AVM Description Compilation using Types as Modes

Gerald Penn
Department of Computer Science
University of Toronto
gpenn@cs.toronto.edu

Abstract

This paper provides a method for generating compact and efficient code to implement the enforcement of a description in typed feature logic. It does so by viewing information about types through the course of code generation as modes of instantiation — a generalization of the common practice in logic programming of the binary instantiated/variable mode declarations that advanced Prolog compilers use. Section 1 introduces the description language. Sections 2 and 3 motivate the view of mode and compilation taken here, and outline a mode declaration language for typed feature logic. Sections 4 through 7 then present the compiler. An evaluation on two grammars is presented at the end.

1 Descriptions

The logic of typed feature structures (Carpenter, 1992) has been widely used as a means of formalizing and developing natural language grammars, notably in Head-driven Phrase Structure Grammar (Pollard and Sag, 1994). These grammars are stated using a vocabulary consisting of a finite meet semi-lattice of types and a set of features that must be specified for each grammar, and this vocabulary must obey certain rules. A set of *appropriateness conditions* must specify, for each feature, which types of feature structures may bear it, and which types of values it may take. *Unique feature introduction* states that every feature has a least type that bears it, called its *introducer*. The effect of these rules is that typed feature structures (TFSs) can be described using a very terse description language. A TFS that matches the description NUMBER : $singular$, for example, might implicitly be of a type *index*, which introduces NUMBER. From that, we can deduce that the TFS also bears values for PERSON and GENDER, with particular appropriate values, because those features are introduced by the same type. Terse descriptions allow us to work with very large TFSs conveniently.

Given this basic vocabulary, descriptions can be used in implicational constraints that encode principles of grammar. These are often restricted to the form $\tau \rightarrow \phi$, where τ is a type, and ϕ is a description. They also appear in several control strategies that are used to combine TFSs, such as extended phrase structure rules for parsing or generation (Wintner, 1997; Malouf et al., 2000), or resolution with a Prolog-like relational language (Carpenter and Penn, 1996; Makino et al., 1998).

While description languages vary, they are usually restricted to a subset of the following:

Definition: *The set of* descriptions *over a countable set of types, T, a finite set of features, Feat, and a countable set of variables, Var, is the smallest set Desc that contains:*

- Var,
- $\doteq X$, *all* $X \in Var$,
- T,
- $\pi : \phi$, *all* $\pi \in Feat^*, \phi \in Desc$, *and*
- $\phi \wedge \psi, \phi \vee \psi$, *all* $\phi, \psi \in Desc$.

Variables are used to enforce sharing of structure among substructures, and are often used with

wider scope than a single TFS to pass structure — for example, from daughter categories to mother categories in phrase structure rules or between arguments in definite clause relations. With inequations ($\not\doteq X$), they can also be used to prohibit structure sharing. Preceding a description with a feature path, π, causes that description to be enforced on the substructure at the end of that path.

2 Types as Modes

What makes descriptions so terse, and what distinguishes the logic of typed feature structures from more general order-sorted terms or record structures, is their strong notion of typing. Types introduce features, types determine the arity (number of features) of a TFS, types are the antecedents to grammar constraints and types are what normally determine unifiability through the existence of least upper bounds.

A similar situation exists in Prolog. The functor and arity of a Prolog term characterize the number of arguments, the interpretation of the argument positions (which, unlike TFSs, are unnamed), and the unifiability of the term with other Prolog terms. In Prolog, however, a term has one of two degrees of instantiation — either its functor/arity is not known (variable), or it is (instantiated term). Advanced Prolog compilers use this information where available to generate faster code for both kinds of arguments. This is called *mode*, because the degree of instantiation often corresponds to whether an argument is being used operationally as "input" or "output" to a Prolog predicate.

To date, TFS-based parsers and logic programming systems have relied on the potential terseness of descriptions for efficiency in enforcing them. It is possible, however, to use information about types derived from appropriateness and knowledge of flow control in grammars/programs as a kind of mode, only now with many more shades of distinction. Figure 1 shows a sample meet semi-lattice annotated with appropriateness conditions. Features are shown annotating their introducers, with the understanding that all subtypes of the introducer bear those features as well. Along with the feature name appears the type to which the feature's value is restricted, called a *value restriction*. The most general type, $\bot$ ("bottom"), corresponds to an uninstantiated variable, but then there are potentially many degrees of successive refinement. A $\bot$-typed TFS could be instantiated to a *head*-typed TFS, for example, at which point we know it has two "arguments" for the features PRD and MOD. But then it could be instantiated further to a *subst*, and even further still to a *noun*, at which point its arity increases by one to accommodate the feature CASE. By contrast, Prolog terms always appear in the same very flat semi-lattice (Figure 2), in which what little inheritance there is is determined structurally rather than from named declarations.

