The ConTroll Manual

(ConTroll v.1.0β, XTroll v.5.0β)

Thilo Götz Detmar Meurers
Dale Gerdemann

Draft of September 17, 1997

Universität Tübingen
Seminar für Sprachwissenschaft
Abt. Computerlinguistik
Kleine Wilhelmstr. 113
72074 Tübingen
Germany

controll-support@sfs.nphil.uni-tuebingen.de

Contents

Chapter 1

Preface

The ConTroll system was developed in the B4 project of the SFB 340, funded by the Deutsche Forschungsgemeinschaft (DFG). The following people contributed to the development of the system: Thilo Götz, Detmar Meurers, John Griffith, Dale Gerdemann, Erhard Hinrichs, Guido Minnen, Gerald Penn, Björn Aldag, Natali Alt, Stephan Kepser, Dieter Martini, Oliver Suhre and Andreas Zahnert. The Xtroll Interface was programmed by Carsten Hess. The system was (and still is) used in the B8 project for the development of a number of grammars, including a large German fragment. The following people were or are involved in the B8 project and provided feedback which significantly improved the system: Elisabeth Courbet, Kordula De Kuthy, Anke Feldhaus, Detmar Meurers, Lucia Reinhard, Frank Richter, Manfred Sailer.

The sytem developed out of the Troll system (Gerdemann, Götz, Griffith, Kepser, and Morawietz 1995). Troll is a phrase structure backbone system very similar to ALE (Carpenter and Penn 1994), but it differs from that system in that it is based on the set theoretic logic of (King 1989, 1994) rather than the information theoretic one of (Carpenter 1992). The practical effect of this is that ConTroll implements an exhaustive typing strategy which provides the stronger kind of inferencing over descriptions (Gerdemann and King 1993, 1994) required by standard HPSG theories.

System availability and ConTroll homepage For people and institutions which make all their research results public, the ConTroll system is available for non-profit use, free of charge, from

`http://www.sfs.nphil.uni-tuebingen.de/controll/`

On this page you'll also find links to ConTroll related publications.

Mailing list There is a mailing list to discuss theoretical and practical issues related to the ConTroll system. To un/subscribe from or obtain help on using the list, send an email with "(un)subscribe" or "help" as subject to

`controll-request@sfs.nphil.uni-tuebingen.de`

To post to the list, please send email to

`controll@sfs.nphil.uni-tuebingen.de`

Finally, to reach the people developing ConTroll directly you can send email to

`controll-support@sfs.nphil.uni-tuebingen.de`

Chapter 2

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 generalizations 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.

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

[2] ALE also has a restricted form of universal constraints.

Our implementation is based on the idea of compiling 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.

While in this manual we will focus on the practical aspects of the ConTroll system, a number of papers discuss the theoretical background of the different parts of the system:

- ConTroll in general: Götz and Meurers (1997a, 1997b),
- Formal foundations: King (1989, 1994), Gerdemann and King (1993, 1994), Meurers (1994)
- Compilation methods: Götz and Meurers (1995, 1996)
- Lexical Rules[3]
  - computational: Meurers and Minnen (1995, 1996, 1997)
  - formal basis: Meurers (1995), Calcagno, Meurers, and Pollard (prep)

In this manual, chapter 3 starts with a very basic introduction to writing grammars in ConTroll for a quick start. The following chapters then provide a more systematic discussion. In chapter 4 we focus on the ontology of the linguistic objects which are intended to be modeled and discuss how the corresponding grammar datastructure can be defined in ConTroll. Chapter 5 then introduces the description language used to refer to these linguistic objects when writing a grammar. In chapter 6 we then introduce the two kinds of grammar constraints available in ConTroll, implicational and relational constraints and discuss how their processing can be interleaved in processing. Having introduced the necessary building blocks of a grammar, chapter 7 then provides more information on the compilation process, the interpreter and the debugger. For more complex grammars, ConTroll provides a more complex file organization, the specification of which is introduced in chapter 8. Finally, the graphical interface XTroll is introduced in chapter 9.

[3]The lexical rule compiler is not yet included in this β-release of ConTroll.

Chapter 3

Getting started

This chapter gives some basic introduction on how to use the ConTroll system. More systematic information on the issues discussed here can be found in later chapters.

To **start** ConTroll using the Xtroll interface, go to the directory containing the grammar and type `xtroll &` at the shell prompt.[1] Two windows should open, the *Xtroll window* and the *Prolog window*. To **exit** ConTroll, press button `Quit` in Xtroll window.

A ConTroll grammar minimally consists of two files, which by default are called **signature** and **theory**.[2]

A simple example grammar A simple grammar consisting of the two files signature and theory can be found in the directory `ConTroll/demo/one`. The file `signature` looks as follows:

```
type_hierarchy
bot
  sign cat:cat agr:number phon:list
    phrase dtr1:sign dtr2:sign
    word
  cat
    np
    vp
    sv
    s
  number
    singular
    plural
  string
.
```

[1] How you can compile and load grammars in directories other than the one where you started ConTroll is described in chapter 8.

[2] Chapter 8 describes how the default filenames can be overridden.

A signature defines the ontology, i.e., the types, their appropriate attributes, and the hierarchical order of types. The type hierarchy is encoded as an indented tree. `bot` is the most general type and has direct subtypes `sign`, `cat`, `number` and `string`. The last type is a special type that allows you to use arbitrary strings in your grammar, even if they haven't been declared as types. Appropriate features are introduced by writing them after the type they should be appropriate for. In our example, `sign` has the features `cat`, `number` and `phon` with the appropriate values `cat`, `number` and `list`, respectively.

The file `theory` contains the following ConTroll code:

```
phrase ==> ( ( cat: s,
               dtr1: (np_sign(Num),
                      phon: Phon1),
               dtr2: (cat: vp,
                      agr: Num,
                      phon: Phon2))
           ; ( cat: vp,
               agr: Num,
               dtr1: (cat: sv,
                      agr: Num,
                      phon: Phon1),
               dtr2: (cat: s,
                      phon: Phon2))
           ),
           phon: Phon3, append(Phon1,Phon2,Phon3).

word ==> ( (np_sign(singular),
            phon: [(john ; mary)])
         ; (np_sign(plural),
            phon: [(cats ; dogs)])
         ; (cat:vp,
            phon: [(runs ; jumps)],
            agr: singular)
         ; (cat:vp,
            phon: [(run ; jump)],
            agr: plural)
         ; (cat: sv,
            phon: [(knows ; thinks)],
            agr: singular)).

append(list,list,list) **> bot.              %  type declaration

append([],L,L) := bot.
append([H|T],L0,[H|L]) := append(T,L0,L).
```

```
np_sign(number) **> sign.                    %  type declaration

np_sign(Num) := sign, cat:np, agr:Num.
```

It shows the two basic forms of constraints there are in ConTroll: implicational constraints and relational constraints. Constraints are expressed over feature terms, formulae reminiscent of HPSG's AVM notation. Feature terms are built out of four basic categories:

1. Variables – all Prolog variables are also ConTroll variables. As a rule of thumb, this is anything that starts with a capital letter. Variables play the role of HPSG tags.
2. Types – these are the things you have declared in your type hierarchy.
3. Attributes – also called features. Everything followed by a colon is a feature. All features must also be declared in the type hierarchy.
4. Relations – express relational dependencies between objects.

Feature terms can be conjoined with a comma and disjoined with a semi-colon. The operator precedence (from high to low) is colon before comma before semicolon before everything else. ConTroll also supports Prolog list notation. `[A|B]` is equivalent to `(ne_list, hd:A, tl:B)`. `list`, `ne_list` and `e_list` are built-in types that can not be redefined by the user. You can freely intermingle the two list notations.

A specification of a definite relation comes in two parts: a compulsory type declaration specifying the types of the arguments, and a set of clauses. As seen in the example of `append`, the clauses may be written in a Prolog-like notation with the notable difference of the type `bot` (the least specific type) replaces the Prolog atom `true`. One may view `bot` as the "output" of the relation, i.e., `bot` is output just in case the relation is satisfied. So if we assume that `Phon3` is the append of lists `Phon1` and `Phon2`, then the description `(phon: Phon3, append(Phon1,Phon2,Phon3))` is equivalent to `(phon: Phon3,bot)`, which in turn is equivalent to `(phon: Phon3)`.

This approach to relations is naturally extended to allow types or descriptions other than `bot` to be the output. This is seen, for example, in the the definition of `np_sign`, where the output is specified by the description `(sign, cat:np, agr:Num)`. This idea of specifying relations with outputs is due to (Dörre and Dorna 1993) and is implemented in the CUF system. The advantage of this approach is that it allows relations, which are also evaluated as relations, to be specified as if they were functions. So `append`, for example, could also be specified as:

```
append(list,list) **> list.
```

```
append([],List) := List.
append([H|T],L) := [H|append(T,L)].
```

and then the `phon` value of `phrase` could be specified more simply as `(phon: append(Phon1,Phon2))`. This functional style is most appropriate when a single argument of a relation is rather naturally viewed as an output value.

The implicational constraints have the intuitive meaning that everything the antecedent is true of, the consequent has to be true of as well. In our example, this means for instance that every phrase is of `cat:s` or `cat:vp`, and that the phonology of every phrase is the concatenation of the phonology value of the first daughter with the one of the second daughter. Notice that there is no explicit variable representing a phrase's phonology, due to our functional representations of relations.

Compiling a grammar Assuming that ConTroll was started by typing

```
$ xtroll &
```

to the prompt in the directory containing the demo grammar shown above, you can compile the grammar by typing

```
?- compile_user_grammar.
```

in the Prolog window of ConTroll.

Interaction with the signature ConTroll provides a number of predicates that can be used to show the result of compiling the signature. For example, if you type

```
?- show_approp(phrase).
```

in the Prolog window of ConTroll, the appropriate attributes and values for `phrase` are shown as a feature structure in a separate window:

$$\begin{bmatrix} \textit{phrase} & \\ \textsc{agr} & \textit{number} \\ \textsc{cat} & \textit{cat} \\ \textsc{dtr1} & \textit{sign} \\ \textsc{dtr2} & \textit{sign} \\ \textsc{phon} & \textit{list} \end{bmatrix}$$

Note that the appropriate features are not exactly as they were declared in the `signature` file. The feature `agr`, for example, was not declared to be appropriate for `phrase`, but since it was declared for the supertype `sign`, the ConTroll compiler computes the inheritance of this feature onto the subtype `phrase`.

To test the satisfiability of a description with respect to the signature, the following test is available:[3]

```
| ?-  check((word,dtr1:sign)).
% compiling file /home/server/gerdemann/xtroll_tmp_window_title.pl
% xtroll_tmp_window_title.pl compiled in module user, 0.000 sec 28 bytes
 ---> (locally) inconsistent
```

Here we can see that the type `word` does not allow a `dtr1` feature. On the other hand, the description `(word,cat:s)` is satisfiable as seen by:

```
| ?-  check((word,cat:s)).
% compiling file /home/server/gerdemann/xtroll_tmp_window_title.pl
% xtroll_tmp_window_title.pl compiled in module user, 0.030 sec 0 bytes
 ---> ok
```

This example shows that `check/1` only considers appropriateness conditions. According to the theory, there are no sentential words, but this is not ruled out by the signature.

Several other useful commands for displaying the results of signature compilation are discussed in Chap. 4.

Interaction with the grammar We call a description **satisfiable** iff it is consistent with the signature and the theory. To test some description for satisfiability, you can for example type the following in the Prolog window of XTroll

```
?- query(phon:[cats,run]).
```

If the query description is unsatisfiable, ConTroll will return with the answer: `no`. If the description is satisfiable, then ConTroll returns a feature structure which represents the application of all relevant constraints to the input query.[4] For this example, the result will be:

[3]Note the use of double parentheses in this case. The extra parentheses are necessary in order to bundle `word` and dtr1:sign into the single argument required by `check`.

[4]To be somewhat more precise, "applying" a constraint means further instantiating a description just enough so that it is consistent with the constraint. A constraint is "relevant" if it could, perhaps in combination with other constraints, lead to failure. A description consisting of just a type with no features is always assumed to be satisfiable. These issues are discussed in more detain in §6.2.3.

$$\begin{bmatrix} \textit{phrase} & \\ \textsc{cat} & s \\ \textsc{dtr1} & \begin{bmatrix} \textit{word} & \\ \textsc{agr} & \boxed{1}\textit{plural} \\ \textsc{cat} & np \\ \textsc{phon} & \left\langle \boxed{2}\textit{cats} \right\rangle \end{bmatrix} \\ \textsc{dtr2} & \begin{bmatrix} \textit{word} & \\ \textsc{agr} & \boxed{1} \\ \textsc{cat} & vp \\ \textsc{phon} & \boxed{3}\left\langle \textit{run} \right\rangle \end{bmatrix} \\ \textsc{phon} & \left\langle \boxed{2}, \boxed{3} \right\rangle \end{bmatrix}$$

Since we had the type `string` in our type hierarchy, we can input arbitrary Prolog atoms as types, but only where appropriate. Since the built-in type `ne_list` allows any type subsumed by `bot` as the value for `hd`, we can have for instance

```
?- query((List,append([uther,sleeps],[furiously],List))).
```

for which the result is pretty printed as:

```
< uther , sleeps, furiously >
```

Queries can be arbitrary feature terms, there are no restrictions.

```
?- query([mary,np_sign(X),(phon:[runs], agr:X)]).
```

$$\left\langle \textit{mary}\ ,\ \begin{bmatrix} \textit{word} & \\ \textsc{agr} & \boxed{1}\textit{singular} \\ \textsc{cat} & np \\ \textsc{phon} & \left\langle \left\{ \textit{john},\ \textit{mary} \right\} \right\rangle \end{bmatrix} ,\ \begin{bmatrix} \textit{word} & \\ \textsc{agr} & \boxed{1} \\ \textsc{cat} & vp \\ \textsc{phon} & \left\langle \textit{runs} \right\rangle \end{bmatrix} \right\rangle$$

This is very useful for grammar debugging, since you can check any kind of sub-expression for grammaticality.

Chapter 4

The signature: types and appropriateness

The two main parts of any ConTroll grammar (or program) are the *signature* and the *theory*. Of these two parts, the signature is the more fundamental. The signature lays out the basic ontology; it tells us what kind of objects the theory is about and it tells us what kinds of things even make sense to say about these objects. Or to put it another way, the signature just tells us what our field of inquiry is, whereas the theory tells us what has been discovered (or claimed to be true) about objects in this field of inquiry.

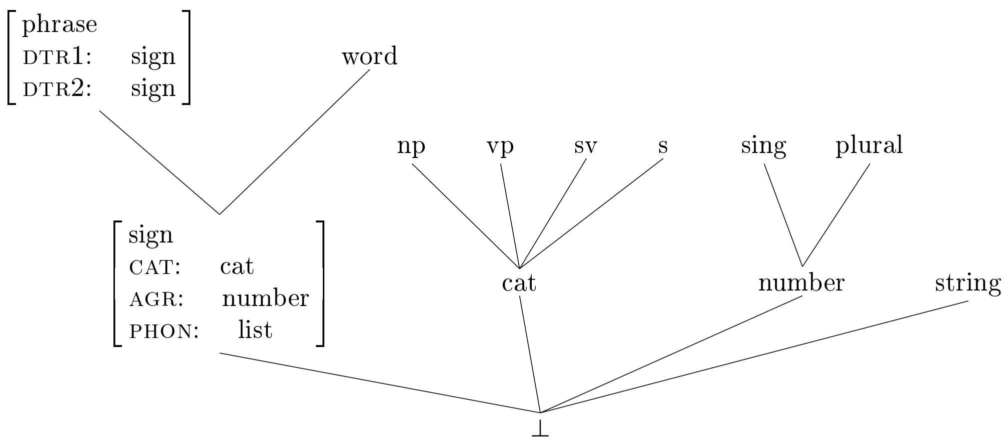
It typically follows from this basic division that the signature is much smaller than the theory. A signature may have, for example, a single type for `word` and then leave it up to the theory to specify a few thousand different instantiations of the feature-values on objects of this type. There is, however, some tradeoff here between what goes into the signature and what goes into the theory. Some grammar writers may prefer to have several subtypes of words, for which the theory would provide different kinds of constraints. And indeed some grammar writers may prefer to have a different type for each word in the lexicon. We will not pretend to decide here which approach is best for linguistic theory. But at least for ConTroll it is generally not a good idea to have a signature with many thousands of types.

Another typical consequence of the signature/theory division is that more grammar development time will be spent on the theory. Since any theory presupposes a given signature, a small change to the signature may require a major restructuring of the theory. Unsurprisingly, the same point applies to the ConTroll compiler. Any small change to the signature will require both the signature and the theory to be recompiled, whereas a change to the theory will require only the theory to be recompiled.

