(cat,LocVal,CatVal),
      farg(head,CatVal,HdVal),
      whentype(verb,HdVal,(farg(marking,CatVal,MkVal),
      whentype(fin,MkVal,
      (X=synsem:loc:cat:head:vform:bse))))).
(6) sign(e_list(_),e_list(_),SynVal,DelayVar)
(7) whentype(Type,FS,Goal) :-
      functor(FS,CurrentType,Arity),
      (sub_type(Type,CurrentType) -> call(Goal)
      ; arg(Arity,FS,DelayVar), when(nonvar(DelayVar),whentype(Type,DelayVar,Goal))).
```

Figure 3: Reduction stages for the Finiteness Marking Principle.

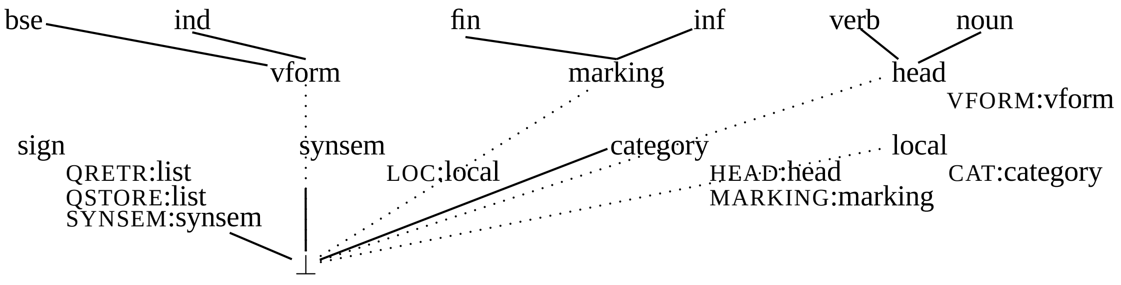


Figure 4: Part of the signature underlying the constraint in Figure 3.

goals for both conditionals eventually unsuspend.

For atomic conditionals, we must thread two extra arguments, `VsIn`, and `VsOut`, which track which variables have been seen so far. Delaying on atomic type conditionals is implemented by a special `whentype/3` primitive (Section 3.4), and feature descriptions reduce using unique feature introduction:

```
whenfs(V=T,Goal,Vs,Vs) :-
   type(T),!,whentype(T,V,Goal).
whenfs(V=(F:Desc),Goal,VsIn,VsOut):-
   unique_introducer(F,Intro),
   whentype(Intro,V,
         (farg(F,V,FVal),
   whenfs(FVal=Desc,Goal,VsIn,
         VsOut))).
```

`farg(F,V,FVal)` binds `FVal` to the argument position of `V` that corresponds to the feature `F` once `V` has been instantiated to a type for which `F` is appropriate.

In the variable case, `whenfs/4` simply binds the variable when it first encounters it, but subsequent occurrences of that variable create a suspension using Prolog `when/2`, checking for identity with the previous occurrences. This implements a primitive delay on structure sharing (Section 3.4):

```
whenfs(V=X,Goal,VsIn,VsOut) :-
    var(X),
   (select(VsIn,X,VsOut)
    -> % not first X - wait
       when(?=(V,X),
         ((V==X) -> call(Goal) ; true))
    ; % first X - bind
       VsOut=VsIn,V=X,call(Goal)).
```

In practice, `whenfs/2` can be partially evaluated by a compiler. In the running example, Figure 3, we can compile the `whenfs/2` subgoal in (2) into simpler `whentype/2` subgoals, that delay until `X` reaches a particular type. The second case of `whenfs/4` tells us that this can be achieved by successively waiting for the types that introduce each of the features, SYNSEM, LOC, and CAT. As shown in Figure 4, those types are *sign*, *synsem* and *local*, respectively (Figure 3 (3)).

The description that `CatVal` is suspended on is a conjunction, so we successively suspend on each conjunct. The type that introduces both HEAD and MARKING is *category* (4). In practice, static analysis can greatly reduce the complexity of the resulting relational goals. In this case, static analysis of the type system tells us that all four of these `whentype/2` calls can be eliminated (5), since `X` must be a *sign* in this context, *synsem* is the least appropriate type of any SYNSEM value, *local* is the least appropriate type of any LOC value, and *category* is the least appropriate type of any CAT value.

