TRALE Reference Manual

DRAFT

Mohammad Haji-Abdolhosseini
Dept. of Linguistics
University of Toronto
130 St. George St. Room 6076
Toronto M5S 3H1
Ontario, Canada
mhaji@chass.utoronto.ca

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

Contents

Chapter 1

ALE Data Structure

Internally, ALE represents a feature structure as a term of the form `Tag-Sort(V1,...,Vn)` where `Tag` represents the token identity of the structure using a Prolog variable, `Sort` is the name of the type of the structure. The terms `V1` through `Vn` are the values for the features `F1` through `Fn` that are appropriate for the type `Sort`. The features are sorted alphabetically based on their names in `V1...Vn`. This results in the kind of record structure presented in Figure 1.1 for feature structures.

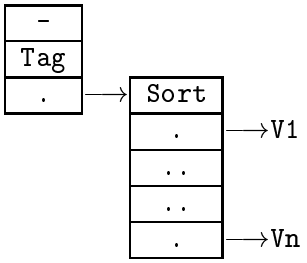

Figure 1.1: Internal representation of `Tag-Sort(V1,...Vn)`

When a type is promoted, `Tag` is replaced with a new pair of `Tag-Sort(V1,...,Vm)` with `m` possibly larger than `n` in order to make room for any new features introduces by the new type. For example, given the type signature in Figure 1.2, the internal ALE representation of the feature structure in (1) will be (2).

(1)
$$\begin{bmatrix} \textbf{head} & \\ \text{MOD} & \text{plus} \\ \text{PRD} & \text{minus} \end{bmatrix}$$

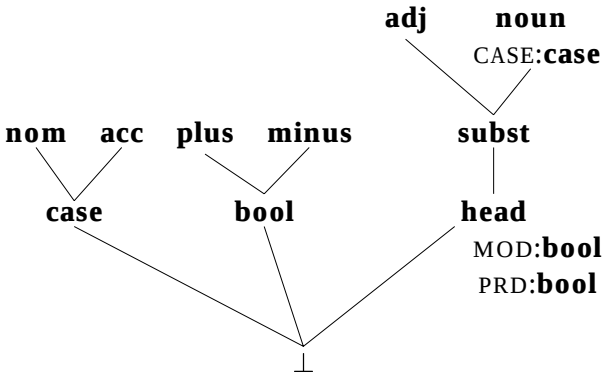

Figure 1.2: A sample type signature

(2) `Tag-head(X-plus,Y-minus)`

But when this type is promoted to **noun**, the ALE representation of that type is updated to (3).

(3) `Tag2-noun(Z-case,X-plus,Y-minus)-head(X-plus, Y-minus)`

Note that in this example, the variable `Tag` of (2) has been bound to `Tag2-noun(Z-case,X-plus,Y-minus)` in (3), and now we have a new tag (`Tag2`) at the beginning of the reference chain. The positional encoding of feature values in this manner means that, at compile time, we know which features any two types have.

1.1 Path Compression

As feature structures get updated, reference chains get very long with certain signatures and thus may become slower to process. ALE remedies this situation by performing an operation called *path compression* to reduce the size of reference chains before an edge is added to a chart and before a feature structure is output to the user. The predicate `fully_deref/4` is responsible for path compression. In `fully_deref(RefIn,SVsIn,RefOut,SVsOut)`, `RefIn` and `SVsIn` are the input reference (or `Tag`) and sort values respectively, and `RefOut` and `SVsOut` will be the output reference and sort values. The path compression algorithm of `fully_deref/4` is described in Table 1.1.

As an example, consider the following:

Condition	Action
RefIn is a variable	RefIn = fully(RefOut,SVsOut)-SVsOut, run fully_deref/4 on arguments of RefIn
RefIn is not a variable	if RefIn == fully(Ref,_), then this means that RefIn has already been processed by fully_deref/4; thus, RefOut = Ref and SVsOut = SVs if RefIn == Ref-SVs, then call fully_deref(Ref,SVs,RefOut,SVsOut).

Table 1.1: Path-compression algorithm

```
| ?- fully_deref(Tag1-s1(Tag2-t2-t3,Tag3-t4-t5),
                 SVsIn,TagOut,SVsOut).

Tag1 = fully(TagOut,s1(_A-t2,_B-t4))-s1(_A-t2,_B-t4),
Tag2 = fully(_A,t2)-t2,
Tag3 = fully(_B,t4)-t4,
SVsOut = s1(_A-t2,_B-t4)
```

The result of running path-compression on

```
Tag1-s1(Tag2-t2-t3,Tag3-t4-t5)
```

is TagOut-SVsOut. The reason for using fully/2 is that we want to know if a path has already been compressed or not due to cycles in the feature structure that it represents.