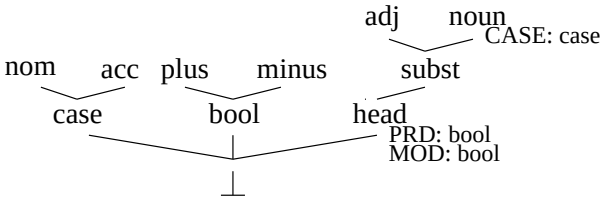

Figure 1: A sample type semi-lattice.

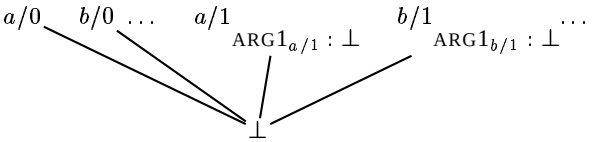

Figure 2: A type semi-lattice for Prolog terms.

Just as with Prolog, it is also possible to augment what we can statically infer from programs with user-provided *mode declarations*. Even with compilers that do not use them, mode declarations are often recommended because they make programs more readable. In the case of typed feature logic, the correspondence to true input/output mode is not as direct, and is probably better distinguished from mode in its typed sense. With definite clauses over TFSs, for example, one would expect to provide an input type and output type for each argument in the general case:

```
:- mode append(list-list,list-list,
                bot-list).
```

This declaration says that `append/3` is called with its first two arguments instantiated to type *list*, and a third argument that might not be more instantiated than $\bot$. Input is given by the first type in each pair. Output is given by the second type — in this case, all *list*-instantiated.

Another possibility is to interpret mode as a promise that a TFS will be no more instantiated than, but still consistent with the type given. This corresponds to viewing declared modes as greatest elements in principal ideals of the type semilattice, rather than as least elements of principal filters. While both views could be exploited by a compiler, unification promotes the type of a TFS within its principal filter, so we would expect to be able to provide useful mode information more often with the filter view. With `append/3`, for example, the first argument could be a *list*, or one of its subtypes, *ne-list* and *e-list*, corresponding to non-empty and empty lists, respectively.[1]

These declarations will not be discussed further here, as they simply serve to seed the description compilation process below with initial mode values. What is more interesting is how mode is used and updated within a description.

3 Descriptions as Compilable Objects

One way to determine whether a given TFS matches a description is to find the description's *most general satisfier*, i.e., the least informative TFS in its denotation, and unify it with the given TFS. This is typically too slow. TFS-unification combines corresponding substructures without making use of the fact that one of them is a most general satisfier. By only combining those parts of the TFSs that are explicitly mentioned in the description plus the types that can be inferred from appropriateness, we avoid a great deal of unnecessary work. TFS-unification must also enumerate all possible most general satisfiers in the case of disjunctive descriptions, but using the description as a guide exploits the naturally localized non-determinism that users often provide, thereby avoiding redundant unifications (Carpenter and Penn, 1996).

There is a limit, however, to how terse a description can become. In part, this is due to the fact that more verbose descriptions may be more readable. With very large descriptions, in fact, it may be quite difficult to find an optimally terse equivalent. But even the description language itself imposes limits. With respect to Figure 1, for example, a purely compositional treatment of the conjunctive description $\textsc{prd} : plus \wedge \textsc{mod} : plus$ would enforce the requirement that the TFS is of type *head* in both conjuncts. Once a given serialization of this description's operations is chosen, the mode of the candidate TFS can be tracked internally to generate more efficient code. In addition, descriptions with shared variables, such as $\textsc{prd} : X \wedge \textsc{mod} : X$ can transmit mode information through their variables from one substructure to another. Tracking the modes of variables can be particularly useful in languages where the scope of variables extends over multiple TFSs, such as the categories of a phrase structure rule — nothing in appropriateness can provide a substitute.