3.4 Primitive delay statements

The two fundamental primitives typically provided for Prolog terms, e.g., by SICStus Prolog `when/2`, are: (1) suspending until a variable is instantiated, and (2) suspending until two variables are equated or inequated. The latter corresponds exactly to structure-sharing in TFSs, and to shared variables in descriptions; its implementation was already discussed in the previous section. The former, if carried over directly, would correspond to delaying until a variable is promoted to a type more specific than $\perp$, the most general type in the type semilattice. There are degrees of instantiation in LTFS, however, corresponding to long subtyping chains that terminate in $\perp$. A more general and useful primitive in a typed language with such chains is suspending until a variable is promoted to a particular type. `whentype(Type,X,Goal)`, i.e., delaying subgoal `Goal` until variable `X` reaches `Type`, is then the non-universally-quantified cousin of the type-antecedent constraints that are already used in the ERG.

How `whentype(Type,X,Goal)` is implemented depends on the data structure used for TFSs, but in Prolog they invariably use the underlying Prolog implementation of `when/2`. In ALE, for example, TFSs are represented with reference chains that extend every time their type changes. One can simply wait for a variable position at the end of this chain to be instantiated, and then compare the new type to `Type`. Figure 3 (6) shows a schematic representation of a *sign*-typed TFS with SYNSEM value `SynVal`, and two other appropriate feature values. Acting upon this as its second argument, the corresponding definition of `whentype(Type,X,Goal)` in Figure 3 (7) delays on the variable in the extra, fourth argument position. This variable will be instantiated to a similar term when this TFS promotes to a subtype of *sign*.

As described above, delaying until the antecedent of the principle in Figure 3 (1) is true or false ultimately reduces to delaying until various feature values attain certain types using `whentype/3`. A TFS may not have substructures that are specific enough to determine whether an antecedent holds or not. In this case, we must wait until it is known whether the antecedent is true or false before applying the consequent. If we reach a deadlock, where several constraints are suspended on their antecedents, then we must use another resolution method to begin testing more specific extensions of the TFS in turn. The choice of these other methods characterizes a true CLP solution for LTFS, all of which are enabled by the method presented in this paper. In the case of the signature in Figure 4, one of these methods may test whether a *marking*-typed substructure is consistent with either *fin* or *inf*. If it is consistent with *fin*, then this branch of the search may unsuspend the Finiteness Marking Principle on a *sign*-typed TFS that contains this substructure.

4 Measuring the cost of delaying

How much of a cost do we pay for using delaying? In order to answer this question definitively, we would need to reimplement a large-scale grammar which was substantially identical in every way to the ERG but for its use of delay statements. The construction of such a grammar is outside the scope of this research programme, but we do have access to MERGE,[2] which was designed to have the same extensional coverage of English as the ERG. Internally, the MERGE is quite unlike the ERG. Its TFSs are far larger because each TFS category carries inside it the phrase structure daughters of the rule that created it. It also has far fewer types, more feature values, a heavy reliance on lists, about a third as many phrase structure rules with daughter categories that are an average of 32% larger, and many more constraints. Because of these differences, this version of MERGE runs on average about 300 times slower than the ERG.

On the other hand, MERGE uses delaying for all three of the purposes that have been discussed in this paper: complex antecedents, explicit `whenfs/2` calls to avoid non-termination problems, and explicit `whenfs/2` calls to avoid expensive non-deterministic searches. While there is currently no delay-free grammar to compare it to, we can pop open the hood on our implementation and measure delaying relative to other system functions on MERGE with its test suite. The results are shown in Figure 5. These results show that while the per call

Function	avg. μs / call	avg. # calls	per sent. avg. % parse time
PS rules	1458	410	0.41
Chart access	13.3	13426	0.12
Relations	4.0	1380288	1.88
Delays	2.6	3633406	6.38
Path compression	2.0	955391	1.31
Constraints	1.6	1530779	1.62
Unifi cation	1.5	37187128	38.77
Dereferencing	0.5	116731777	38.44
Add type MGSat	0.3	5131391	0.97
Retrieve feat. val.	0.02	19617973	0.21

Figure 5: Run-time allocation of functionality in MERGE. Times were measured on an HP Omnibook XE3 laptop with an 850MHz Pentium II processor and 512MB of RAM, running SICStus Prolog 3.11.0 on Windows 98 SE.