With this basic understanding of the division between the signature and the theory, we now turn our attention more directly to the signature.

4.1 Interpretation of the Signature

In Chap. 3, an example grammar was given with the following type hierarchy and appropriateness conditions:



In this chapter, we will consider in more detail what such a signature means.

A type hierarchy expresses the intuitive idea of subtyping. So, in this example, phrases and words are intended to be subtypes or kinds of signs (alternatively `sign` *subsumes* the types `phrase` and `word`). Moreover, phrases and words are the only kinds of signs. It is this last point that distinguishes the approach to typing in ConTroll from the approach taken by ALE and other related systems. In ALE, this example signature would indicate that phrases and words are kinds of signs, but it would not indicate that these are the only kinds of signs.

The maximal types, i.e., the leaves, are called *species*. The general principle is that types are partitioned by the species that they subsume. In other words, if an object is of a particular type then it must be of exactly one species subsumed by that type. This condition is referred to variously as *exhaustive typing* or the *partition condition*. Type hierarchies interpreted in this way can also be referred to as *closed world*, though this usage is rather more controversial.[1]

In any case, the difference between an exhaustive and non-exhaustive interpretation is quite profound. Suppose that we've loaded the theory on p. 9 and we try the following query:

```
?- query((sign,cat:np,dtr1:sign)).
```

[1]The problem is that the terms *closed/open world* have a different, though perhaps related, meaning within logic programming.

If you try this query, you will see that ConTroll returns with the answer 'no'. Although the description is certainly consistent with the appropriateness conditions, it still cannot describe any well-formed object. If it did describe an object it would, by exhaustive typing, have to be either a phrase or a word. But in this simple example grammar, there are no phrases of category `np`, so then it must be a word. But it also cannot be a word, since the feature `dtr1` is inappropriate for words. This example thus shows how the appropriateness conditions, constraints from the theory and the exhaustive typing condition can interact to rule out inconsistent descriptions.[2]

4.1.1 Some Consequences of Exhaustive Typing

To a large extent, type inference or inference about appropriateness in ConTroll works the same as it does in systems without exhaustive typing (such as ALE). Just as in ALE, for example, we can conclude that `cat` is an appropriate feature for `phrase` since it is declared as appropriate for the supertype `sign`. Also, just as in ALE, one can conclude that `color` is not an appropriate feature for `phrase` since it is not declared to be appropriate for `phrase` or any of its supertypes. Furthermore, just as in ALE, the appropriate value for a feature `f` on a type `t` must be subsumed by the appropriate values for `f` on all supertypes of `t`.

However, there are also some differences that exhaustive typing makes which affect how the signature and feature terms are compiled. Three important differences are the introduction of *downward inheritance*, the treatment of *unary branches* and the compilation of negative feature terms.

Downward Inheritance

One important consequence of exhaustive typing is that features may sometimes be inherited downward. Given the direction of our type hierarchies, this means that features may be inherited from subtype to supertype. Upward inheritance (from supertype to subtype) represents the normal case that applies even in systems without exhaustive typing. For example, `cat` is inherited upward onto `phrase`. Downward inheritance, for example inheriting `dtr1` onto `sign`, is generally not permitted. In this case, if `dtr1` were inherited downward onto `sign`, then by normal upward inheritance, we would infer that `dtr1` is appropriate for `word`, which is not intended.

In ConTroll, a feature `f` is inherited downward onto a type `t`, whenever `f` is appropriate (possibly by upward inheritance) for all species subsumed by `t`. For example, if we changed the example signature to make `f` appropriate for

[2] One could mimic exhaustive typing in a system without exhaustive typing: for each type t which subsumes species $s_1 \ldots s_n$, simply add the implication $t \Rightarrow s_1 \vee \ldots \vee s_n$. This could easily be done automatically. The difference is, however, that ConTroll has been optimized (and continues to be optimized) so that exhaustive typing is handled as efficiently as possible, i.e., by employing lazy evaluation and by adding as few explicit disjunctions as possible.

np, vp, sv and s, then since anything which is a cat is of one of these four species, we could conclude that f is also appropriate for cat. This downward inheritance is performed automatically by the ConTroll compiler.

When a feature f is inherited downward onto a type t, an appropriate value for this feature must also be inferred. Normally this value is the generalization of the appropriate values for f on all species subsumed by t. For example, if feature f takes the value phrase on np and vp and it takes the value word on sv and s, then since sign is the generalization (greatest lower bound) of phrase and word, feature f on the type cat will have the appropriate value sign.

In some cases, however, the use of generalization loses too much information. Suppose that f is a weird feature that takes the value sign on np and vp and number on sv and s. The generalization of sign and number is $\perp$, but this is clearly too general. In cases such as this, the ConTroll compiler will introduce a disjunction of values as appropriate for f.

Readers who are familiar with ALE will notice a correspondence between downward inheritance and the *feature introduction condition*. This condition, which is enforced in ALE but not in ConTroll, requires that for each feature there must be a most general type on which this feature is introduced. So, if f is declared to be appropriate on sign, np, vp and number on sv and s, then the grammar writer must "introduce" this type on the general type cat. In effect, the grammar writer performs downward inheritance of features. Beyond this superficial similarity, however, downward inheritance is rather different from the feature introduction condition: downward inheritance follows directly from the principle of exhaustive typing, whereas the feature introduction condition is a constraint which is added purely for the sake of efficiency.[3]

Unary Branching

A second consequence of exhaustive typing is that types are semantically equivalent to the disjunction of species that they subsume.[4]

[3] This efficiency gain comes since with the feature introduction condition, certain disjunctions may be expressed as underspecification. For example, if you have types zebra and prison_uniform which both have stripes as an appropriate feature, then a description such as stripes:vertical would have to be expanded into the disjunction: {zebra prison_uniform}. If on the other hand, the feature stripes is introduced on a supertype stripy_object, then such descriptions may be left non-disjunctive. Although ConTroll does not enforce the feature introduction condition, grammar writers may still find it useful to follow this condition purely for efficiency reasons.

[4] This view of typing is now apparently the accepted wisdom in HPSG. Bob Carpenter (in the HPSG mailing list) writes:

> If you look at the logic of 'underspecification' in HPSG, it's just disjunction in a very strong sense. Specifically, if we have a type A with subtypes B and C, then we'd always get exactly the same resulting signs by replacing any occurrence of A with (B or C).

Whether or not this is ultimately the right logic for linguistic theory is an open question. Carpenter continues (after the cited text) with arguments for a kind of logic as used in current versions of categorial grammar. (see: ftp://ling.ohio-

In fact types could be viewed as convenient and efficient abbreviations for disjunctions of species. Given this view of types, there is no need for unary branches in the type hierarchy. If `t` directly subsumes only `t1`, then these two types are abbreviations for exactly the same set of species.[5] Such unary branches can thus be eliminated by consistently using just one of these two types in both the grammar and in queries. In any case, the type `t` will always be replaced by `t1` in answers to queries.

Negation

Exhaustive typing is also of importance for the treatment of negation (and by extension also implication). A feature term of the form `~t` (where `t` is a type) can be compiled into a disjunction of all species that are not subsumed by `t`. By using DeMorgan's laws, negation can generally be pushed down onto types so it is critical that negation on types can be compiled out. In fact all negations in ConTroll are compiled into disjunctions with the one exception being negations of path equations.[6]

4.1.2 Multiple Inheritance

Multiple inheritance is fully supported in ConTroll. Just as in ALE the order restriction on type hierarchies is that they must be a *bounded complete partial order* or BCPO. This means that if types `t1` and `t2` have a common subtype then they must have a most general common subtype, which is called the unification of `t1` and `t2` (or $t1 \sqcup s2$). The reason for imposing this condition is primarily for efficiency. If `t1` and `t2` has a couple of incompatible common subtypes, then internally whenever `t1` and `t2` got unified, the result would introduce a disjunction.

The BCPO condition is very useful for imposing a certain degree of modularity into a system. There is a fundamental distinction between the simple constraint solving involved in type unifications and the more complex constraint solving involved in checking satisfiability of descriptions with respect to general implications in the theory. With the BCPO condition, the simple form of constraint solving can be performed very quickly by table lookup. Of course, grammar writers do not need to know the internal mechanism by which types get unified, but it may be useful for them to consider this basic issue of modularity: constraints expressed purely in terms of types and appropriateness conditions are fast, whereas constraints expressed in terms of general implications are slow.[7]

state.edu/pub/HPSG/Log/MailArchives/Archive4/0131.html)

[5]Naturally this also implies that the appropriateness conditions for these two types (after upward and downward inheritance) will also be identical. It is a fatal error if these appropriateness conditions are not identical.

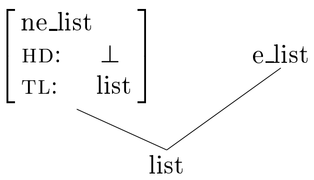
[6]The current implementation of ConTroll does not allow variables in the scope of negation, though there is no reason in principle why this could not be added in future releases.

[7]Actually it is still unclear to what extent grammar writers should be concerned with such

4.1.3 Additional Built-in Types

In addition to the types specified in the type hierarchy, a few built-in types are added during compilation. The most important of these are `list` and `string`. Additionally the type `constraint` is added, but this is a type which is used only for internal purposes, so its use will be hidden from the user in future releases. Also relation predicate names like `append` and `member` will be added to the hierarchy if these are defined. These relation types are shown by the user interface predicates in §4.4. For most purposes, however, predicate names do not function as types. The practical effect of adding them to the type hierarchy is to block the user from using the same name for both a type and a relation.

Since lists form an essential part of nearly all grammars, they are added as a built-in type. The advantage of having a built-in type is that it allows pretty printing routines to display lists in a more readable, Prolog-like, fashion. The following signature is used:



Note that ConTroll does not have parametric types such as `list(X)`, where X can be instantiated to any other type. One can, of course, define a variety of types like `list_sign`, `list_quant`, etc. But practically, if it is desired that all elements of a list are of the same type, it may be easier to define a relation such as:[8]

```
all_quant(list) **> bot.

all_quant([]) := bot.
all_quant([quant|R]) :=
   all_quant(R).

delay_deterministic(all_quant).
```

The type `string` is in a sense not built-in, but it has built-in properties. As opposed to `list`, `string` must be explicitly entered into the type hierarchy as an immediate subtype of `bot` and with no appropriate features.[9] Once `string` is added to the hierarchy, then implicitly any sequence of characters which is not

issues. The problem of automatically compiling slow implicational constraints into fast type constraints is an active area of research.

[8] For the use of `delay_deterministic` here see §7.2.1.

[9] No warning is given if `string` is specified as a non-immediate subtype of `bot` or is specified to have appropriate features. However, such unusual signatures will lead to unexpected results.

already used as a type or relation name, is interpreted as a subtype of string.[10] Strings are primarily useful for specifying values for features like `phon` in HPSG.

4.2 Input Format

The input to the type hierarchy compiler is one or more files containing information about a type hierarchy. Syntactically, a type hierarchy (or a fragment of it) is everything between the key word `type_hierarchy` and a period (`.`) each being the sole contents of an otherwise blank line. The first hierarchy, which may be the only one, must always begin with the type `bot` as the first word in the first informative line following `type_hierarchy`.[11] (Of course there may be empty lines or comment lines lying between them.) The following lines consist of regularly indented lines each containing one type and optionally its appropriateness conditions. Here is a small example:

```
type_hierarchy
bot
  s1 f:t1
    s4 f:t4
  s2 f:t2
    s4
  s3 f:t3
    s4

  t1
    t4
  t2
    t4
  t3
    t4
.
```

In general, each nonempty line consists of three parts:

Indentation Type [**Features**]

with:

Indentation consists of a number of spaces. We strongly suggest to use spaces, not tabs, since tabs are always considered to be equivalent to 8 spaces and this won't necessarily be what you see in your editor.[12] The subtypes

[10] If the string name begins with a capital letter, then it must be quoted when using the Prolog interface. Otherwise it will be interpreted as a variable.

[11] Some people prefer that the type hierarchy should begin with `top`. In a future release of ConTroll, we intend to provide this as an option.

[12] Also, be aware when mixing tabs and spaces: If you type in a space and then a tab, the space may become invisible on the screen, but still counts when calculating the indentation level!

to each type are listed below that type at a consistent level of increased indentation just as headings and subheadings are indented in an outline. So, if a type t occurs at indentation level I, then there must be a positive n such that all of the subtypes of t occur at indentation level $I + n$. For each type, a different indentation level n may be chosen for the subtypes, though in general, it will be preferable to chose an indentation level of 3 or 4 spaces and use that level consistently.[13]

Type is syntactically a term. It is the name of the type to be specified in the current line. Each informative line must contain at least a type name. A type may never occur twice as the subtype of one and the same supertype. It can, however, occur as a subtype of two different supertypes, i.e., for multiple inheritance. In this case the type should be proceeded with an ampersand (`&`).[14] A type name must always start with a lower case letter.

Features consists of a possibly empty number of terms of the form `feat:val` each two separated by white space. `feat` is the name of a feature and `val` is the name of the type appropriate for this feature. As with the types, a feature name must always start with a lower case letter. If a feature is specified for a type, all its subtypes inherit this feature without any need for the user to say so explicitly. But he still can, if he wants. There is also a case of downward inheritance. If all the subtypes of a type share a feature, this feature will be introduced on the common supertype. The type appropriate for this feature at the supertype must then be the disjunction of all the types appropriate for the feature on the subtypes. Often, though, such a disjunction of types may be expressed as a semantically equivalent single type.

If the type hierarchy is divided into several parts, the additional information is hooked into the first, root hierarchy in the following way. Each part consists of a type where to hook the information into and its subtypes specified in just the same manner as in the first hierarchy part. Therefore the format looks like

`type_hierarchy`
Type
Indentation Type [**Features**]
...
Indentation Type [**Features**]

[13]We feel that the advantage of this approach is that it allows the hierarchical nature of the type hierarchy to be seen without too much obscuring syntax. A purely graphical type hierarchy interface would be even better, but this is as close as we could come to a graphical presentation within the confines of ASCII. It must also be admitted that using level of indentation to indicate subtypes a certain level of complexity. It is no longer possible to give a BNF definition of the type hierarchy since the level-of-indentation must be passed around as a context sensitive feature.

[14]The ampersand is not necessary but it is recommended. Our experience has shown that unintended multiple inheritance, by accidently using the same type name for different purposes, is a common error. If the ampersand is omitted, the system will give a non-fatal error message.

.

The first line after `type_hierarchy` consists of a single type, the type to which the following additional hierarchy should be linked. Thus this type must have occurred before. Since the type is used as a hook, no feature specifications are allowed here. The following lines are type specification lines obeying the rules explained above. All types of the first indentation level are regarded as new subtypes of the type to be hooked up to. Re-specifying an identical subtype of one and the same type results in a syntax error even if the second entry specifies features non-existing on the first entry. To give an example, here is the same type hierarchy as above specified in three pieces:

```
type_hierarchy
bot
  s1 f:t1
  s2 f:t2
  s3 f:t3
    s4

  t1
    t4
  t2
    t4
  t3
    t4
.

... % some other stuff

type_hierarchy
s1
    s4 f:t4
.

... % some more stuff

type_hierarchy
s2
    s4
.
```

Finally since each type should be specified only once, cases of multiple inheritance need a special key word to signal to the compiler that the re-occurrence of this type is indeed intended. The key word is the & (ampersand) which may be placed in front of the type name when the type multiply inherits. It is left to the user where he wishes to place the &, that means, it is immaterial, whether the & is placed in front of all or some appearances of the type, or whether at the first appearance or at later ones. It is recommended that the & is used, but this is not mandatory. Indeed, the default for ConTroll is to make the use of it optional. See below under **Options** for more information.

4.3 Options

In the current implementation of ConTroll the only type hierarchy compilation option concerns the treatment of multiple inheritance.[15] A multiply inheriting type may be marked with a & in the type hierarchy. The option `must_mark_multi` determines whether this marking is optional or mandatory. Values for this option are

`yes` Every multiply inheriting type must be marked with a &. Failure to do so causes an error message and an abnormal program termination.

`no` The user may mark these types, but is not forced to do so. If a multiply inheriting type which is not syntactically marked is found, the user is warned, and the program execution continues regularly. This is the default option.

To set the option use either of the following:

```
?- set_option(must_mark_multi,yes).

?- set_option(must_mark_multi,no).
```

4.4 User Interface

A signature may be compiled in either of two ways:

command	**effect**
`compile_user_grammar`	compiles the signature along with the theory
`compile_user_sig`	compiles just the signature

Once a grammar has been compiled, the following predicates can be used to view the compiled signature:

command	**effect**
`show_approp(`*<Type>*`)`	displays the appropriateness conditions of type *<Type>*
`show_subtypes(`*<Type>*`)`	displays the direct subtypes of type *<Type>*
`show_all_subtypes(`*<Type>*`)`	displays all subtypes of type *<Type>*
`show_supertype(`*<Type>*`)`	displays the direct supertype of type *<Type>*
`show_all_subtypes(`*<Type>*`)`	displays all supertypes of type *<Type>*
`check(`*<Desc>*`)`	tests description *<Desc>* for consistency with the signature

[15] Readers who are familiar with previous versions may recall that there was also an option concerning the treatment of unary branches. The option allowed unary branches to be either ignored, removed or reported as errors. In the current implementation, unary branches are uniformly ignored. See §4.1.1 above.