1.2 Unification Algorithm

ALE uses the Martelli-Montanari algorithm, which is a variation of the union algorithm for union-find data structures. The Martelli-Montanari algorithm operates on a finite set of equations

$$E = \{s_1 = t_1, \ldots, s_n = t_n\}$$

and a substitution θ. Initially $E = \{s = t\}$, where s and t are the terms to be unified, and $\theta = \emptyset$. Then the algorithm chooses from E an arbitrary equation and performs acts according to the following rules:

	Equation from E	**Action**
1	$f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$	replace by the equations $s_1 = t_1, \ldots, s_n = t_n$
2	$f(s_1, \ldots, s_n) = g(t_1, \ldots, t_n)$	fail
3	$X = X$	delete equation
4	$X = t$ or $t = X$ where X does not occur in t	add $X \longleftarrow t$ to θ, apply the substitution $\{X \longleftarrow t\}$ to E and the term in θ
5	$X = t$ or $t = X$ where X occurs in t and $X \neq t$	fail

The above procedure is repeated until E becomes empty, or the procedure fails.

Feature structure unification in ALE is done by `ud/2` predicate, which first *dereferences* the two feature structures to be unified getting the most recent term encoding of the feature structure together with its tag, and then it runs the Martelli-Montanari algorithm on the term encodings.

Chapter 2

Signature files

W. Detmar Meurers, Ohio State University
Thilo Götz, Dale Gerdemann, Eberhard-Karls-Universität Tübingen [1]

T2.1 Signature Input Format

The input to the signate is one or more files containing information about a type hierarchy and the appropriateness conditions. 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`. (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.[2] The subtypes 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.[3]

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 (`&`).[4] A type name must always start with a lower case letter.

[2] 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!

[3] 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.

[4] 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.

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 θ 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**]
.

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. Respecifying 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.

Chapter 3

Signature Compilation

Efficiently unifying two objects or types, which corresponds to finding their most general common extension in a partially ordered set of objects/types, is of central importance for ensuring the efficient processing of typed feature structures in general. Penn (2000) shows that every operation necessary for computing the closure of attributed type signature specifications in the logic of typed feature structures (Carpenter, 1992) can be reduced to matrix arithmetic. ALE uses these matrix operations to efficiently compute type inheritances and least upper bounds at compile time.

3.1 Subsumption Matrices and Transitive Closure

It has been shown (Aït-Kaći et al., 1989) that partially ordered types can be represented in the form of a bit-vector encoding.

Definition 1 *Given a partially ordered set, $\langle P, \sqsubseteq \rangle$, and a total ordering of P's elements, $p_1, p_2, \ldots, p_{|p|}$, the* subsumption matrix, *S, of P is a $|P| \times |P|$ Boolean matrix, where $S_{i,j} = 1$ iff $p_i \sqsubseteq p_j$.*

We can use the ith row of S to encode the type p_i, with unification corresponding to a bit-wise AND. For instance, the subsumption matrix of the type hierarchy in Figure 3.1 is given in Figure 3.2. The AND of the rows for a and d yields the row e, for example.

In practice, however, we do not initially have access to this information and we need to compute that. What we do have access to is the immediate subsumption relation that is specified in type subsumption declarations,

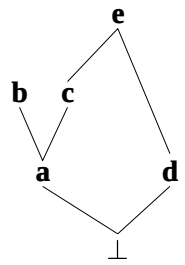

Figure 3.1: An example partial order of types

	⊥	a	b	c	d	e
⊥	1	1	1	1	1	1
a	0	1	1	1	0	1
b	0	0	1	0	0	0
c	0	0	0	1	0	1
d	0	0	0	0	1	1
e	0	0	0	0	0	1

Figure 3.2: The subsumption matrix, S, of Figure 3.1

with reflexive transitive closure being implicit. We can build a base subsumption matrix, H, in the same way, by using the immediate subsumption relation. The question is then how to obtain S from H quickly. The base subsumption matrix in Figure 3.2 is given in Figure 3.3.