[1] As useful idioms, one should also provide the declaration `type` or `+type` as shorthand for `type-type`, and `-type` for `bot-type`.

4 Parameters of Compiler

Compiling descriptions is already more efficient than unifying with most general satisfiers, but tracking mode (type) with both substructures and variables makes it even more so. The compiler described here uses Prolog as the target language, but it should be obvious that any other internal representation that supports unification with partial templates of TFSs and the atomic operations used below will suffice.

Our compiler requires six **inputs**: **(1)** the description to be compiled, **(2)** a *TFS pointer* that points at run-time to the TFS that the description is being enforced on, **(3)** the input mode of TFS, **(4)** the input modes of previously seen variables, **(5)** a description variable binding flag (`CBSafe`), and **(6)** a template substitution flag (`TemplSubst`). If no input mode information is available for the TFS at compile-time, the mode is taken to be $\perp$. Previously unseen variables also have $\perp$ as their input mode. Seen variables are those variables that *may* have been bound earlier in the code — because of disjunctive descriptions, it not always possible to say with certainty. `CBSafe` tells us when it is safe to bind a description variable at compile-time. This is generally true, with the exceptions being compilation of disjunctive descriptions, in which the same variable used in one disjunct should not be bound by the other disjunct, and compilation of co-routined predicates (such as SICStus `when/2`), in which binding a description

variable at compile-time in the delayed body could transmit structure back into the guard.

The second flag, `TemplSubst`, tells us whether it is safe at compile-time to instantiate the Prolog variable that implements the TFS pointer[2] passed to the description compiler. Normally, this should be true. With some kinds of description compilation, such as type-antecedent implicational constraints, propagating templates from one constraint into another at compile-time can often create compile-time unification failures. Without the right support for tracing where these failures occur, they are better left to a run-time system.

Templates for us will be partial Prolog term encodings of TFSs. In general they are *partial* representations of TFSs, in that by themselves they may not look well-typed or even well-formed — certain feature values might be missing or occur with invalid types. In the context of a given input mode, however, they will still be sufficient for unifying with particular kinds of TFSs, and more efficient than their well-formed, well-typed counterparts. The **outputs** of our compiler are then: **(1)** the TFS pointer, which now possibly points to a template, **(2)** output modes for that pointer and any variables seen so far, and **(3)** a stream of code that should be executed at run-time on the TFS pointed to by the TFS pointer, after it has been unified with its template, if any.

In constructs for which variables have wider scope than a single description, it is assumed that descriptions will be compiled in the order established by the model of control flow for that construct. Compilation of a phrase structure rule in a left-to-right bottom-up parser, for example, would compile the daughter descriptions from left to right, and finally the mother description. This allows variable modes to be threaded from one description to the next.

5 Pre-processing

Before compilation proper begins, we must convert the input description to a canonical form. The major stages of pre-processing are given in Figure 3. In the example shown, a list macro is expanded to its syntactic form as a TFS description, but other user-defined macros may exist too. Conjunction of descriptions translates during compilation into continuations of code, so wherever possible, we would like to represent descriptions as streams of (conjoined) subdescriptions that will directly correspond to the streams of code we generate. Empirically, one can also observe that descriptions tend to be highly conjunctive, so this step can save quite a bit of time later. This associative reshuffling is disrupted by disjunctive, inequational or path descriptions, whose internal subdescriptions are also reshuffled but not combined with their external contexts (that is, nothing like DNF is computed).

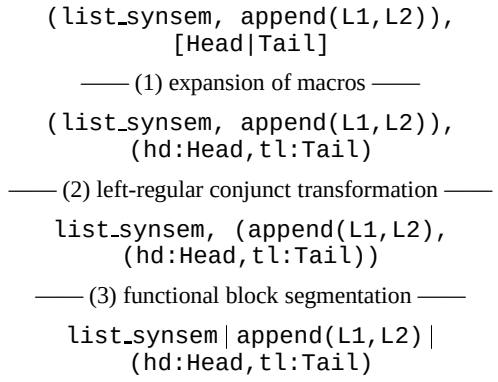

Figure 3: Stages of Pre-processing.