4.5 Errors and Warnings

The type hierarchy compiler is able to detect quite a number of critical conditions in which a given input is obviously erroneous in some way. In the current version, the complier doesn't try to recover from user errors but rather issues an error message and aborts execution to force the user to correct his input. Warnings with continuation of the compilation are used only in relatively harmless cases.

4.5.1 Syntactic Errors

These errors are discovered while parsing. They arise as typos or faulty indentation.

"bot" expected The parser expected to find the string "bot" (without the quotes, of course). "bot" is the initial type which must start the root hierarchy specification, which must be the first hierarchy specification found.

'Type' expected The parser expected a type but found something else. This error may arise as a result of starting a type name with a capital letter.

'Feature:Value' expected The parser expected to find a feature-value specification. A feature can be any (non-capitalized) word and the value is an ordinary type. There must not be a space between the colon and either of feature or value.

Indentation error The indentation of the current line does not fit with any of the indentations on previous lines. In each new line, the indentation is either identical with some previous one or deeper than in the previous line.

Parsing error Unspecified parsing error in one of the hierarchy fragments. Unfortunately, no more specific information can be given. This error should not occur.

4.5.2 Semantic Errors

These errors occur when the type hierarchy is correctly parsed, but the specified type hierarchy is in some way malformed.

Unexpected multiple inheritance The option *must_mark_multiple_ inheritance* is set. A type is found which multiply inherits from two or more supertypes, although the type was not specified to be multiply inheriting. The type and its supertypes are printed out.

Types multiply defined One or more types are defined more than once. This can happen, if a type appears twice as a subtype of one and the same supertype.

Multiple appropriateness specification An appropriateness specification (feature declaration) for some type is given more than once.

No subsumption of values A feature occurs on a type and one subtype. But the value of the feature on the subtype is not subsumed by that of the feature on the supertype.

No complete partial order The type hierarchy is not a BCPO.

Incompatible values (multiple inheritance) A type inherits from several supertypes and there is a feature which occurs on more than one supertype. But the values for that feature on the supertypes are incompatible, that is, they don't unify.

Undefined value The value given for some feature is not a type. The value for any feature must be a type elsewhere defined.

No file The user specified no file where to find a type hierarchy.

Subtyping cycle Some type reappears as its own subtype. These subtyping cycles are not allowed.

4.5.3 Warnings

No type hierarchy found The parser could not find a type hierarchy in the input file(s). Consequently, nothing is done by the compiler.

Approp Specification changed If all subtypes of a type share a feature, this feature is inherited downward to that type. If there was already a specification for that feature on the type which is less specific then the disjunction of the values on the subtypes, then this disjunction is chosen as the new value. Type, feature, original value and new value are printed out to inform the user.

No multiple inheritance The user specified some types as multiply inheriting, i.e. he fronted them with an &, but the types actually don't inherit multiply. The types are treated as not being specified as multiply inheriting, i.e. the & is ignored.

Uninformative supertype This warning occurs if: Type t inherits from two (or more) supertypes s_1 and s_2 but s_1 subsumes s_2. In this case, s_1 is an uninformative supertype in the sense that whatever information may come from s_1 must also be present at s_2. A simple example is the following:

```
type_hierarchy
bot
  s1
    t
    s2
      &t
.
```

Chapter 5

Descriptions

Descriptions are fundamental throughout ConTroll. Both queries and answers to queries are descriptions. Implicational constraints in the theory file are descriptions that apply universally, i.e., to all objects. Also definite relations in the theory file are specified with descriptions as arguments. Since implications and definite relations are discussed in Chap. 6, this chapter will focus on descriptions as queries and answers to queries.

Let's consider what we mean when we say that a query is a description. Suppose that we are using ConTroll for parsing. In this case, we would not want the input to be some fully specified sign, with all features filled in and all types maximally specific.[1] Rather we would want the input to be just some partial specification of a sign; that is, a partial specification that describes just the phonology and perhaps some category information. Similarly, for generation the input query would describe just the logical form of a sign. In general, even in non-NLP domains, queries will consist of partial specifications, which (if consistent) will describe some kind of object.

It is perhaps a little less obvious that answers to queries should also be descriptions. After all, answers to queries are given as attribute value matrices (AVMs), which are syntactically different from the feature terms used in queries.[2] But this syntactic difference is minor: AVMs can be trivially translated into feature terms and consistent feature terms can also be translated into AVMs. Beyond the syntactic difference, however, there is also the issue of the differing roles that queries and answers play in a system like ConTroll. If a query is a description, one might have expected an answer to be an object that satisfies that query. Or to put it more concretely, one might have supposed that answers would be fully specified feature structures with all types being maximally specific and all

[1]See chapter 9 of Carpenter (Carpenter 1992) for more information on what it might mean for a feature structure to be maximally specific.

[2]AVMs are sometimes referred to in this manual as *feature structures* even though these are considered different entities in HPSG. Possibly this is an abuse of terminology, but it is also pretty standard usage within computational linguistics.

features filled in.[3] Returning such maximally specified feature structures turns out, however, to be impractical for reasons discussed in more detail in §6.2.3. The central problem is that the attempt to return maximally specific feature structures can result in many uninformative disjunctions or infinite structures. The lazy evaluation approach described in §6.2.3 yields underspecified structures which are generally smaller, easier to look at, and more insightful.

It is unlikely that users will have much trouble with the AVMs given as answers to queries. After all these are automatically generated. The bigger problem is to learn the syntax of the feature terms, which are used as descriptions in queries.

5.1 Feature Terms

The full syntax for feature terms is given in Appendix A. In this section, we will explain the syntax rather more informally. In order to provide specific examples, we will assume that the signature on page 9 has been loaded along with an empty theory.[4] Since descriptions are explained here with respect to an empty signature, the use of relational atoms in descriptions will be ignored until §6.2.

The simplest feature term is just a type name. If we use such a description in a query:

```
?- query(word).
```

then Prolog will return with the answer:

word

This may not seem like a very interesting answer, but then the question was not very interesting either. The question is simply: are there any words. ConTroll assumes that there are objects of every type, so no expansion or checking of constraints is necessary.

A second type of feature term consists of a feature value pair, where the value is another feature term. The pair is combined with a colon, as in:

```
?- query(agr:singular).
```

For this query, ConTroll returns with the answer:

[3] One may well have philosophical objections to considering objects to be such maximally specified feature structures rather than some kind of real-world linguistic objects. But if the objects of interest are non-graph theoretic, real-world objects, then it is clear that answers to queries could only be descriptions of such objects.

[4] If you use `compile_user_grammar`, ConTroll expects to find both a signature and a theory file. ConTroll does not complain, however, if the theory file is empty or if it contains only blank lines. Alternatively, one can use `compile_user_grammar`, which just compiles a signature.

$\begin{bmatrix} \textit{sign} & \\ \text{AGR} & \textit{singular} \end{bmatrix}$

In this case, ConTroll has figured out from the signature that only signs can have `agr` as an appropriate feature. Readers familiar with Carpenter (Carpenter 1992) will recognize this inference as an instance of TypDom, i.e., the type inferencing procedure that finds appropriate domain types for features. Such readers may also find the answer to the following similar query surprising:

```
?- query(agr:number).
```

For this query, ConTroll simply responds with the following *unfilled* AVM:

sign

The fact that the `agr` feature is present is enough to infer that this is a sign. But once we say that it is a sign, we don't need to additionally say that `agr` takes the value `number` since (in our sample type hierarchy) this is true of all signs. An alternative strategy (not employed by ConTroll) would be to return *totally well-typed* feature structures (Carpenter 1992). In this case, the answer would be:

$\begin{bmatrix} \textit{sign} & \\ \text{CAT} & \textit{cat} \\ \text{AGR} & \textit{number} \\ \text{PHON} & \textit{list} \end{bmatrix}$

But this is as if to say: Any sign is an answer to the query as long as `cat` has a value of type `cat`, `agr` has a value of type `number` and `phon` has a value of type `list`. This answer is extensionally equivalent to the answer given by ConTroll, but it is clearly not equivalent in terms of natural language usage: the totally well-typed answer falsely suggests that there might be signs not meeting all of the side conditions.

The additional advantage of the unfilled feature structures is that they are smaller. In complex grammars, it is not unusual for answers to queries to be AVMs with several hundred nodes, so it is important that this complexity be reduced as much as possible by not including any uninformative features. Unsurprisingly, reducing the size of feature structures is also useful for improving efficiency. Smaller feature structures can be processed more quickly, using less memory. Total well typing, on the other hand, in combination with ConTroll's type inferencing, easily leads to infinite feature structures, which could not be processed at all.

Another basic type of feature term is a variable. However variables make little sense by themselves, so we will first treat boolean combinations of feature terms and then return to variables.

Conjunctions are specified with a comma. A query such as:

```
?- query((sign,cat:np)).
```

describes something which is both a `sign` and has `np` as the value of the `cat` feature. Note that the colon has higher precedence than the comma, so this feature term is equivalent to: `(sign,(cat:np))`. One has to be careful with commas, however, since they have two other meanings within ConTroll.

1. List item separator example: [(cat:np),(cat:vp)|RestSigns]

2. Argument separator in relations example: append(A,B,C)

Additionally, commas are problematic since they are used as the argument separator in Prolog's relations. This is the reason why double parentheses are needed in the query given above. Without these double parentheses, Prolog will give an error message such as:

```
| ?-  query(sign,cat:np).
! Existence error in query/2
! procedure user:query/2 does not exist
! however, query/1 is defined
! goal:  query(sign,cat:np)
```

Despite the problems with using the comma for conjunction, there is also a great advantage; namely, with commas for conjunction and proper formatting, feature terms can be made to look very much like the easier-to-read AVM notation:[5]

```
phrase,
cat: s,
dtr1: ( word,
        agr: plural,
        cat: np),
dtr2: ( word,
        agr: plural,
        cat: vp)
```

A semi-colon is used for disjunction, just as in Prolog. The comma has higher precedence than the semi-colon, so `( t1 ; t2 , t3 )` is equivalent to `( t1 ; (t2 , t3) )`. Most disjunctions are compiled into disjunctive normal form, though this may well change in subsequent releases.

Negation is indicated with a tilde. As discussed in Chap. 6, negation is implicit in expressing implicational constraints in the theory. Negations are used less frequently in queries, though they may well arise in this context as well. A negation could, for example, be used to express: non-third person singular, as in the following type hierarchy and query:

[5] Complex queries are hard to format in the Prolog window. In such cases, place the query in a separate file in the form: `test :- query( <formatted_complex_feature_term> ).` Then load this into Prolog with `compile(<file_name>)` and call `test`.

```
type_hierarchy
bot
  sign agr:agr
  agr number:number person:person
  number
    singular
    plural
  person
    first
    second
    third
.
```

```
?- query(~((agr,person:third,number:singular))).
```
[6]

The answers to this query are perhaps more ambiguous than the user expected. The following three answers are given:[7]

$\left\{ \textit{constraint},\ \textit{e-list},\ \textit{first},\ \textit{ne-list},\ \textit{plural},\ \textit{second},\ \textit{sign},\ \textit{singular},\ \textit{third} \right\}$

$$\begin{bmatrix} \textit{agr} \\ \textsc{person}\ \left\{\textit{first},\ \textit{second}\right\} \end{bmatrix}$$

$$\begin{bmatrix} \textit{agr} \\ \textsc{number}\ \textit{plural} \end{bmatrix}$$

The problem is that something that is not an `agr` could possibly be of any other type in the type hierarchy. Normally, however, this is not a problem since a negation would usually be embedded in a specific context where appropriateness conditions would rule out most of the possibilities. For example, only two answers are given to the following query:

```
?- query((sign,agr:~((agr,person:third,number:singular)))).
```

$$\begin{bmatrix} \textit{sign} & \\ \textsc{agr} & \begin{bmatrix} \textit{agr} \\ \textsc{person}\ \left\{\textit{first},\ \textit{second}\right\} \end{bmatrix} \end{bmatrix}$$

$$\begin{bmatrix} \textit{sign} & \\ \textsc{agr} & \begin{bmatrix} \textit{agr} \\ \textsc{number}\ \textit{plural} \end{bmatrix} \end{bmatrix}$$

Negations are also subject to the restriction that no relations or variables may

[6]One needs to take care here that the comma is not interpreted as Prolog's argument separator. This is the reason for the double opening parens before `agr`

[7]The type `constraint` actually should not appear in the first answer. This is an internal type that should be hidden from the user (and it will be hidden in future releases). This is hardly a practical problem, however, since negations are virtually never used in such unconstrained contexts.

occur in their scope, though it is likely that variables will be allowed in the scope of negation in future releases. This brings us then to the question of variables.

Syntactically, ConTroll variables are just Prolog variables, i.e., a sequence of characters beginning with an uppercase letter. Variables are used to express equality. For example, to describe a phrase with agreement between the two daughters, one can use the following query:

```
?- query((phrase,dtr1:agr:X,dtr2:agr:X)).
```

Now that we've seen all the various types of descriptions, we will take a closer look at what ConTroll does when it sees these one of these descriptions.

5.2 Compilation of Descriptions

When ConTroll receives a query, the first thing it does is to check the query against the signature for satisfiability. Three possible results arise here:

1. The query is unsatisfiable and hence is rejected.
2. The query is satisfiable and is accepted.
3. The query is multiply satisfiable, and is split up into separate cases.

The first case arises in a query such as:

```
?- query((word,dtr1:np)).
```

Since words don't have daughters, this query is immediately rejected and ConTroll says "no". This means that no query was even sent to the ConTroll interpreter. Perhaps some user will be curious as to why a query such as this one fails and will try to invoke the debugger (see §7.3). This user will perhaps be baffled by the response:

```
ConTroll Debugger: No more solutions!
```

The debugger, of course, shows no solutions (or even attempts at solutions) since the query never made it to the interpreter.

The second case is perhaps the normal case. But even when the query is accepted, it may be altered by type inferencing in the process. So, if the query is:

```
?- query((sign,dtr1:np)).
```

type inferencing will change `sign` to `phrase` before submitting this to the interpreter. Similarly, with the following type hierarchy and query:

```
type_hierarchy
bot
  t f:bool g:bool
    t1 f:plus g:plus
    t2 f:plus g:minus
    t3 f:minus g:plus
  bool
    plus
    minus
.


?- query((t,f:(X,bool),g:X)).
```

ConTroll will determine that the reentrancy forces the type `t` to be a `t1` and will change the query accordingly before submitting it to the interpreter.[8]

The third case arises naturally when the query feature term contains a disjunction that is expanded into disjunctive normal form. What is perhaps more surprising is the fact that multiple cases can arise even when the input feature term does not contain disjunctions. Take, for example, the following type hierarchy and query:

```
type_hierarchy
bot
  t f:bool g:bool
    t1 f:plus g:plus
    t2 f:minus g:minus
    t3 f:minus g:plus
  bool
    plus
    minus
.


?- query((t,f:(X,bool),g:X)).
```

In this case the reentrancy forces `t` to be either `t1` or `t2`, but crucially not `t3`. Thus, this query is compiled into two cases, which are submitted sequentially to the interpreter.[9]

[8]This, by the way, shows a difference between type inferencing in ConTroll and type inferencing in ALE. In ALE the original query is considered to be *totally well typed* so no type inferencing would need to be done.

[9]In the older TROLL system, such cases were handled differently. It was noted that whenever a feature term is compiled into multiple cases, the underlying graph structure for all these

5.2.1 Compilation in the Theory

Constraints in the theory are dealt with in the next chapter. At this point, however, we may look ahead to see one way in which compilation of constraints is the same as the compilation of queries. It turns out that precisely the same 3 cases listed above apply when the theory is compiled. Consider, for example, the word principle from the grammar on p. 9:

```
word ==> ( (np_sign(singular),
            phon: [(john ; mary)])
         ; (np_sign(plural),
            phon: [(cats ; dogs)])
         ; (cat:vp,
            phon: [(runs ; jumps)],
            agr: singular)
         ; (cat:vp,
            phon: [(run ; jump)],
            agr: plural)
         ; (cat: sv,
            phon: [(knows ; thinks)],
            agr: singular)).
```

This implication has 5 disjuncts on its right hand side and indeed when we compile this, we receive the message:

```
   * word * 5 clauses compiled.
```

Now change this so that the last disjunct is of the form:

```
           (cat: sv,
            phon: [(knows ; thinks)],
            agr: np)     % an impossible value for agr
```

With this change, we now receive the message:

```
   * word * 4 clauses compiled.
```

No error messages are generated, but one of our lexical entries has vanished. Also similar to the example given above, it is possible for a description to compile into multiple cases. So, even when the word principle has 5 disjuncts on the right hand side, the user may find that 6 or more clauses were compiled.[10]

cases is identical. Thus a feature term could be compiled into a single feature structure, with distributed (named) disjunctions used to indicate the possible combinations of types on all the nodes. This kind of structure with multiple disjunctions on types has turned out to be more complicated to work with in a system like ConTroll that allows general implicational constraints. Thus, most disjunctions are currently treated as top-level disjunctions. The proper treatment of these disjunctions is still an active area of investigation.