cost of delaying is on a par with other system functions such as constraint enforcement and relational goal resolution, delaying takes between three and five times more of the percentage of sentence parse time because it is called so often. This reflects, in part, design decisions of the MERGE grammar writers, but it also underscores the importance of having an efficient implementation of delaying for large-scale use. Even if delaying could be eliminated entirely from this grammar at no cost, however, a 6% reduction in parsing speed would not, in the present author's view, warrant the loss of modularity in a grammar of this size.

[2]The author sincerely thanks Kordula DeKuthy and Detmar Meurers for their assistance in providing the version of MERGE (0.9.6) and its test suite (1347 sentences, average word length 6.3, average chart size 410 edges) for this evaluation. MERGE is still under development.

5 Conclusion

It has been shown that a simple generalization of conventional delay statements to LTFS, combined with a subsumption-based interpretation of implicational constraints and unique feature introduction are sufficient to restore much of the functionality and concomitant benefit that has been routinely sacrificed in HPSG in the name of parsing efficiency. While a definitive measurement of the computational cost of this functionality has yet to emerge, there is at least no apparent indication from the experiments that we can conduct that disjunction, complex antecedents and/or a judicious use of recursion pose a significant obstacle to tractable grammar design when the right control strategy (CLP with subsumption testing) is adopted.

References

H. Aït-Kaci, A. Podelski, and G. Smolka. 1992. A feature-based constraint system for logic programming with entailment. In *Proceedings of the International Conference on Fifth Generation Computer Systems*.

H. Aït-Kaći. 1984. *A Lattice-theoretic Approach to Computation based on a Calculus of Partially Ordered Type Structures*. Ph.D. thesis, University of Pennsylvania.

B. Carpenter and G. Penn. 1996. Compiling typed attribute-value logic grammars. In H. Bunt and M. Tomita, editors, *Recent Advances in Parsing Technologies*, pages 145–168. Kluwer.

B. Carpenter. 1992. *The Logic of Typed Feature Structures*. Cambridge.

A. Copestake, D. Flickinger, C. Pollard, and I. Sag. 2003. Minimal Recursion Semantics: An introduction. Journal submission, November 2003.

J. Doerre, M. Dorna, J. Junger, and K. Schneider, 1996. *The CUF User's Manual*. IMS Stuttgart, 2.0 edition.

J. Doerre. 1993. Generalizing Earley deduction for constraint-based grammars. Technical Report R1.2.A, DYANA Deliverable.

D. Flickinger. 2000. On building a more efficient grammar by exploiting types. *Natural Language Engineering*, 6(1):15–28.

T. Goetz and W.D. Meurers. 1997. Interleaving universal principles and relational constraints over typed feature logic. In *Proceedings of the 35th ACL / 8th EACL*, pages 1–8.

M. Höhfeld and G. Smolka. 1988. Definite relations over constraint languages. LILOG Report 53, IBM Deutschland.

H.-U. Krieger and J. Nerbone. 1991. Feature-based inheritance networks for computational lexicons. In *Proceedings of the ACQUILEX Workshop on Default Inheritance in the Lexicon*, number 238 in University of Cambridge, Computer Laboratory Technical Report.

H.-U. Krieger and U. Schäfer. 1994. TDL — a type description language for HPSG part 1: Overview. Technical Report RR-94-37, Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), November.

LinGO. 1999. The LinGO grammar and lexicon. Available on-line at http://lingo.stanford.edu.

G. Penn and K. Hoetmer. 2003. In search of epistemic primitives in the english resource grammar. In *Proceedings of the 10th International Conference on Head-driven Phrase Structure Grammar*, pages 318–337.

G. Penn. 2001. Tractability and structural closures in attribute logic signatures. In *Proceedings of the 39th ACL*, pages 410–417.

C. J. Pollard. 1998. Personal communiciation to the author.

G. Smolka. 1991. Residuation and guarded rules for constraint logic programming. Technical Report RR-91-13, DFKI.

G. Smolka. 1994. A calculus for higher-order concurrent constraint programming with deep guards. Technical Report RR-94-03, DFKI.

P. Van Roy. 1990. *Can Logic Programming Execute as Fast as Imperative Programming?* Ph.D. thesis, University of California, Berkeley.

S. Wintner. 2002. Modular context-free grammars. *Grammars*, 5(1):41–63.