Some description languages also have functional or procedural extensions that extend beyond the power of the normal description language and may even perform I/O operations. Since our compiler will re-order many components of the input description, it is important to preserve the order in which these extensions occur relative to other components of the description. So the third step of pre-processing divides a description into functional or non-functional blocks. Non-functional blocks will be re-ordered internally, and mode will be transmitted from one block to the next, but the order of the blocks themselves must be preserved in the generated code. Functional block compilation is outside the scope of this paper.

The result of block compilation, apart from an output mode, is one of three things:

[2]Note that because we are compiling a description language with variables into a target language (Prolog) with variables, we need to distinguish the two kinds. We will refer to the target-language variables as pointers, since this is how they are realized internally in the Warren Abstract Machine.

`compile_desc(Desc,FS,CBSafe,TemplSubst,Mode)`
Compile the blocks:

1. if the result is `empty`, return `Code-Code`.
2. if the result is `template(Template,TCode,Rest)`:
   (a) if `TemplSubst` is `true`, then bind the TFS pointer, `FS=Template` and return `TCode-Rest`
   (b) else return `[FS=Template|TCode]-Rest`.
3. if the result is `var(FS,Code,Rest)`, then return `Code-Rest`.
4. if the result is failure, then return `fail-Rest` with an output mode of `top` ($\top$).

Figure 4: The top-level description compiler.

`empty`, meaning that the description is redundant with the given input mode, `template(Templ,Code,Rest)`, meaning there is a template with code stream `Code-Rest` encoded as a difference list, and `var(Ptr,Code,Rest)`, meaning there is no template — only the TFS pointer. Top-level description compilation is shown in Figure 4. Depending on the value of the `TemplSubst` flag, the TFS pointer, `FS`, is instantiated to a template, if one exists, either at compile-time, or at run-time by adding a unify instruction to the front of the code stream. The blocks compiler threads together code from individual blocks in a similar fashion given the results on each block.

6 Block Compilation

The first step of block compilation is then to **partition** top-level conjuncts into their kind of description. These classes are then compiled separately, and their code is threaded together in this **order**: **(1)** seen variables, **(2)** type, **(3)** unseen variables, **(4)** feature descriptions, **(5)** inequation descriptions, and **(6)** disjunctive descriptions. The order has been chosen according to the extent to which each class can deterministically and quickly produce a failure. Those that potentially can very well, such as types, occur early in the thread in order to avoid wasting time on the code of the others. This is admittedly rather naive, since unification with a seen variable or a type could result in applying a highly disjunctive constraint that is not considered here. In addition, small non-disjunctive feature descriptions could be much quicker than unification with a large TFS to which a seen variable is instantiated. The right way to determine this order is through empirical estimation of the likelihood of failure. This problem, in fact, is closely related to the general indexing problem for TFSs and to that of quick-check paths for early detection of unification failure (Malouf et al., 2000).

The last three classes have subdescriptions. These are recursively block-segmented, preprocessed and compiled using the same partitioning method. Nothing more needs to be said about the inequation and disjunctive classes.

The type in the type class is computed by unifying: (1) all of the type descriptions in the current block, (2) all of the introducers of the feature descriptions in the block, and (3) the meet (greatest lower bound) in each disjunctive description of the type class of each of its disjuncts. In other words, we are extracting as much type information as we deterministically can from all but one class in the current block and applying it first. We do not include the input modes of variables because unification will add the type information in these automatically, as well as enforce any necessary implicational constraints. The type class contains the type information that we may need to enforce constraints on ourselves.

The output mode of the TFS pointer is this type unified with the input mode and the input modes of the seen variables. Note that while code for the type class is among the first to be executed at run-time, it must be the last generated at compile-time.

Variables (seen or unseen) either bind to the TFS pointer at compile-time or generate code to bind the variable at run-time according to the value of `CBSafe`. Variable modes are updated by unification with the TFS pointer's output mode.

When feature descriptions are collected in a block, they are also further partitioned by the first feature in their paths and compiled together so that the value at that feature only needs to be dereferenced once (at least for feature descriptions). Conveniently enough, appropriateness provides us with an input mode for our recursive compilation of a feature's descriptions — namely the value restriction of that feature at the TFS pointer's output mode. This is everything we will know about

this feature's value after executing the code for the type class.