[10] These discrepancies sound like something that could lead to terrible confusion, but in practice, no problems have been reported.

5.3 Extended Example

Given that queries (and descriptions in general) may be compiled into multiple cases, the reader may perhaps wonder: how many cases? It turns out that the number is an exponential function of the number of nodes in the compiled feature structure. This is the case since ConTroll's type inferencing must consider every possible extension of the types on all of the nodes, where "extension" means replacing a type by a species subsumed by that type. One can employ various strategies to try to be lazy in this process, but in the worst case, this is precisely what ConTroll must do.

As an example of this worst case, we consider in this section how the 2-Sat problem can be encoded with just a signature and no theory. Understanding this example is not crucial to being able to use the system. Those readers who are less formally inclined or who would like to get on to using ConTroll for linguistic purposes are advised to skip ahead to the next chapter.

The 2-Sat[11] problem is the following: test the satisfiability of a propositional logic formula given in conjunctive normal form, where each of the conjuncts is a binary disjunction. For example, an instance of the problem would be to test the satisfiability of:

$$(x_1 \vee \neg x_2) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_3 \vee x_1)$$

The following type hierarchy can be used for the entire class of 2-Sat problems.

```
type_hierarchy
bot
  true_disjunction
    td11
      td11_tt d1:t d2:t
      td11_tf d1:t d2:f
      td11_ft d1:f d2:t
    td10
      td10_tt d1:t d2:t
      td10_tf d1:t d2:f
      td10_ff d1:f d2:f
    td01
      td01_tt d1:t d2:t
      td01_ft d1:f d2:t
      td01_ff d1:f d2:f
    td00
      td00_tf d1:t d2:f
      td00_ft d1:f d2:t
      td00_ff d1:f d2:f
```

[11] The 3-Sat problem can also be encoded. It just takes a lot more types. A (rather different) encoding of the 3-Sat problem was first shown to us by Bob Carpenter (p.c.).

```
  bool
    t
    f
  cnf
    ne_cnf first:true_disjunction rest:cnf
    e_cnf
.
```

The idea here is that binary disjunctions have four forms:

td11	$(X \vee Y)$
td10	$(X \vee \neg Y)$
td01	$(\neg X \vee Y)$
td00	$(\neg X \vee \neg Y)$

Each of these disjunction types can true in 3 different ways; hence the 3 species for each. Given this type hierarchy, a query for:

$$(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$$

is formulated as:

```
query((
  cnf,
  first: ( td11,
           d1:X1,
           d2:X2 ),
  rest: ( cnf,
          first: ( td10,
                   d1:X1,
                   d2:X2 ),
          rest: ( cnf,
                  first: ( td01,
                           d1: X1,
                           d2: X2 ),
                  rest: ( cnf,
                          first: ( td00,
                                   d1:X1,
                                   d2:X2 ),
                          rest: e_cnf ))))).
```

This particular query happens to be unsatisfiable, but remove any one of the conjuncts and it becomes satisfiable.

Chapter 6

Expressing the grammatical constraints

ConTroll supports two basic kinds of grammar constraints: universal implicational constraints with complex antecedents, and definite relations. As an example of the first kind of constraint, consider the Head Feature Principle of HPSG (Pollard and Sag 1994). This universal constraint can be directly encoded in ConTroll as shown in figure 6.1.

```
phrase, dtrs:headed_struc  ==>  synsem:loc:cat:head: X,
                                head_dtr:synsem:loc:cat:head: X.
```

Figure 6.1: The HFP as example for a universal constraint

The ConTroll system allows a direct implementation of HPSG grammars without forcing the grammar writer to introduce a phrase structure backbone or recode the theories as logic programs. In addition, the availability of universal constraints in ConTroll also allows for a more modular encoding of traditional grammars using a relational backbone. This is because in a relational encoding all subcases need to be covered. The universal constraints with complex antecedents, on the other hand, generalize over all occurrences of some data structure and can attach the constraint to the relevant subset. Universal constraints are thus constraint-based in the intuitive sense: each structure which is not explicitly excluded is well-formed. Internally, the complex antecedents of such universal constraints and the occurrences of negation are eliminated, which is possible due to the exhaustive typing we assume. The resulting type constraints are then compiled into definite clauses using the method described in (Götz and Meurers 1995).

The second kind of grammar constraints are ordinary definite clauses with feature term arguments. The compiler detects places in which constrained types can occur and integrates the type constraints into the code by adding calls to the relational encoding of the universal constraints. As described in (Götz and Meurers 1996), the universal constraints are integrated in a lazy fashion, i.e. only in case the argument of a relation is specific enough to cause a conflict with a universal constraint does the compiler attach a call to the universal constraint. Such lazy interpretation has a significant efficiency payoff, especially for large grammars, since it results in preferred processing of those constraints in the large constraint pool which are specific enough to fail.

While in this section we deal with the specification of the two kinds of grammar constraints. In §7.2.1 we then discuss the procedural interpretation of the constraints and the control mechanisms of ConTroll which are included to allow the grammar writer to specify how the universal constraints and definite clauses are intended to interleave in processing.

6.1 Implicational constraints

General well-formedness conditions on objects are expressed via implicational constraints. These generally take the form

Term1 ==> Term2

where Term2 can be any term, and Term1 underlies the usual restrictions of terms in the scope of negation (no calls to relations and no variables). The meaning of such expressions is that *all* objects must satisfy them.

Implicational constraints have the following properties:

1. They're interpreted universally: every node in every feature structure constructed during computation is required to satisfy every implication in the grammar.

2. Satisfaction of implicational constraints is lazy: terminal nodes in feature structures are not checked for consistency with implicational constraints, since no features are defined for them and thus no contradiction *should* arise. This presupposes that the system of implications is consistent in the sense that there is a well-formed structure for each type. Since this property is in general undecidable, it is up to the grammar writer to ensure it and thus correctness of computation.

3. As in type inference, for implication satisfaction we also assume a closed world interpretation.

4. The first step in the compilation of implicational constraints is an equivalence transformation that generates implications with types as antecedents. It proceeds in two steps. First, the most specific type `t` for the antecedent A is inferred: A ==> B $\leadsto$ `(t,A)` ==> B. In a second step, A is moved to

the righthand side with `(t,A) ==>` B $\rightsquigarrow$ `t ==> ~A;(A,B)`. To see what happened during the transformation from implicational constraints with complex antecedents to type constraints, see the file:

```
compiled(complex_antecedent_elimination)
```

The negated antecedent on the right-hand side (~A) may, depending on the structure of A, be a more or less big disjunction. Suppose ~A is transformed to B ; C. Now suppose you apply the resulting constraint to a structure that contains the information from both B and C. Then you will get two solutions that are exactly identical, since both disjuncts can apply. To avoid this case (there are other, more subtle ones), we offer an option that transforms ~A not into B ; C, but the logically equivalent (B,~C) ; (C,~B) ; (B,C). Surprisingly enough, this turns out not to explode the number of disjuncts for real grammars, since many of them turn out to be inconsistent. See App. B for how to turn this option off.

5. Two properties follow directly from universal interpretation as described above:

   (a) Implications are inherited. If there are implications `a ==> T1` and `b ==> T2`, and `b` is a subtype of `a`, then this interpreted to mean `b ==> T1, T2` (assume that `T1` and `T2` are standardized apart).

   (b) Implications are interpreted conjunctively: if there are implications `a ==> T1` and `a ==> T2`, then these are interpreted as if you'd written `a ==> T1, T2`.

6.1.1 Naming implications

For purposes of fine-tuning control, and for debugging, it is often convenient to be able to refer to implications individually. As implications are first transformed to type constraints, and then conjoined and inherited, it is sometimes difficult to find out how certain global effects came about. To this end, ConTroll provides the === construct:

```
Name === A ==> B.
```

This construct instructs the compiler to generate an auxiliary relation that will be inserted into the grammar as a place holder for the implication. It generates the following code (where `t` is the most specific type deducible for A, as described above):

```
Name **> t.
Name := ~A.
Name := A,B.
t ==> Name.
```

6.1.2 A special notation for the lexicon principle

In principle, the lexicon in ConTroll has no special status. A typical HPSG lexicon is an implicative constraint on objects of type *word*. The consequent of this constraint is a disjunction of descriptions, where each description is a *lexical entry*. Such an HPSG lexicon can be directly expressed in a ConTroll grammar in the following way, where the *<LE>* represent the different lexical entries:

```
word ==> <LE1> ; <LE2> ; ...  ; <LEn> .
```

Since encoding the lexicon in such a disjunction is notationally impractical (and most Prologs only allow terms of limited length), ConTroll offers a special notation to specify lexical entries. This notation consist of three specifications:

- Each lexical entry is specified as a statement of the form:

  <Desc1> ---> *<Desc2>*.

- How the two descriptions to the left and right of the ---> operator are mapped into an ordinary description, i.e., a description like the *<LE>* in the traditional HPSG lexicon we saw above, is determined by including one statement of the following form in a theory file:

  lexicon_style(*<Var1>*,*<Var2>*,*<Desc>*).
  <Var1> is a variable corresponding to the left-hand side of each ---> statement, and *<Var2>* to the right-hand one. Using these variables, the description *<Desc>* specifies how the left-hand side and the right-hand side are supposed to be combined to form an ordinary lexical entry.

- A statement to declare for which type of object the new notation is intended to introduce the implicational constraint encoding the lexicon. For example, in a traditional HPSG lexicon, objects of type *word*. One such declaration of the lexicon type obeying the following syntax should be included in a theory file:

  lexicon_type(*<Type>*).

This special notation is compiled exactly like an ordinary implicative statement on the type defined to be the lexicon type. Note that this means that additional implicative constraints on the lexicon type are interpreted conjunctively in the usual way!

Consider as an example the lexicon from the grammar on p. 3:

```
word ==> ( (np_sign(singular),
            phon: [(john ; mary)])
         ; (np_sign(plural),
            phon: [(cats ; dogs)])
         ; (cat:vp,
```

```
            phon: [(runs ; jumps)],
            agr: singular)
         ; (cat:vp,
            phon: [(run ; jump)],
            agr: plural)
         ; (cat: sv,
            phon: [(knows ; thinks)],
            agr: singular)).
```

This can be equivalently expressed as

```
lexicon_type(word).
lexicon_style(Phon,Term,(Term,phon:[Phon])).

(john ; mary) ---> np_sign(singular).
(cats ; dogs) ---> np_sign(plural).
(runs ; jumps) ---> cat:vp, agr: singular.
(run ; jump) ---> cat:vp, agr: plural.
(knows ; thinks) ---> cat: sv, agr: singular.
```

This is both more compact and easier to read. A slightly more complex example might use the following lexicon style to encode the string as a difference list under the features `phon1` and `phon2`:

```
lexicon_style(Phon,Term,(Term,phon1:[Phon|Rest],phon2:Rest)).
```

6.2 Relations

The definite clauses of ConTroll are modeled on those of CUF – both syntax and semantics are very similar in the two systems. Relations are written in functional notation, i.e., one argument is always the distinguished result argument.

Unlike CUF, type declarations for definite relations are not optional in ConTroll. This enforces a strict typing discipline on the side of the user and allows compile time type checking of relation arguments. A clause typing statement consists of a relation term with types as arguments, followed by the typing arrow (`**>`) followed by a type.

```
append(list,list) **> list.
```

Notice that ConTroll treats relation names like types. This means that the set of relation names and the set of types must be disjoint: you can't have a type `phrase` and a relation of the same name, for example. Also, you're not allowed to use the same relation name with different arities.

A definite clause consists of a relation term, followed by `:=`, followed by a feature term.

```
append([],L) := L.
```

Of course, definite clauses can also be recursive. The feature term on the right side of the `:=` can contain any number of definite relation calls.

```
append([H|T],L) := [H|append(T,L)].
```

If more than one relation call appears in a definition, the order of their execution (without delays) is approximately left-to-right, innermost-first. If there's more than one clause to a definition, the interpreter will try them in the order they appear in in the program.

```
naive_reverse(list) **> list.

naive_reverse([]) := [].
naive_reverse([H|T]) := append(naive_reverse(T),[H]).
```

In this example, the base case of `naive_reverse` is always tried first. In the recursive case, the call to `naive_reverse` is executed before the call to `append`, since `naive_reverse` appears inside of `append` (the innermost-first principle). In general, it is hard for the user to predict in which order goals will be executed by the system. To (partially) determine the order, use delay statements.

Notice that if you don't like the functional style, or if it is simply not appropriate for some relation (e.g., if there is no result argument in an intuitive sense), there is a simple way of using standard relational style instead. As an example, consider the definitions of `append` and `naive_reverse` in relational style.

```
append([],L,L) := bot.
append([H|T1],L,[H|T2]) := append(T1,L,T2).

naive_reverse([],[]) := bot.
naive_reverse([H|T],R) := naive_reverse(T,R1), append(R1,[H],R).
```

This is still a functional definition, but the result argument is simply left unspecified. As no result is specified, the atoms in the bodies of the clauses can in effect occur anywhere.

6.2.1 Abbreviating descriptions (Macros)

In an HPSG theory, usually one does not only want to describe attributes which are appropriate for a certain object, but also attributes, which are defined on an object that can be accessed by following a whole path of attributes. For example, in HPSG II the VFORM attribute of a *sign* needs to be accessed by following the path SYNSEM|LOC|CAT|HEAD.

When writing a grammar one wants to abbreviate often used access paths. In ConTroll this can be done using the definite relation mechanism already supplied for formulating theories.[1]

For example, after writing the definite relation definition below, one can describe the value of the VFORM attribute of a *sign* as being *fin* by writing the relation call `vform_s(fin)` instead of the longer `synsem:loc:cat:head:vform:fin`.

```
vform_s(vform) **> sign.

vform_s(X) := synsem:loc:cat:head:vform(X).
```

Efficiency considerations However, from a processing point of view, it is inefficient to treat macros in the same way as ordinary relations. We thus implement a fast, purely syntactic preprocessing step that finds the relations that can be treated as macros, i.e., unfolded at compile time. These macro relations are then compiled into an internal representation during a second preprocessing step. When the rest of the grammar is parsed, any macro goal will simply be replaced by its internal representation.

After the actual compilation, ConTroll closes the grammar under deterministic computation. This step must be carefully distinguished from the macro detection described above. A *goal* is deterministic in case it matches at most one defining clause, but a *relation* is a macro by virtue of its definition, irrespective of the instantiation of the actual calling goals. Of course, the macro detection step can be eliminated, since the deterministic closure will also unfold all macros. However, for our reference grammar, adding the macro detection step reduced compile times by a factor of 20. Thus, for large grammars, compilation without macros is simply not practical.

The criteria for a relation to be treated as a macro are the following.

- The definition must consist of exactly one defining clause.
- The clause body must be purely conjunctive, with the exception of type disjunctions.
- All arguments of the clause head must be variables.
- All arguments of the clause head must also occur in the body of the clause.

The first two conditions ensure determinacy of the relation. It would in principle be possible to also treat non-deterministic relations as macros, but these would then have to be marked by the user. The other two conditions ensure correctness of macro unfolding. No checks are performed on the arguments of macros,

[1] Since deterministic definite relations are expanded at compile time, using definite relations as abbreviations has no negative effect on efficiency. But note that this also means that the abbreviations are usually not visible when tracing through the grammar.

so it must be ensured that whatever the arguments are, they will be checked eventually. This is done by ensuring that no argument is “lost” at compile time, as they also occur in the body.

Obviously, making automatic macro detection a property of the compiler relieves the grammar developer from the burden of distinguishing between macros and relations, thereby eliminating a potential source of errors.

6.2.2 Automatic generation of relations on different types

Since HPSG theories usually formulate constraints about different kind of objects, the grammar writer usually has to write a large number of relations to access the same attribute, or to make the same specification, namely one for each type of object which this macro is to apply to. For example, when formulating immediate dominance schemata, one wants to access the VFORM specification of a *sign*. When specifying the valence information one wants to access the VFORM specification of a *synsem* object. And when specifying something about non-local dependencies, one may want to refer to VFORM specifications of *local* objects.

ConTroll therefore provides a mechanism which derives definitions of relations describing one type of object on the basis of relations describing another type of object – as long as the linguist tells the system which path of attributes leads from the first type of object to the second.

The path from one type of object to another is specified by including a declarations of the following form in the grammar:

```
access_rule(type1,path,type2).
```

Such a statement is interpreted as: From `type1` objects you get to `type2` objects by following path `path`.

Since only certain relations are supposed to undergo this extension, they are specified using slightly different operators than standard ConTroll relations: access relations are specified using the ':==' instead of the ':=' operator of ordinary relations. Furthermore, access relations do not require explicit '**¿' typing statements. The type of the relation is determined on the basis of the access suffix. The typing of each of the arguments (if any) is added using the `-` operator after each argument.

To distinguish the relation names for different type of objects, a naming convention for relations is needed. All relations on objects of a certain type therefore have the same suffix, e.g., "_s" for all relations describing *signs*. The type-suffix pairing chosen by the linguist is specified by including declarations of the following form in the grammar:

```
access_suffix(type,suffix).
```

Such a statement declares that the names of relations describing objects of type `type` end in `suffix`.

So the extend access mechanism reduces the work of the linguist to providing

- relations describing the 'most basic' type of object, and
- `access_suffix` and `access_rule` declarations.

Access relations are not compiled directly. Instead extension of the access relations is done as a pre-processing step at the very beginning of grammar compilation. It results in definitions of ordinary ConTroll relations which are written to the file `access_relations` in the compiled directory. Note that this file is automatically included in the grammar, i.e., it does not need to be explicitly specified as part of the grammar.

A small example As an example, say we want to have abbreviations to access the VFORM of a *sign*, a *synsem*, *local*, *cat*, and a *head* object. Then we need to define a relation accessing the most basic object having a VFORM, namely *head*:

```
vform_h(X-vform) :== vform:X.
```

Second, (once per grammar) `access_suffix` and `access_rule` declarations for the grammar need to be provided. The former define a naming convention for the generated relations by pairing types with relation name suffixes. The latter define the rules to be used by the mechanism by specifying the relevant paths from one type of object to another.

```
access_suffix(head,"_h").
access_suffix(cat,"_c").
access_suffix(loc,"_l").
access_suffix(synsem,"_s").
access_suffix(sign,"what_a_great_suffix").

access_rule(cat,head,head).
access_rule(loc,cat,cat).
access_rule(synsem,loc,loc).
access_rule(sign,synsem,synsem).
```

This results in the following macros to be generated:

```
vform_h(X)                   :=  vform:X.
vform_c(X)                   :=  head:vform_h(X).
vform_l(X)                   :=  cat:vform_c(X).
vform_s(X)                   :=  loc:vform_l(X).
vformwhat_a_great_suffix(X)  :=  synsem:vform_y(X).
```

If we were only interested in a vform relation for certain objects, say those of type *sign* and *synsem*, it would suffice to specify access rules for those types instead of the access rules specified above. The following specifications would do the job:

```
access_suffix(head,"_h").
access_suffix(synsem,"_s").
access_suffix(sign,"what_a_great_suffix").

access_rule(synsem,loc:cat:head,head).
access_rule(sign,synsem:loc:cat:head,head).
```

The result would then be:

```
vform_h(X)                     :=  vform:X.
vform_s(X)                     :=  loc:cat:head:vform_h(X).
vformwhat_a_great_suffix(X)    :=  synsem:loc:cat:head:vform_h(X).
```

Warnings

Several kinds of warnings can appear on user output. Access expansion continues.

1. A ConTroll relation already defined always has priority over a derived relation, regardless of whether
   (a) the derived relation is the direct translation of an access relation defined by the user or
   (b) the derived relation is the result of access_rule applications.
2. If a ConTroll relation has already been derived by translation of an access relation with or without access_rule application, an access relation occurring later in the grammar which would derive the same relation is not translated an no further rules are applied to the later access relation. Currently this even is the case if the two predicates differ in arity, since in ConTroll a relation name has to be unique.

Errors

Several types of errors can be detected and a message is printed on user output. Access expansion then aborts.

1. A type occurring in an access_rule is not allowed to have
   (a) multiple suffixes defined for it
   (b) no suffixes defined for it

2. Two suffixes defined must not
    (a) be identical
    (b) have a common ending

6.2.3 Compiling Constraints and Relations

While the uniform representation of all information in one kind of datastructure and the formulation of a grammar as one big collection of constraints can be seen as a clean theoretical basis for HPSG, from an implementational point of view, the large size of the single data structure and the problem that the formulation of the grammar does not distinguish between constraints used to license the recursive constituent structure and other 'less important' constraints is a problem. The large data structures specified can make it hard to see the relevant aspects of the reply given by the system. Furthermore, the system spends much time applying constraints which can never fail just to be able to enumerate specific answers. One way to decide which constraints from the large pool should be applied when, is to only apply those constraints which can lead to a conflict given the information provided in the query. The basic idea of lazy evaluation is that nodes with *more information content* should be preferred in evaluation over nodes with less information content. This suggests an on-line strategy for goal selection based on the idea of laziness. However, we would like to take a compilation approach to laziness. Instead of reordering goals on-line, we compute off-line which nodes need to checked *at all* to guarantee there is a solution. This means that our on-line proof strategy is exactly identical to the non-lazy case, but it needs to do less work. In (Götz and Meurers 1996) we discuss, how the ConTroll compiler takes a grammar and compiles it such that standard evaluation yields a lazy strategy. Lazy evaluation in our approach therefore is not an on-line goal reordering strategy, but the result of an optimizing compiler. This removes the overhead of on-line goal reordering from processing.

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 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 are 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.

Chapter 7

Running and debugging ConTroll grammars

7.1 The compiler

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

The compiler prints some information when compiling constraints. Below is part of the compiler log for the grammar in `.../demo/one`.

```
 *** Compiling constraints on phrase
  ** Computing inheritance on type sign
+

  ** Computing inheritance on type phrase
~~~
+++

   * phrase * 3 clauses compiled.
```

The first line indicates that the compiler will now compile the constraints on type phrase. Since phrase inherits from sign, the constraints on sign have to be computed first. There are no constraints on sign, so the trivial constraint (that does nothing) is recorded. This is expressed by the single `+`. Now the compiler turns back to phrase, as it has finished with sign. Each of the `~`s indicates one disjunct of the constraint on phrase read in by the ConTroll parser. Each of the `+`s indicates on clause actually generated. The number of `~` and `+`s does not have to be identical, since some of the disjuncts may be inconsistent, or some of them may be further split up by type inference. Finally, the compiler gives a message saying how many clauses it created. It is useful to check this number against your expectations.

The internal representation of the constraints may be viewed with the command `show_constraint/1`. It takes a type or a relation name as argument and pops up an XTroll window for each clause.

7.2 The interpreter

As a guiding principle, the interpreter follows the ideas of the Andorra Model[1] 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.

7.2.1 Delay statements

There are several different delay statements. The basic one is `delay/2`. It takes a relation name as first argument, and a delay pattern as second. Delay patterns are simple terms with no negation, relation calls and list syntax. The arguments of relations may be referred to as `arg1` for the first argument, `arg2` for the second, and so on. The result argument is always called `arg0`. Here's an example for the append relation.

```
delay(append,arg1:(e_list ; ne_list)).
```

Notice that what the statement above says is that the first argument must be more specific than the plain `list` type, i.e., an instantiated list. Since this is a common occurrence, there is a special syntax to express the requirement that the type on some node must be properly more specific than some given type `t: @ t`. Thus, the delay statement above could be equivalently expressed as follows.

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

```
delay(append,arg1: @ list).
```

It is generally more efficient to use the `@` operator than a disjunction of types. This is because disjuntive delay statements are compiled to disjunctive normal form.

Finally, it is sometimes desirable to require that some argument be instantiated to arbitrary depth. Since this cannot be stated in terms of ordinary delays, ConTroll allows a very restricted form of regular expressions: `(f1:f2:...fn * Term)`. For example, one might require that the length of the input argument of `naive_reverse` be known.

```
delay(naive_reverse,arg1:(tl* e_list)).
```

If after following the tail path a finite number of steps, one encounters the `e_list`, then the length of the list is known. These Kleene star expressions come with the restriction that they may not be nested. This is to ensure that they can be deterministically checked at run-time.

In general, there can be more than one delay statment for any given relation. Multiple delays are interpreted *conjunctively*. It is thus very rarely useful to distribute delaying information for a relation over several delay statements.

Another form of delay statement is `delay_deterministic/1`. It requires that a goal of a given relation only be executed if at most one clause matches it. Example:

```
delay_deterministic(naive_reverse).
```

This is even stronger than the delay for `naive_reverse` shown above, since it not only talks about the first argument, but the result argument as well. Computationally, `delay_deterministic/1` is the cheapest delay statement, and should thus be preferred.

Implicational constraints In principle, the same delaying mechanisms apply to relations and constraints. Suppose you have a type `phrase` with a feature `phon` that takes a list of strings as value. You can delay applying constraints on nodes of type phrase with, e.g.:

```
delay(phrase,phon:(tl* e_list)).
```

Notice, however, that this delays applying *any* constraint on objects of type phrase, not just those that are actually affected by the instantiation of the `phon` feature. As mentioned above, this may not always be appropriate. Some constraints might fruitfully apply and provide further instantiation for other constraints to fire. As an example, suppose you have a constraint about the distribution of the `phon` value of a phrase among its various daughters.

```
phrase ==> phon: append(P1,P2), hd_dtr:phon:P1, comp_dtr:phon:P2.
```

For parsing, you may want to delay the application of this particular constraint as described above, but to be able to do so, you need to be able to refer to this constraint individually (assuming that this is not the only constraint on `phrase`). You need to give the constraint a name and then put a delay on that name.

```
id_principle ===
phrase ==> phon: append(P1,P2), hd_dtr:phon:P1, comp_dtr:phon:P2.

delay(id_principle,arg0:phon:(tl* e_list)).
```

This way, other constraints that are independent of the `phon` features value can fire.

The `*` has the highest binding precedence of all operators. Thus, if you want to star a path you will have to put parens around it: `f:g:h*t` is `f:g:(h*t)`, not `(f:g:h)*t`.

7.2.2 Indexing of grammar constraints

ConTroll supports arbitrary multiple indexing of grammar constraints. The system allows the user to specify indexing information for predicates individually. This is comparable to the indexing of terms in relational databases, e.g., the SICStus Prolog external database (Nilsson 1995). Figure 7.1 shows the definition of a two-place relation `r` including a typing declaration and two indexing instructions. Given a fully instantiated goal for the relation `r`, the run-time environment of ConTroll can deterministically pick the right clause without leaving behind a choice-point.

```
r(t)   **>   t.
r(a)   :=    b.    index(r,arg0:t).
r(a)   :=    c.    index(r,arg1:t).
r(b)   :=    c.
```

Figure 7.1: Indexing specification for `r`

The indexing mechanism not only works for relations, but also implicational constraints. Figure 7.2 shows possible indexing instructions for the lexical type `word`, namely for the phonological form, and the syntactic category.

```
index(word,phon:hd:string).
index(word,synsem:loc:cat:head:head).
```

Figure 7.2: Indexing for the type `word`

7.3 The debugger

The ConTroll debugger can be invoked by the user with the command `inspect/1` that takes a query as argument. The debugger may also be called at other places, for example when no more undelayed goals exist in a computation, or when one tries to find out why a certain clause fails deterministically at compile time.

In XTroll, the debugger controls its own window. Figure 7.3 shows an example of the debugger window after the command `inspect(append([a,b],[c]))`. We

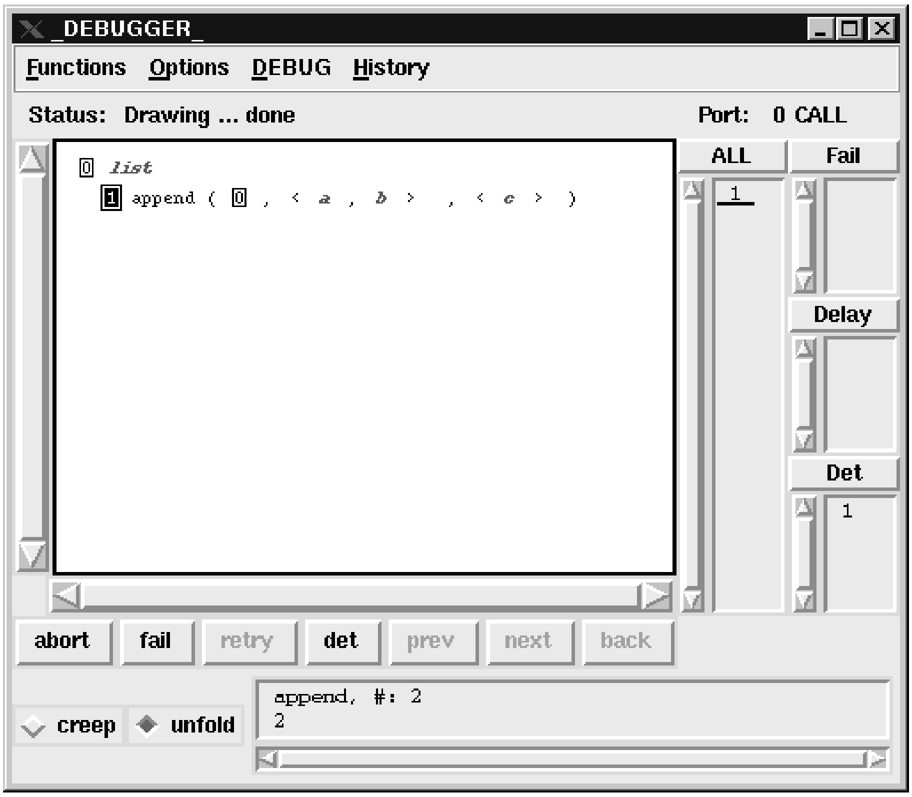


Figure 7.3: Debugger window for `inspect(append([a,b],[c]))`

will describe this example in some detail, and then give a full description of the debugger.

The debugger window consists of a central display area showing the state of the computation, plus some additional display areas and buttons. The central display area shows an AVM that represents the answer that is being constructed, with relational goals displayed underneath. In our example, the answer constructed so far is just a `list` with tag 0. We still need to solve our original query. The tag in front of relational goals identifies them for the debugger. All goals are listed in the lists to the right of the display area. There are three lists: the list ALL contains all goals, in the order in which the interpreter would select them. That is, the topmost goal is always the selected goal. A different goal may be selected by the user by clicking on it in the ALL list with the left mouse button. The selected goal is underlined. Clicking with the right mouse button on a goal in a list will display the matching clauses for the goal in question. By matching clauses we mean those clauses whose heads unify with the goal.

The other list are Fail, Delay and Det, which show the failed, delayed and deterministic goals, respectively. A goal is failed, if no clause head unifies with it, and deterministic, if exactly one does. The list of delayed goals may intersect with both deterministic and failed goals. The selected goal, i.e., the goal that the next resolution step will act upon, is underlined in the lists. As mentioned above, the user may select a different goal by clicking on it with the left mouse button.

Underneath the usual menu bar, the debugger displays some status information. To the left, XTroll says what it's doing at the moment. At the right end of the status line, the debugger displays information about the port and depth of computation. There are only three ports in our debugger model: *call*, *redo* and *frame*. Since ConTroll doesn't follow Prolog's strict left-to-right, depth first proof strategy, *exit* and *fail* are not appropriate for our debugger.

The window in the lower right-hand corner, finally, provides information about the selected goal. It first gives the type of the goal (or the relation name, as in the example), and the number of clauses defined for that type/relation. In the second line, it shows a list of numbers describing the matching clauses for the goal. In case of a redo, only those matching clauses are shown that haven't been tried before.

There are two kinds of buttons: seven standard buttons and two radio buttons, exactly one of which is pressed at all times. Not all standard buttons can be pressed at all times. Here's a description of their functionality:

abort kills the current debugger process and closes the debugger window. This is the correct way of terminating a debugging session.

fail tells the debugger to consider the current state of computation as failed and to induce backtracking.

retry is only possible in a redo or frame port. In a redo port, all clauses of the selected goal are considered untried and the debugger enters call mode. In a frame port, the debugger makes the frame state the current state and discards all work done following that state.

det induces the debugger to take the deterministic closure over the current state. This is only possible in call mode.

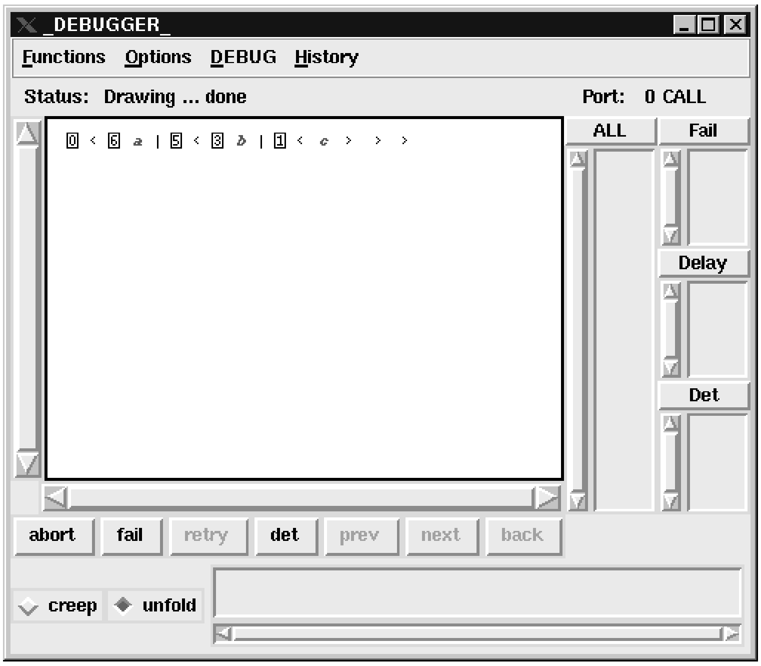


Figure 7.4: Debugger window after pressing det button

prev takes the computation back a frame for the user to view a previous state of the computation. The port changes to frame. None of the usual actions are possible when in frame mode. Press retry to resume tracing in a frame state.

next is the partner command to prev. If in a frame state, it moves the debugger one state further. Next is only applicable in frame mode.

back moves the debugger back to where framing was started. Obviously, back can only be selected in frame mode.

The radio buttons control the behaviour of the resolution steps. When in creep mode, one step consists of using the first matching clause for the selected goal for resolution, and then closing the resulting state under deterministic computation. Unfold does the same thing, except for the deterministic closure. Doing one creep step is thus equivalent to doing one unfold step, and then press the det button for deterministic closure. When the debugger comes up, the unfold button is pressed by default.

Let us now return to our example. The state in Fig. 7.3 is not very interesting. There is only one goal, and it is deterministic. This can be seen by the fact that it is listed in the Det list, as well as that there is only one matching clause

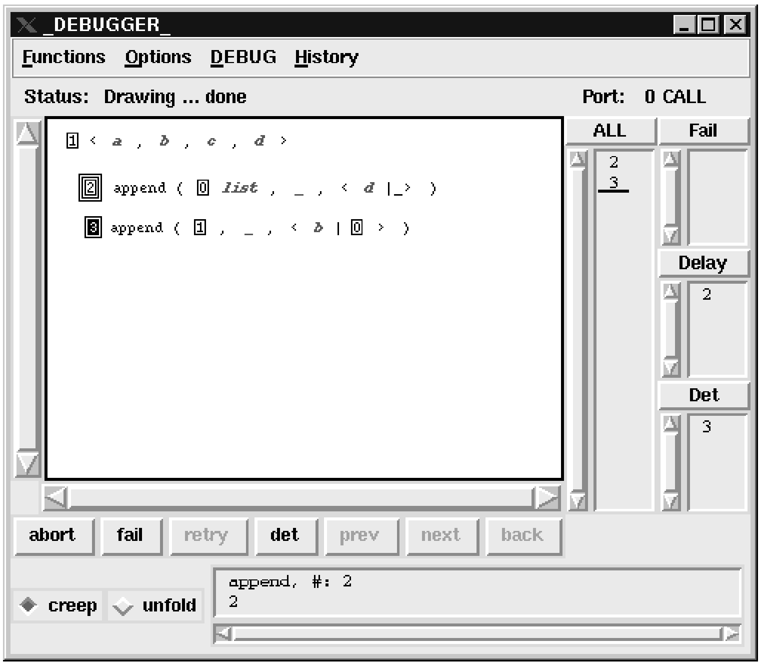


Figure 7.5: `inspect(([a,b,c,d],append(_,[b|append(_,[d|_])]))).`

displayed in the info area. Figure 7.4 shows the state after pressing the det button. We can see that the query is solved, there are no more remaining goals. This tells us that the query only involved deterministic computation. We can try to induce backtracking, but as there are no choice points left, the debugger aborts and prints the message:

```
ConTroll Debugger: No more solutions!
```

Let us now consider a more complicated query:

```
inspect(([a,b,c,d],append(_,[b|append(_,[d|_] ) ] ) ) ).
```

For the purposes of this example, we assume that append is delayed with the following statement:

```
delay(append, (arg0: @list ; arg1: @list)).
```

Figure 7.5 shows the initial debugger display with the creep radio button pressed. The two append goals are shown as tags 2 and 3. The first one (2) is delayed, since neither the first nor the result argument are instantiated. Consequently, 2

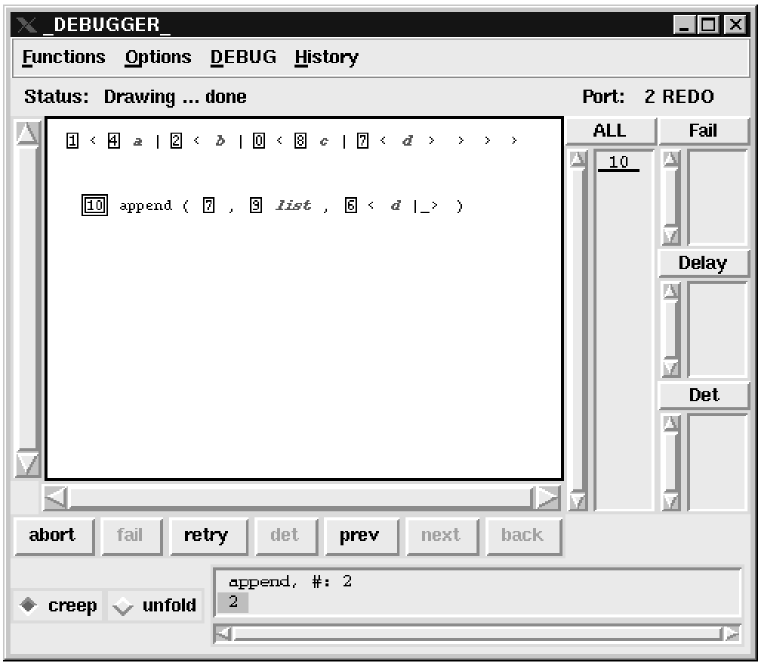


Figure 7.6: A REDO state

is listed in the Delay list, and the box is shown with a double frame. The other append goal (3) is deterministic, since only the second append clause matches it. The tag 3 is thus listed under Det, and is displayed with a quadruple frame in the main display area.

If we now chose the first clause of the selected goal 3 times in a row, we end up with a solution. This time, as we made some non-deterministic choices, inducing backtracking with fail produces a result, namely the REDO state shown in fig. 7.6. On the first try, we used the first clause of append, so now only the second one remains to be tried. We expect no more solutions in this branch of the search space and would like to see exactly where search fails. To be able to do so, we select the unfold radio button. This is necessary as in creep mode, the debugger always closes a state under deterministic computation, which includes backtracking on failure. In creep mode, it is thus not possible to exactly localise a failure point.

After having pressed the unfold button, we select the only available clause of the only goal. The resulting state is displayed in fig. 7.7. The only remaining append goal (13) is failed as it unifies with neither of the two clause heads. It is listed in the Fail list, and the tag in the main display area is shown white on a black background. Pressing the fail button induces further backtracking, but no more solutions exist.

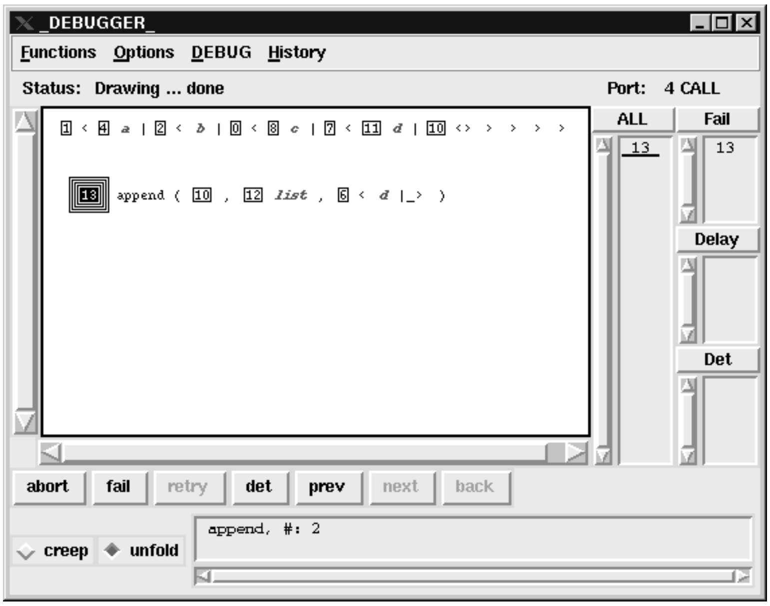


Figure 7.7: A state with a failed goal

The ConTroll debugger can also be used in text mode. Type '`h.`' at the debugger prompt for a short description of the available commands.

Chapter 8

The Organization of grammar files in ConTroll

To organize grammars in a modular fashion it is important to be able to distribute the grammar into several files to permit modification, loading, and testing of the different parts of a grammar separately. Also, only a modular file organization allows for distributed grammar development since such an organization makes it possible to coordinate the work on the individual files with software engineering tools such as a revision control system. ConTroll supports the use of a grammar configuration file which contains basic directory and file specifications, as well as a more sophisticated system allowing the linguist to specify the dependencies between signature and theory files.

8.1 Basic file and directory specification

To find out which signature and theory is supposed to be used and in which directories the files are supposed to be found, the system looks for a grammar configuration file called `config.pl` in the grammar directory. If this file is not found, the default names `signature` and `theory` are used for the signature and the theory files. If there is a configuration file, it can specify the list of theory and signature files to be used, as well as the directories where these files are to be found in the following way:

```
core_theory([<TheoryFileName1>,...,<TheoryFileNameN>]).
core_signature([<SignatureFileName1>,...,<SignatureFileNameM>]).
```

If the signature files are to be kept in a separate directory, this directory can be specified by including a statement of the form:

```
signature_dir(<DirectoryName>).
```

While this provides very basic file and directory specifications, it forces the grammar writer to explicitly list all the files to be loaded for the different parts of the grammar to be used. The downside of this is that each time one wants to load a part of the grammar, e.g. for testing, one nees to realize which files needed are needed to test this part of the grammar in order to be able to list them explicitly. While this might seem like a trivial task, our experience has shown that in a distributed grammar development environment, such a complete specification can require significant insight into the entire grammar.

8.2 File dependencies

ConTroll provides a more way tod specify the files needed for a module by allowing statements which make the dependencies between the different theory, signature, and lexicon files explicit. These specifications were modelled after the Makefiles used in some programming environments.

The first kind of statement specifies which theory and signature files a certain theory file depends on, i.e., they will be loaded and compiled in case that theory file is included in the `core_theory/1` list.[1]

```
theory_dep(+Theory filename, +List of dependent grammar files,
                             [],
                             +List of dependent signature files).
```

The second kind of file dependency statement specifies which signature files a certain signature file depends on.

```
sig_dep(+Signature filename, +List of dependent signature files).
```

Once the dependencies are provided in such a way, specifying which parts of the grammar should be loaded can be done without having to being aware of the whole grammar organization, since only one file for each module of the grammar needs to be included in the `core_theory` declaration. All signature, theory, and lexicon files which are needed for the selected modules are then automatically loaded according to the dependency specifications. The list of files used is displayed to make the result of file specification transparent to the user.

Example Assume we have structured a simple HPSG grammar into different files according to the different ID schemata. Then we can specify for the different schemata which lexical entries are relevant and which subsignatures introducing the relevant phrasal subtypes as well as the nonempty and empty word subtypes are needed in the following way:

[1] The third argument of `theory_dep/4` is reserved for the lexicon files of the lexical rule compiler to be included in the next version of ConTroll. It currently has to be specified as `[]`.

```
theory_dep(core_principles,[hfp,sem_p],[],[]).

theory_dep(hc_id,[verbs,nouns],[],[hc_phrase_sig]).
theory_dep(vc_id,[all_verbs],  [],[vc_phrase_sig]).
theory_dep(fh_id,[nfp,trace],  [],[fh_phrase_sig]).

theory_dep(all_verbs,[verbs,vc_verbs],[],[ne_word_sig]).
theory_dep(trace,    [],              [],[e_word_sig]).

sig_dep(hc_phrase_sig,[basic_hc_phrase_sig]).
sig_dep(vc_phrase_sig,[basic_hc_phrase_sig]).
```

In case we want to load the whole grammar, it then suffices to specify the following core statements:

```
core_signature([basic_signature]).
core_theory([core_principles,         % Principles
                                      % ID Schemata:
             hc_id,                   % a) head-complement
             vc_id,                   % b) verbal complex
             fh_id,                   % c) filler-head

             dc,                      % definite clauses, macros
             ]).
```

To instead load only the grammar without the verbal complex and the nonlocal dependencies, one comments out the `vc_id` and the `fh_id` line. This simple change removes all theory, lexicon, and signature files no longer needed, but keeps those in which are still needed for the remaining schema.

8.3 File search paths and the directories used by ConTroll

File search paths are a method offered by Prolog to associate a symbol, the so called 'file search path', with a list of directories. Using that method, apart from local and global filenames, a local filename wrapped with a file search path is also a valid way to reference a file. When such a filename is used, the file is searched in the list of directories associated with the file search path until the file is found.

For example, one can associate the file search path `grammar` with the directories `/a`, `/home/b` and `./c`. If one then asks prolog to compile the file `my_gram` in the grammar file search path by typing `[grammar(my_gram)]`, the system first tries to load the file `my_gram` from directory `/a`. If this file does not exist in that directory, it looks in directory `/home/b`, and finally in `./c`.

8.3.1 File search paths used by ConTroll

The file search path mechanism is used in ConTroll to located the grammar files and the files output by the compiler. The following file search paths are relevant to the grammar writer:

Grammar directory: file search path `grammar` Directory containing the theory specified by the linguist. Set at startup to the directory ConTroll was started from.

Signature directory: file search path `sig` Directory containing the signature specified by the linguist. Unless set by a `signature_dir/1` command in a configuration file, this file search paths defaults to the grammar directory.

Compiled directory: file search path `compiled` Set at startup to be the directory `grammar('.compiled')`. If that directory does not yet exist, the system asks whether it should create the directory or a directory at some other location.

8.3.2 Changing file search paths

ConTroll provides commands to set and change file search paths. Compared to the base functionality provided by, these commands in addition

- provide some error messages,
- make some type checks on the arguments,
- check whether the directories exist and are read/writable and if this is not the case asks whether it should be created.

Commands of interest to the grammar writer are:

- `file_search_path(+Abbrev,-DirName):` queries for the directories defined for `Abbrev`. Succeeds once for every directory defined for `Abbrev`.
- `append_search_path(+Abbrev, +DirName):` adds `DirName` as first directory to be searched for search path Abbrev.
- `appendz_search_path(+Abbrev, +DirName):` adds `DirName` as last directory to be searched for search path Abbrev.
- `change_search_path(+Abbrev, +DirName):` removes all previous directories for `Abbrev` and adds `DirName` as only directory.

- `new_search_path(+Abbrev, +DirName)`: like `change_search_path/2`, but checks that `Abbrev` was not previously defined. If `Abbrev` is already defined, the user is asked whether the definition should be replaced.
- `set_grammar_fallback(+DirName)`: keeps the first directory searched for search path *grammar*, removes the others, and adds `DirName` as second directory searched for search path *grammar*.

Chapter 9

The XTroll interface

Two practical problems arise 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 grammar constraints tends to get very complex for realistic linguistic theories. To deal with the presentation problem, ConTroll relies on the graphical user interface XTroll, which we will introduce in this section. To deal with the second problem, a sophisticated graphical debugger is included in ConTroll, which is described in §7.3.

The XTroll GUI allows the user to interactively view AVMs, search attributes or values in those representations, compare two representations (e.g. multiple results to a query) and highlight the differences, etc. Fonts and Colors can be freely assigned to the attributes and types. The displayed structures (or any part of it) can be can be printed or saved as postscript file. The GUI comes with a clean backend interface and has already been used as frontend for other natural language applications, e.g., in the VERBMOBIL project.

9.1 Mapping AVMs to tree output

A special feature of XTroll is that it offers a mechanism for displaying feature structures as trees according to user specified patterns. Note that displaying trees is not an obvious display routine in ConTroll, since the system does not impose a phrase structure backbone but rather allows a direct implementation of HPSG grammars which usually encode the constituent structure under DTRS or some similar attribute. Since trees are a very compact representation allowing a good overview of the structure, XTroll allows the user to specify that certain paths under a type are supposed to be displayed in a tree structure. As labels for the tree nodes, XTroll can display a user definable selection of the following: the feature path to the node, the type of the structure, the phonology, and finally an abbreviation resulting from matching user specified feature structure patterns. An example for such a tree output is shown in Fig. 9.1. In this

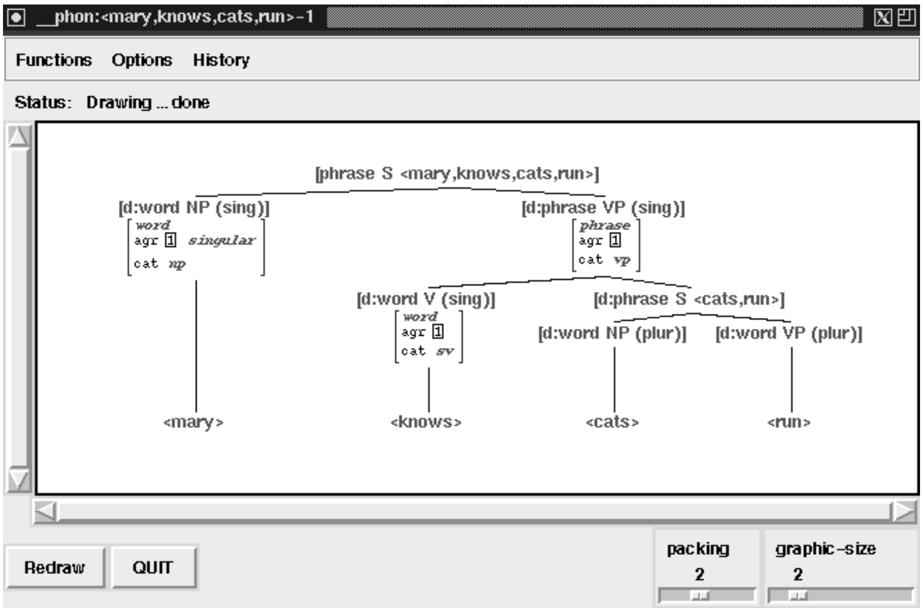


Figure 9.1: Screen shot of a window showing user configured tree output

tree, the abbreviations were used to display category information in an X-bar fashion. Clicking on the labels displays the AVM associated with this node. In the example, we did open three of the nodes to show the modification going on between the adjective *schnelles (fast)* and the noun *fahrrad (bike)*. Note that those attributes which are irrelevant to this matter were hidden by clicking on those attributes.

9.1.1 Specifying tree output

To obtain tree output like the one shown above, one needs to include the following specifications in one of the grammar files.

Node descriptions and paths to daughters

To determine which objects and paths are to be represented as a tree, for each kind of object which is intended to be a node in a tree one needs to include a description of the form:

```
ptree_desc(<Desc>,[<Path1>,...,<PathN>]).
```

where <Desc> is a description of the object and <Path1> to <PathN> are the paths to the daughters of that node.

Sometimes one does not encode each daughter under a separate attribute, for example when the number of complements in a construction can vary. Rather, the daughters are encoded in a list. To provide tree output for such a case, one does not need to include all paths to possibly occurring list members, like `dtrs:tl:hd, dtrs:tl:tl:hd, ...`. If the system encounters a list under one of the paths specified, the list members are automatically included as daughters. Note that also the last tail of a list is displayed, since otherwise the user could not tell from the tree whether all list members are shown (in which case the tail is the empty list).

Example To show trees for the small example grammar shown in §C.1, one wants to have phrases and their dtrs as tree nodes. The daughters of phrases can be found under the attributes *dtr1* and *dtr2*. This is expressed by the following statement:

```
ptree_desc(phrase,[nonhead_dtr,head_dtr]).
```

Labels for the nodes

To be able to label each tree node with an abbreviation of the linguistic object represented by that node, for each kind of object which is intended to be a node in a tree, the following mapping needs to be provided:

```
ptree_cat(<Abbrev>,<Desc>).
```

where <Abbrev> is the Prolog atom which should be used to abbreviate objects described by <Desc>.

The system uses only one abbreviation, namely the abbreviation of the first matching description. One should therefore include `ptree_cat/2` statements in the grammar such that the more specific descriptions appear before the more general ones.

Example For our example grammar, we could, for example, show the categories as node labels in our trees. To do this, we would include the following statements.

```
ptree_cat('S',(cat:s)).
ptree_cat('VP',(cat:vp)).
ptree_cat('V',(cat:(v;sv))).
ptree_cat('NP',(cat:np)).
```

If one also wants to see the agreement information at all but the S nodes, one would specify the following.[1]

```
ptree_cat('S',(cat:s)).
ptree_cat('VP (sing)',(cat:vp,agr:singular)).
ptree_cat('VP (plur)',(cat:vp,agr:plural)).
ptree_cat('V (sing)',(cat:(v;sv),agr:singular)).
ptree_cat('V (plur)',(cat:(v;sv),agr:plural)).
ptree_cat('NP (sing)',(cat:np,agr:singular)).
ptree_cat('NP (plur)',(cat:np,agr:plural)).
```

Phonology

Since the phonology (or morphology) of a sign is an important property of a node in a tree, it is displayed below each terminal node. For this purpose, one needs to specify, where the relevant information can be found:

```
phon_feat(<Featurename>).
```

where `<Featurename>` is an attribute the value of which is a list encoding the phonology. Note that currently only a simple feature, not a path is allowed. Also, a grammar should only contain one `phon_feat/2` statement. If no statement is provided, ConTroll uses `phon` as default.

On non-terminal nodes, it is possible to display the phonology in addition to the abbreviating labels. To do so, for each tree node that is supposed to show its phonology value, one should specify a statement of the form

```
ptree_phon(<Desc>).
```

where `<Desc>` is a description of a linguistic object represented as a tree node.

Example For the example grammar, no `phon_feat/1` declaration is needed, since the phonology is encoded under the attribute `phon`, which is the default location. But it naturally doesn't hurt to include the following specification:

```
phon_feat(phon).
```

To also show the phonology on a nonterminal node, for example all sentential nodes, one can provide the following `ptree_phon/1` declaration.

```
ptree_phon((cat:s)).
```

[1]The ConTroll distribution contains the `ConTroll/demo/one` grammar including the tree output specifications discussed here in directory `ConTroll/demo/tree-output`.

9.2 Keybindings

Commands which prompt for user interaction before they are executed are marked with ....

9.2.1 XTroll window

delete selected data ...	`C-d`
deiconify prolog window	`C-p`
quit XTroll ...	`C-q`

9.2.2 Prolog window

Motion

move forward over character	`C-f` or →
move backward over character	`C-b` or ←
go to beginning of line	`C-a` or `Pos1` or `Home`
go to end of line	`C-e` or `End`

Command history

go backwards in command history	`C-p` or ↑
go forwards in command history	`C-n` or ↓

Text deletion

kill line after cursor	`C-k`
yank last kill	`C-y`
delete next character	`C-d` or Del
delete previous character	`C-h` or Backspace

If partial input has been provided in the command line, typing `C-p` will go backwards in those commands in the history which complete the input.

Miscellaneous

iconify prolog window	`C-z`
send interrupt to prolog	`C-c`

9.2.3 AVM/Tree windows

redraw	`C-l`
start text search from top...	`C-s`
start text search from bottom...	`C-r`
show next match	`C-n`
show previous match	`C-p`
hide attributes ...	`C-h`
expand reentrancies ...	`C-e`
quit this window ...	`C-q`

9.2.4 Debugger window

redraw	`C-l`
start text search from top...	`C-s`
start text search from bottom...	`C-r`
show next match	`C-n`
show previous match	`C-p`
hide attributes ...	`C-h`
expand reentrancies ...	`C-e`

9.3 Mouse Interaction

Apart from the selection of menu and submenu items, the following things can be done using the mouse.

9.3.1 XTroll window

- open an avm or tree window (the preferred representation set in Menu Options):
  double click with left mouse button on line in scroll window
- open an avm or tree window (the dispreferred representation):
  double click with right mouse button on line in scroll window
- delete a data package:
  select a line in scroll window using left mouse button, then type `C-d` or chose menu item *Delete selected data* from *Functions* menu.

9.3.2 Prolog window

- selecting text in the window:
  dragging mouse over text while pressing the left mouse button

- extending the selection:
  clicking in text to be included with left mouse button while holding the shift key

- pasting a selection into the prolog or some other window:
  middle mouse button

- note: clicking in the command line to position the cursor (currently) is not supported

9.3.3 AVM/Tree window

Middle mouse button

Holding the middle mouse button while moving the mouse moves the displayed picture in the window.

Left mouse button

Clicking the left mouse button over the different displayed elements pops up a menu displaying the possible interaction with this element:

- type:
  - Show hidden features (by selecting the corresponding subitem)
- attribute:
  - Hide/Show substructure
  - Select node as root
- reentrancy (numbered box):
  - Hide/Show substructure
  - Select node as root
- tree node:
  - toggle structure, i.e. show AVM below node
  - show structure (this window)
  - show structure (new window)

Right mouse button

Clicking the right mouse button has the following effect on the different elements displayed:

- attribute: hide attribute
- reentrancy (numbered box): open/close reentrancy
- tree node: show AVM corresponding to this node

9.3.4 Debugger window

Middle mouse button

Holding the middle mouse button while moving the mouse moves the displayed picture in the window.

Left mouse button

Clicking the left mouse button over the different displayed elements pops up a menu displaying the possible interaction with this element:

- type:
  - Show hidden features (by selecting the corresponding subitem)
- attribute:
  - Hide/Show substructure
  - Select node as root
- reentrancy (numbered box): executes command (unfold or creep) for the first matching clause of the selected goal in the debugger.[2]
- tree node:
  - toggle structure, i.e. show AVM below node
  - show structure (this window)
  - show structure (new window)
- numbers displayed under **All**, **Fail**, **Delay**, **Det**:
  check which defining clauses match this goal, this information is shown in the clause display (horizontal window in the bottom right corner)

[2] Note that only clicking on goals corresponding to goals which can be executed will result in legal debugger commands. Others will only result in a warning in the Prolog window.

- the buttons **abort**, **fail**, **retry**, **det**, **prev**, **next**, **back**:
  select the corresponding debugger command
- the radio buttons **creep** and **unfold**:
  select the debugger action performed when clicking on a reentrancy
- numbers in clause display (horizontal window in the bottom right corner):
  executes command (unfold or creep) for the selected clause of the displayed goal in the debugger.

Right mouse button

Clicking the right mouse button has the following effect on the different elements displayed:

- attribute: hide attribute
- reentrancy (numbered box): open/close reentrancy
- tree node: show AVM corresponding to this node
- numbers under **ALL**,**Fail**, **Delay**, **Det**:
  show all matching clauses of selected goal
- numbers in clause display (horizontal window in the bottom right corner):
  show selected matching clause of the displayed goal in the debugger.

9.4 Menus

The XTroll interface makes it possible to attach often used prolog commands to menu items. When such a menu item is chosen, the command associated with that menu item is entered in the prolog window, just as if it had been typed.[3] Menu-items can also be selected using the keyboard by holding the Meta-key and typing the underlined letter occurring in the menu item to be chosen.

The following menu items with prolog commands are currently set up for ConTroll:

9.4.1 File

- ConTroll state
  - Save: `save_state(compiled(state)).`
  - Load: `load_state(compiled(state)).`

[3] Currently the command is not entered in the command history though, which will be changed in a future release of XTroll.

- Load compiled grammar: `reload.`
- Change file search path
  - grammar: `change_search_path(grammar,`
  - compiled: `change_search_path(compiled,`
- Quit Xtroll: `xtroll_quit.`

9.4.2 Options

- Show data
  - Immediately: `option_set(show_data).`
  - Upon request: `option_unset(show_data).`
- List abbreviation
  - on: `options:option_set(list_abbrev).`
  - off: `options:option_unset(list_abbrev).`
- Echo detected strings
  - on: `assert(show_string_message).`
  - off: `retractall(show_string_message).`
- Latex output
  - on: `set_option(latex_output,on).`
  - off: `set_option(latex_output,off).`
- Remaining delays action
  - start debugger: `set_option(show_unresolved,full).`
  - report number (default): `set_option(show_unresolved,simple).`
  - simply fail: `set_option(show_unresolved,nothing).`
- Trace queries
  - no (default): `set_option(interpreter,par_ind_det).`
  - only choice points:
    `set_option(interpreter,tracing),`
    `set_option(tracing_interpreter,no_det).`
  - choice points & delayed deterministic goals:
    `set_option(interpreter,tracing),`
    `set_option(tracing_interpreter,delayed_det).`
  - all calls:
    `set_option(interpreter,tracing),`
    `set_option(tracing_interpreter,all).`
- Test Sequence
  - with phonology: `phon_result_on.`
  - without phonology (default): `phon_result_off.`

9.4.3 Compiler Options

- Expand Det Goals
  * on (default): `det_expand_on.`
  * off: `det_expand_off.`
- Complex Antecedent Elimination
  * simple disjuncts: `disjoint_disjuncts_off.`
  * disjoint disjuncts (default): `disjoint_disjuncts_on.`
- Compiler Messages
  * more: `unification_warnings.`
  * less (default): `no_unification_warnings.`
- Debug on show failed
  * on (default): `set_option(debug_det_failed,on).`
  * off: `set_option(debug_det_failed,off).`
- Save strings in compiled(strings)
  * on: `save_strings_on.`
  * off (default): `save_strings_off.`

9.4.4 Work

- Load & compile grammar: `compile_user_grammar.`
- Load & compile signature only: `compile_user_sig.`
- Reload and process
  * delay patterns: `reload_source_and_process_delay_patterns.`
  * tree-output specification: `reload_source_and_process_tree_config.`
- Expand deterministic constraints: `deterministically_expand_constraints.`
- Show ptree_descs: `show_ptree_descs.`
- Show ptree_cats: `show_ptree_cats.`
- Show ptree_phons: `show_ptree_phons.`
- Show phon_feat: `show_phon_feat.`

Bibliography

Calcagno, M., D. Meurers, and C. Pollard (In prep). On the nature of lexical rules in head-driven phrase structure grammar. Ms. (Version: 9. August 1996).

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.

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.

Gerdemann, D., T. Götz, J. Griffith, S. Kepser, and F. Morawietz (1995, October). *Troll manual* (draft ed.). Seminar für Sprachwissenschaft, Universität Tübingen.

Gerdemann, D. and P. J. King (1993). Typed feature structures for expressing and computationally implementing feature cooccurence restrictions. In *Proceedings of 4. Fachtagung der Sektion Computerlinguistik der Deutschen Gesellschaft für Sprachwissenschaft*, pp. 33–39.

Gerdemann, D. and P. J. King (1994). The correct and efficient implementation of appropriateness specifications for typed feature structures. In *Proceedings of COLING-94*, Kyoto, Japan.

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*, Marseille, France.

Götz, T. and W. D. Meurers (1997a). 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.

Götz, T. and W. D. Meurers (1997b). Interleaving universal principles and relational constraints over typed feature logic. In *Proceedings of the 35th Annual Meeting of the ACL and the 8th Conference of the EACL*, 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.

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

King, P. J. (1994). An expanded logical formalism for head-driven phrase structure grammar. Arbeitspapiere des SFB 340 Nr. 59, Universität Tübingen.

Meurers, W. D. (1994). On implementing an HPSG theory – aspects of the logical architecture, the formalization, and the implementation of head-driven phrase structure grammars. In: Erhard W. Hinrichs, W. Detmar Meurers, and Tsuneko Nakazawa: *Partial-VP and Split-NP Topicalization in German – An HPSG Analysis and its Implementation.* Arbeitspapiere des SFB 340 Nr. 58, Universität Tübingen.

Meurers, W. D. (1995). Towards a semantics for lexical rules as used in HPSG. In *Proceedings of the Conference on Formal Grammar*, Barcelona. Also in: Proceedings of the ACQUILEX II Workshop on Lexical Rules, 1995, Cambridge, UK.

Meurers, W. D. and G. Minnen (1995). A computational treatment of HPSG lexical rules as covariation in lexical entries. In *Proceedings of the Fifth International Workshop on Natural Language Understanding and Logic Programming*, Lisbon, Portugal.

Meurers, W. D. and G. Minnen (1996). Off-line constraint propagation for efficient HPSG processing. In *HPSG/TALN Proceedings*, Marseille, France.

Meurers, W. D. and G. Minnen (1997). A computational treatment of lexical rules in HPSG as covariation in lexical entries. *Computational Linguistics 23*(4).

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

Nilsson, H. (1995). The external storage facility in SICStus Prolog. Technical report R91:13, Swedish Institute of Computer Science.

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 *Pro-*

ceedings of the Eighth International Conference on Logic Programming, pp. 443–456. MIT Press.

Shieber, S. M. (1986). *An Introduction to Unification-Based Approaches to Grammar*. Number 4 in CSLI Lecture Notes. Chicago: 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.

Appendix A

EBNF syntax specification