7 Combining Class Information

The last step of block compilation is then to combine the results of compiling each class to generate the minimal amount of code. There are three cases to consider here, which refer both to the output of class compilation and to a relation defined by where implicational constraints occur in the type semi-lattice. That relation is defined as follows:

Definition: *A type, τ, is* relatively constrained *by a mode μ iff there exists a $\sigma \sqsupseteq \mu$ which is unifiable with τ and a ρ such that $\rho \sqsubseteq \tau \sqcup \sigma$, $\rho \not\sqsubseteq \sigma$, and there is an implicational constraint $\rho \rightarrow \phi$ for some description ϕ.*

If we add τ to a TFS pointer with input mode μ, then under the filter view of mode, we might need to enforce some constraint on the resulting TFS if τ is relatively constrained by μ. Since seen variables are unified before type class information is added, we can actually take μ to be the input mode unified with the input modes of the seen variables instead. This will possibly result in a smaller filter, and therefore possibly fewer constraints to add.

Case 1: $\tau \sqsubseteq \mu$ and all feature descriptions compile to `empty`. In this case, the type and features add no new information. If no run-time code is added by variables, inequations or disjunctions either, then we can return `empty`. Otherwise, we can thread the code together, and return `var(Ptr,Code,Rest)`, where `Ptr` is the current TFS pointer.

Case 2: τ is relatively constrained by μ. In this case, we must add the type using code, so that constraints can be enforced if necessary at run-time. This is actually the case we want to avoid, but unless $\rho = \tau$, there is no guarantee that any particular constraint will definitely be required, so nothing else should be done at compile-time. All TFS-based abstract machines possess a primitive operation for this. In the Carpenter-Qu machine (implemented in LiLFeS (Makino et al., 1998)), it is `addtype`. AMALIA (Wintner, 1997) and ALE (Carpenter and Penn, 1996) have something similar. This instruction must then be threaded with the code generated by other classes.

Case 3: Otherwise. In this case, we should try to do as much work as possible in a template. If feature descriptions, inequation descriptions and disjunctive descriptions generate no code, then we can do all of the work in a template. If they do generate code, then we can still build a template, but we should instantiate the TFS pointer with it at run-time to avoid propagating the template's structure through the generated code at compile-time. This would result in an unnecessary amount of extra structure in the code stream, and therefore extra work at run-time.

In Prolog, our templates look like terms `f(Type,F1,...,Fn)`, where `Type` is a term encoding of the type, given by one of several first-order encodings available for meet semi-lattices (Mellish, 1992; Penn, 2000), and the `Fi` are encodings of appropriate features. `f` is determined by a modularization algorithm that breaks a semi-lattice into inconsistent pieces that can be encoded separately. `n`, the number of feature positions, is determined by a graph coloring algorithm on that module that finds the minimum number of required positions (Penn, 2000). Unused positions are left as variables, and in general a description compiler must be prepared to fill those variables in once a TFS becomes specific enough to employ them (Penn, 2002). TFS-based abstract machines can construct a similar fixed-width encoding.

The term encoding of τ should fill the `Type` position. It only needs to be filled with those parts of the term encoding that the encoding of μ lacks, however. Filling the type position is not the only requirement for adding a type with a template, however. When we add a type, new features may be introduced and value restrictions may be refined, requiring extra structure to be added in an argument position. At the same time, feature descriptions may demand to use those positions as TFS pointers for compiled code. The algorithm for combining these requirements is given below. We initially call the algorithm by first allocating our *Template*, and then calling `build_template`(τ,μ,ϵ,*Template*), where ϵ is the empty path.

`build_template`(τ,μ,π,*Template*)
$\mathrm{TypePos}(\mathit{Template}) := \mathrm{TEncoding}(\tau, \mu)$
For each feature F appropriate to τ:

1. If $\mathrm{VR}(F, \tau) \sqsubseteq \mathrm{VR}(F, \mu)$ or $\mathrm{VR}(F, \tau) \sqsubseteq$

$\mathrm{VR}(F, \mathrm{Intro}(F))$:

(a) If $\mathrm{FRes}(\pi, F) =$`empty`, then continue.
(b) If $\mathrm{FRes}(\pi, F) =$`template(FTem,Code,Rest)`:
- $\mathrm{Pos}(\mathit{Template}, F) :=$`FTem`,
- add `Code-Rest` to code stream,
- and continue.