```
Program         ::= ε | ProgramStat Program
ProgramStat     ::= Implication. | DefSpec | ControlStat.
Implication     ::= FeatureTerm ==> FeatureTerm |
                    ⟨ImplicationID⟩ === FeatureTerm ==> FeatureTerm
DefSpec         ::= TypeDecl Clauses
TypeDecl        ::= PredSpec **> ⟨Type⟩.
PredSpec        ::= ⟨PredName⟩ | ⟨PredName⟩( TypeVector )
TypeVector      ::= ⟨Type⟩ | ⟨Type⟩, TypeVector
Clauses         ::= ε | Clause Clauses
Clause          ::= Atom := FeatureTerm.
Atom            ::= ⟨PredName⟩ | ⟨PredName⟩( TermVector )
TermVector      ::= FeatureTerm | FeatureTerm, TermVector
FeatureTerm     ::= ⟨Type⟩ | ⟨Variable⟩ | Atom |
                    ⟨Feature⟩: FeatureTerm | ~ FeatureTerm |
                    FeatureTerm , FeatureTerm |
                    FeatureTerm ; FeatureTerm |
                    (FeatureTerm) | [] | [ TermVector ]|
                     [ TermVector | FeatureTerm ]
ControlStat     ::= Delay | TermExp | FeatureExp | TreeSpec | AccessRelSpec
Delay           ::= delay( ⟨Type⟩ , DelayTerm )|
                    delay( ⟨PredName⟩ , DelayTerm )|
                    delay_deterministic( ⟨Type⟩ )|
                    delay_deterministic( ⟨PredName⟩ )
DelayTerm       ::= BasicDelayTerm | StarDelayTerm | (DelayTerm)
BasicDelayTerm  ::= ⟨Type⟩ | @ ⟨Type⟩ | ⟨Feature⟩: DelayTerm
```

		DelayTerm , DelayTerm | DelayTerm ; DelayTerm
StarDelayTerm	::=	Path * BasicDelayTerm | Path :(Path * BasicDelayTerm)
TermExp	::=	⟨Type⟩ `<<` ⟨Type⟩.
FeatureExp	::=	⟨Feature⟩ `<<<` ⟨Feature⟩.
TreeSpec	::=	`ptree_desc(` ⟨PrologAtom⟩ , FeatureTerm). |
		`ptree_cat(` FeatureTerm). | `ptree_phon(` ⟨Feature⟩).
AccessRelSpec	::=	`access_suffix(` ⟨PrologAtom⟩ , ⟨Type⟩). |
		`access_rule(` ⟨Type⟩ , Path , ⟨Type⟩).
Path	::=	⟨Feature⟩ | ⟨Feature⟩ : Path

Appendix B

Options

Options can be set with the command `set_option/2`, where the first argument is a ConTroll option, and the second one is a parameter for that option. Each option has its own set of legal paramaters. Options are set to a default value at start-up, and can then be set by the user. You can test an option with the command `controll_option/2`, where the first argument is the name of the option and the second a prolog variable which is instantiated to the current option setion. There are menu items in XTroll to set some options.

In the following, we give a short description of each option including the name of the option, the parameters it can take, the default setting, and a short description of the options influence on the behaviour of ConTroll.

Option: `must_mark_multi`

Parameters: `yes` or `no`

Default setting: `no`

Description: In a type hierarchy, the second and all further occurences of a type can be marked with `&`. Setting this option to `yes` makes this obligatory. It helps to avoid accidental multiple inheritance, through typos etc.

Option: `compiler_write_strings`

Parameters: `on` or `off`

Default setting: `off`

Description: When this option is set to `on`, the compiler issues a message every time it reads in a string type from the grammar. This can be useful for debugging purposes, e.g., to find places where a misspelt type is read in as string.

Option: `disjoint_disjuncts`

Parameters: `on` or `off`

Default setting: `on`

Description: As explained in Sec. 6.1, the elimination of complex antecedents involves a step where parts of the antecedent of an implication are moved to the consequent: `(t,A) ==> B` $\leadsto$ `t ==> ~A;(A,B)`. When the option is set to `on`, then `~A` is reduced so that the resulting disjuncts are all mutually incompatible. See Sec. 6.1 for explanation.

Option: `det_expand`

Parameters: `on` or `off`

Default setting: `on`

Description: By default, the compiler performs a deterministic closure operation on the compiled program. Each body of a clause is closed under deterministic unfolding. Switching off the option may be useful for debugging purposes.

Option: `debug_det_failed`

Parameters: `on` or `off`

Default setting: `on`

Description: This option determines the behaviour of the commands `show_failed/0` and `show_failed/1`. By default, ConTroll calls the debugger with the relevant clause, so that the user can trace the cause of the failure. If the option is set to `off`, the failed clause is simply displayed.

Option: `show_unresolved`

Parameters:

- `nothing`
- `simple`
- `full`

Default setting: `simple`

Description: This option instructs the interpreter what to do when it comes to a point in computation where there are no more undelayed goals. If the option is set to `nothing`, the computation will fail and induce backtracking. Under the default setting `simple`, the interpreter will display a message and ask for further instructions. Under the option `full`, the interpreter calls the debugger with the state that it is in. This enables the user to look for the source of the error with all the tools that the debugger provides.

Option: `latex_output`

Parameters: `on` or `off`

Default setting: `off`

Description: When set to on, this option induces the system to write result AVMs to a file in addition to printing them to the screen. The resulting LaTeX source file is then latexed, and xdvi is called to display the result. Only one file is used for the output, so save the ones you want to keep.

Option: `debugger_redraw`

Parameters:

- `expand_reentrancy`
- `redraw`
- `off`

Default setting: `expand_reentrancy`

Description: This option controls the behaviour of the debugger window when displaying a new state. The default is `expand_reentrancy`, which is a redraw with display of the value at the first occurence of every reentrancy. `redraw` does a simple redraw, without expansion of reentrancies. Since redrawing is expensive, it is sometimes useful to switch this option `off`. Expansion of reentrancies (including a redraw) can be forced manually with C-e, or via the file menue.

Appendix C

Example files

C.1 A simple example grammar `demo/one`

C.1.1 Signature `demo/one/signature`

```
type_hierarchy

bot
  sign cat:cat agr:number phon:list
    phrase dtr1:sign dtr2:sign
    word
  cat
    np
    vp
    v
    vpt
    sv
    s
  number
    singular
    plural
  string
.
```

C.1.2 Theory `demo/one/theory`

```
phrase ==> ( ( cat: s,
                 dtr1: (np_sign(Num),
                      phon: Phon1),
                 dtr2: (cat: vp,
```

```
                        agr: Num,
                        phon: Phon2))
                ; ( cat: vp,
                agr: Num,
                  dtr1: (cat: sv,
                        agr: Num,
                        phon: Phon1),
                  dtr2: (cat: s,
                        phon: Phon2))
             ;( cat: vp,
                agr: Num,
                  dtr1: (cat: v,
                        agr: Num,
                        phon: Phon1),
                  dtr2: (cat:np,
                        phon: Phon2)
             )),
% A. with append in relational notation:
             phon: Phon3, append(Phon1,Phon2,Phon3).

% B. with append in functional notation:
%            phon: append(Phon1,Phon2).

word ==> ( (np_sign(singular),
            phon: [(john ; mary)])
         ; (np_sign(plural),
            phon: [(cats ; dogs)])
         ; (cat:vp,
            phon: [(runs ; jumps)],
            agr: singular)
         ; (cat:vp,
            phon: [(run ; jump)],
            agr: plural)
           ; (cat: sv,
            phon: [(knows ; thinks)],
            agr: singular)
          ;(cat:v,
            phon:[(loves)],
            agr:singular)).

% A. append/3 in relational notation

append(list,list,list) **> bot.
append([],L,L) := bot.
append([H|T],L0,[H|L]) := append(T,L0,L).
```

```
% B. append/3 in functional notation
%
% append(list,list) **> list.
%
% append([],List) := List.
% append([H|T],L) := [H|append(T,L)].

np_sign(number) **> sign.
np_sign(Num) := sign, cat:np, agr:Num.
```

C.2 An example for the access relation mechanism

```
%% Some examples to help explain the accessRelation mechanism:

%% Some expansion rules and suffix declarations:

access_rule(cat,head,head).
access_rule(loc,cat,cat).
access_rule(synsem,loc,loc).
access_rule(sign,synsem,synsem).

access_suffix(head,"_h").
access_suffix(cat,"_c").
access_suffix(loc,"_l").
access_suffix(synsem,"ASuffixWithoutUnderscore").
access_suffix(sign,"_s").

%%% Some access relation (:==) definitions:

fin_h  :== (vform:fin).
nfin_h :== (vform:nfin).
acc_h :== (case:acc).
nom_h :== (case:nom).
dat_h :== (case:dat).
gen_h :== (case:gen).

marking_c :== (marking:unmarked).

no_bkg_l :== (context:background:eset).
no_quants_l :== (content:quants:[]).

cont_conx_l(X-cont,Y-conx) :== ((content:X), (context:Y)).
```

```
test_h((test_c(X),testB_d(X))-t1,X-t2) :== (case:(X,testC_h(X))).

head_c(X-head) :== (head: X).


%%% To try out the following warnings and errors, uncomment the
%%% relevant line(s) of code:

%%% I. Some examples producing WARNINGS:
%%% 1) A real relation already defined always has priority
%%% over newly computed access:

%%% a) (acc_h/1 is defined as :== relation above and would like to
%%%     become a real := relation

%acc_h := testkrams.

%%% b) (gen_h/1 is defined above and access_rules would like to
%%%     produce real := relations gen_h, gen_c, gen_l, gen_y, and gen_s)

%gen_l := testkrams.


%%% 2) A :== relation for a predicate which has already been
%%%    a) translated to a := relation or
%%%    b) access expanded
%%% because it was defined earlier in the text is not translated
%%% into a real relation and no rules are applied:

%%% a) (nom_h/0 is defined above)

%nom_h :== trallala.

%%% b) (acc_h/0 is defined above and access_rules produce acc_y, ... )

%dat_l :== dubidu.

%%% ... this is even the case if the two predicates differ in arity:
%%% (head_c/1 is defined above)

%head_l(X-ta,Y-tb,Z-tc) :== (loc:cat:head:(X)).


%%% II. Some examples producing ERRORS:

%%% 1) A type occuring in an access_rule is not allowed to
```

```
%%% a) have multipe suffixes defined for it

%access_rule(test,feature1,typ1).
%access_suffix(test,"_a").
%access_suffix(test,"_b").

%%% b) have no suffixes defined for it

%access_rule(typeWithoutSuffixDeclaration,feature1,typ1).

%%% 2) Two suffixes defined must not

%%% a) be identical

%access_suffix(w,"abcd").
%access_suffix(u,"abcd").

%%% b) have a common ending

%access_suffix(c,"dkd").
%access_suffix(c,"kd").
%access_suffix(c,"d").
%access_suffix(c,"adkd").
```

C.3 An example for a configuration file

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Id: config.pl,v 1.14 1996/03/09 13:19:33 dm Exp dm
%% Configuration file for the PVP Grammar (np version)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 % DIRECTORIES FOR THE LEXICON AND THE SIGNATURES
 % - if nothing is specified, the grammar directory is used.
 % - specified relative to grammar(.)

lexicon_dir(lexicon).
signature_dir(signatures).


 % THE SIGNATURES TO BE LOADED
 % (in order of dependencd, i.e. first a signature and then those
 % which depend on it!)
```

```
core_signature([
                signature
                ]).

 % THE THEORY FILES TO BE LOADED
core_theory([
              macros,
              dc,
              computation,

              get_phon,
                                                % ID Schemata
              hc_id,                            % a) head-complement
              hsc_id,                           % b) head-subject-complement
              fh_id,                            % c) filler-head
              % ah_id,                          % d) adjunct-chead
              % sh_id,                          % e) specifier-head
              mh_id,                            % f) marker-head

              % e_word
              ]).


 % THE LEXICON TO BE LOADED (Constraints on the lexicon_type)
core_lexicon([
              % nouns
              ]).


 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 % GRAMMAR DEPENDENCIES
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 % 1. THEORY DEPENDENCIES
 %
 % theory_dep(TheoryFileName,
 %             list of dependent theory files,
 %             list of dependent base lexicon files,
 %             list of dependent signature files)

theory_dep(hc_id,[proper_names],[verbs],[hc_sig]).
theory_dep(hsc_id,[proper_names],[verbs],[hsc_sig]).
theory_dep(fh_id,[],[],[fh_sig]).
theory_dep(ah_id,[],[ah_lex],[ah_sig]).
theory_dep(sh_id,[],[sh_lex],[sh_sig]).
theory_dep(mh_id,[],[mh_lex],[mh_sig]).
```

```
theory_dep(e_word,[sh_id],[],[e_word_sig]).
theory_dep(proper_names,[],[],[lex_phrase_sig]).


 % 2. LEXICON DEPENDENCIES
 %
 % lexicon_dep(LexiconFileName,
 %             list of dependent theory files,
 %             list of dependent base lexicon files,
 %             list of dependent signature files)

lexicon_dep(ah_lex,[],[],[ne_word_sig]).
lexicon_dep(sh_lex,[],[],[ne_word_sig]).
lexicon_dep(mh_lex,[],[],[ne_word_sig]).
lexicon_dep(verbs,[get_phon],[],[ne_word_sig]).
lexicon_dep(nouns,[],[],[ne_word_sig]).


 % 3. SIGNATURE DEPENDENCIES
 %
 % sig_dep(SignatureFileName,
 %         list of dependent signature files)

sig_dep(hsc_sig,[struc_phrase_sig]).
sig_dep(hc_sig,[struc_phrase_sig]).
sig_dep(fh_sig,[struc_phrase_sig]).
sig_dep(ah_sig,[struc_phrase_sig]).
sig_dep(sh_sig,[struc_phrase_sig]).
sig_dep(mh_sig,[struc_phrase_sig]).
```

Appendix D

Reporting bugs, known bugs, and future releases

D.1 Reporting bugs

In case you encounter what appears to be a ConTroll bug, please report it to

`controll_support@sfs.nphil.uni-tuebingen.de`.

In order for us to track down the bug, we need to obtain certain information. To report a bug, therefore please follow these four steps:

1. Save a state of the ConTroll system: Under the XTroll interface, use the ConTroll_state:save item under the file menu. Else, type

   ```
   save_state(compiled(state)).
   ```

   in the ConTroll window. The full name of the compiled state is echoed by the system. By default, it will be the file `state.sic` or `state.qui` (depending on whether you're using sicstus or quintus prolog) in the `.compiled` directory which located in the directory where you started ConTroll. In the following steps, ***`filename`*** should be replaced by the path and name of the state you saved.

2. Since the resulting state is rather big, compress it by typing

   `gzip` *`filename`* or
   `compress` *`filename`*

   to a UNIX prompt, depending on which of the two utilities is available on your system.

3. To obtain a format which can be sent by email, the resulting compressed file needs to be uuencoded. To do so, type

   `uuencode` *`filename`*`.gz` *`filename`*`.gz >` *`filename`*`.uu`   or
   `uuencode` *`filename`*`.Z` *`filename`*`.Z >` *`filename`*`.uu`

   depending on whether you used `gzip` or `compress`.

4. Send an email to

   `controll_support@sfs.nphil.uni-tuebingen.de`

   containing a short description on how to reproduce the error and the file *`filename`*`.uu`.

D.2 A list of known bugs and problems

1. The XTroll debugger window can only display relational atoms of up to 10 arguments (including the result argument). Attempts at displaying bigger terms will result in wrong displays or system crash.
2. The debugger's *unfold* command does not unfold deterministic goals the same way the interpreter does. Specifically, the interpreter goes unfolds them in a breadth-first manner, avoiding some potential non-termination. The debugger does not do this. In some strange grammars, it can thus happen that the debugger shows an infinite loop, where the interpreter in fact terminates. The *creep* command uses exactly the same order of execution as the interpreter.
3. The cursor sometimes disappears from the Prolog window of the XTroll interface. The cursor should reappear when you move the mouse out of the Prolog window and then back again. This is a tcl/tk problem which hopefully one of the future releases of these packages will repair.
4. If the user defines features called LIST, E_LIST, or NE_LIST and the abbreviating list output is switched on, then tree output will not work correctly.

D.3 Plans for future releases

In the next releases of ConTroll, we plan to include the following:

- (re)integrate lexical rule compiler
- (re)integrate incremental compiler
- introduce phrase structure and parser as special relational constraint
- use our own abstract machine