(c) otherwise $\mathrm{FRes}(\pi, F) =$`var(FV,Code,Rest)`:
- $\mathrm{Pos}(\mathit{Template}, F) :=$`FV`,
- add `Code-Rest` to code stream,
- and continue.

2. If $\mathrm{VR}(F, \tau)$ is relatively constrained by $\mathrm{VR}(F, \mu)$:
add an `addtype(`$\mathrm{VR}(F, \tau)$`)` instruction to the code stream,

(a) If $\mathrm{FRes}(\pi, F) =$`empty`, then continue.
(b) If $\mathrm{FRes}(\pi, F) =$`template(FTem,Code,Rest)`:
- $\mathrm{Pos}(\mathit{Template}, F) :=$`FV`,
- add `[FV=FTem|Code]-Rest` to the code stream,
- and continue.

(c) otherwise $\mathrm{FRes}(\pi, F) =$`var(FV,Code,Rest)`:
- $\mathrm{Pos}(\mathit{Template}, F) :=$`FV`,
- add `Code-Rest` to the code stream,
- and continue.

3. otherwise:
allocate a new template, *FTemplate*,

(a) If $\mathrm{FRes}(\pi, F) =$`empty`, then $\mathrm{Pos}(\mathit{Template}, F) :=$*FTemplate*,
(b) If $\mathrm{FRes}(\pi, F) =$`template(FTem,Code,Rest)`:
- $\mathrm{Pos}(\mathit{Template}, F) :=$`FV`,
- call `build_template`($\mathrm{VR}(F, \tau)$,$\mathrm{VR}(F, \mu)$, $\pi : F$,*FTemplate*), which returns code stream `FCode-FRest`,
- add `[FV=`*`FTemplate`*`|FCode]-FRest` to the code stream,
- add `[FV=FTem|Code]-Rest` to the code stream,
- and continue.

(c) otherwise $\mathrm{FRes}(\pi, F) =$`var(FV,Code,Rest)`:
- $\mathrm{Pos}(\mathit{Template}, F) :=$`FV`,
- call `build_template`($\mathrm{VR}(F, \tau)$,$\mathrm{VR}(F, \mu)$, $\pi : F$,*FTemplate*), which returns code stream `FCode-FRest`,
- add `[FV=`*`FTemplate`*`|FCode]-FRest` to the code stream,
- and `Code-Rest` to the code stream,
- and continue.

Return the code stream.

Here, $\mathrm{FRes}(\pi, F)$ is the result of compiling the feature descriptions at path $\pi : F$, $\mathrm{VR}(F, t)$ is the value restriction of feature F at type t, $\mathrm{Intro}(F)$ is the introducer of F, $\mathrm{TEncoding}(\tau, \mu)$ is the portion of the term encoding of τ that the term encoding of μ lacks, $\mathrm{TypePos}(\mathit{Template})$ is the position of the type encoding in *Template*, and $\mathrm{Pos}(\mathit{Template}, F)$ is the position for F in *Template*. The first case determines that the added type does not need the position for F, and allows the feature descriptions to use it. The second case determines that the added type does need the position for F as a variable so that it can make an `addtype` call in the code stream to add a constraint. That means that we can use a template in that position, but we must instantiate it at run-time, since otherwise the template would propagate through the run-time code. The third case determines that the added type needs the position for adding appropriateness information. This appropriateness information is added using a new template, *FTemplate*, in each feature position that requires it. A recursive call is also made to fill in the feature positions of *FTemplate*.

8 Evaluation

To test the effectiveness of this mode-sensitive compiler, it was implemented to replace ALE 3.3's compiler and measured against the old one on parsing with two grammars. All other components of the system were identical. Neither grammar has mode declarations for relations, so only the benefit of internal mode tracking is measured here. The first grammar is a version of the original English Resource Grammar (ERG, 1999), ported to ALE and modified by Kordula DeKuthy and Detmar Meurers of Ohio State University to use fewer types. Compilation times with the two compilers were roughly the same. The generated code was run over a test suite distributed with this version of 61 sentences, 41 of them grammatical.[3] For both versions of code, the suite was parsed in order 5 times and the times for each sentence were averaged to control for variations in system load. Those averages are plotted in Figure 5. Sentences are ordered by average parse time with the new mode-sensitive compiler. The code generated by this compiler parses sentences in this test suite on an average of 41.8% less time, with a minimum of 36.5% less, and a maximum of 50.7%. On an absolute basis, the new compiler's code parses an average of 541 ms faster, with a minimum of 34 ms and a maximum of 9082 ms.

The second grammar is the HPSG gram-

[3]The small size of the suite is due to the fact that only a small part of the ERG lexicon had been converted as of this writing.

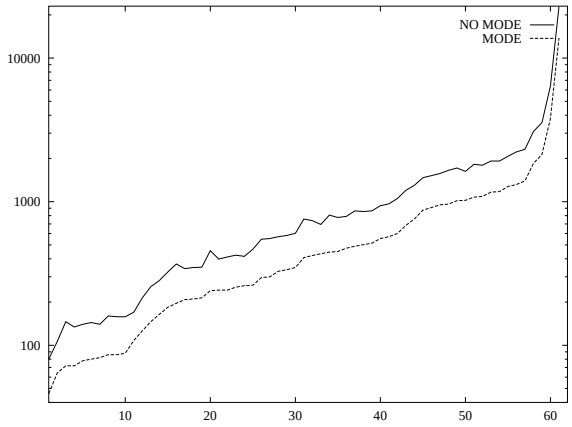

Figure 5: Average parse times (in ms) for ALE compiled code with and without modes on the ERG test suite.

mar distributed with ALE. This grammar and was designed to be a literal translation of Pollard and Sag (1994). It has many more relations, applies more descriptions at run-time, and is far less efficient. It was benchmarked on 274 sentences randomly selected from a larger test suite that was automatically generated by a context-free grammar with the same lexicon. The grammar found 80% of the sample to be grammatical. The same protocol as above was used. The results are shown in Figure 6. Here, the average reduction

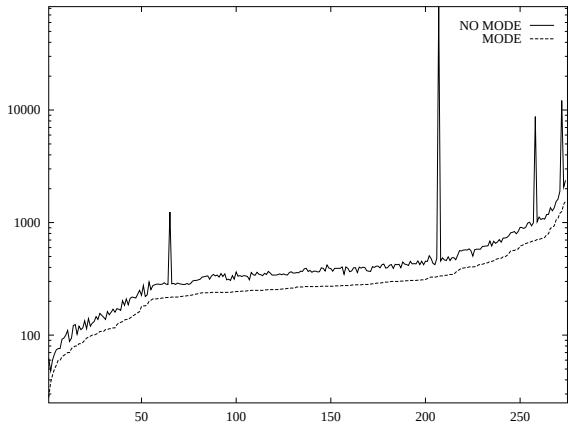

Figure 6: Average parse times (in ms) for ALE compiled code with and without modes on an HPSG test suite.

in parse time is 28.9%, with a minimum of 17.3% and a maximum of 99.6%. The average absolute reduction was 497 ms with a minimum of 10 ms, and a maximum of 81,952 ms. The smaller average reduction and larger variability are both likely due to the greater modularity of the grammar.

9 Conclusion

Using modes plus reordering description contents has been shown to measurably improve parse times on two grammars. The best way to find a reordering is of course to profile the input grammar on representative data to gather such information empirically.

References

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.

ERG. 1999. The English Resource Grammar and lexicon. Available on-line at **http://lingo.stanford.edu**.

T. Makino, K. Torisawa, and J. Tsuji. 1998. LiLFeS — practical unification-based programming system for typed feature structures. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and the 17th International Conference on Computational Linguistics (COLING/ACL-98)*, volume 2, pages 807–811.

R. Malouf, J. Carroll, and A. Copestake. 2000. Efficient feature structure operations without compilation. *Natural Language Engineering*, 6(1):29–46.

C. Mellish. 1992. Term-encodable description spaces. In D.R. Brough, editor, *Logic Programming: New Frontiers*, pages 189–207. Kluwer.

G. Penn. 2000. *The Algebraic Structure of Attributed Type Signatures*. Ph.D. thesis, Carnegie Mellon University.

G. Penn. 2002. Generalized encoding of description spaces and its application to typed feature structures. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL-02)*, pages 64–71.

C. Pollard and I. Sag. 1994. *Head-driven Phrase Structure Grammar*. Chicago.

S. Wintner. 1997. *An Abstract Machine for Unification Grammars with Applications to an HPSG Grammar for Hebrew*. Ph.D. thesis, Technion.