	⊥	a	b	c	d	e
⊥	0	1	0	0	1	0
a	0	0	1	1	0	0
b	0	0	0	0	0	0
c	0	0	0	0	0	1
d	0	0	0	0	0	1
e	0	0	0	0	0	0

Figure 3.3: The base subsumption matrix, H, of Figure 3.1

One way to achieve this is a reflexive-transitive closure, by directly filling in the diagonal of H with 1s (reflexive) and multiplying the result by itself until it reaches a fixed point, S (transitive). This fixed point is reached after no more than $|P|$ iterations.

3.2 Monoids, Rings, Quasi-Rings, and Semirings

To understand the underlying mathematics of signature compilation, we need to understand the algebraic structures of *monoids*, *rings*, *quasi-rings*, and *semi-rings*.

Definition 2 *A* monoid *is a structure* $\langle P, \cdot, e\rangle$ *such that:*

- P *is a set closed under* $\cdot$*:* $a \cdot b \in P$*, for all* $a, b \in P$*,*
- $\cdot$ *is an associative binary operator:* $a \cdot (b \cdot c) = (a \cdot b) \cdot c$*, for all* $a, b, c \in P$*, and*
- $e \in P$ *is an identity for* $\cdot$*:* $e \cdot a = a \cdot e = a$ *for all* $a \in P$*.*

Definition 3 *A* quasi-ring *is a structure* $\langle P, \oplus, \otimes, \bar{0}, \bar{1}, \rangle$ *such that:*

- $\langle P, \oplus, \bar{0}\rangle$ *is a monoid,*
- $\langle P, \otimes, \bar{1}\rangle$ *is a monoid,*
- $\bar{0}$*, is an annihilator of* $\otimes$*:* $a \otimes \bar{0} = \bar{0} \otimes a = \bar{0}$ *for all* $a \in P$*, and*
- $\otimes$ *distributes over* $\oplus$*:* $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ *and* $(b \oplus c) \otimes a = (b \otimes a) \oplus (c \otimes a)$*, for all* $a, b, c \in P$*.*

Definition 4 *A* ring *is a quasi-ring with an additive inverse, i.e., for all* $a \in P$*, there exists* $b \in P$ *such that* $a \oplus b = b \oplus a = \bar{0}$*.*

If P is a quasi-ring, then multiplication of matrices is well-defined and has certain nice properties such as associativity and the existence of an identity.

Definition 5 *Given a quasi-ring,* $Q = \langle P, \oplus, \otimes, \bar{0}, \bar{1}\rangle$*, an* $m \times n$ *matrix,* A*, over* Q*, and an* $n \times p$ *matrix,* B*, over* Q*, then* $A \cdot B$ *(*matrix multiplication*) is the* $m \times p$ *matrix,* C *over* Q *such that:*

$$c_{i,j} = \bigoplus_{k=1}^{n} a_{i,k} \otimes b_{k,j}.$$

B_{XOR} and B_{OR} are both Boolean quasi-rings. B_{XOR} is also a Boolean ring; B_{OR} is not. B_{OR} is a closed Boolean semi-ring:

Definition 6 *A* closed semiring *is a quasi-ring,* $\langle P, \oplus, \otimes, \bar{0}, \bar{1}\rangle$*, such that:*

- $\oplus$ *is idempotent:* $a \oplus a = a$*, for all* $a \in P$*,*
- P *is closed under countable infinite summaries,* $a_1 \oplus a_2 \oplus \ldots$*, which are well-defined,*
- *associativity, commutativity, and idempotence extend to infinite summaries, and*
- $\otimes$ *distributes over infinite summaries.*

In a Boolean semiring, $\oplus$ corresponds to OR and $\otimes$ corresponds to AND. OR is idempotent, i.e., $1 \oplus 1 = 1$. This is vital for ensuring that matrix multiplication can compute a transitive closure, since transitively closed subsumption should not be 'turned off' by immediate subsumption chains on more than one subtyping branch. So we need idempotence in the underlying Boolean quasi-ring. XOR is not idempotent; so the Boolean ring is not the correct structure to use.

3.3 Topological Sorting

Before creating a subsumption matrix for a type hierarchy, ALE needs to topologically sort the types. The topological sorting of the types in a type hierarchy ensures the placement of more general types before more specific ones. This is performed through a depth-first traversal of the hierarchy. For example, to topologically sort the type hierarchy presented in Figure 3.1, we start at $\bot$. Once we have added $\bot$ to our list of topologically sorted types, we continue to **a**, then **b**. Since **b** does not have any subtypes, we go back to **a** and move to **c**. Carrying on in the same fashion and adding new types to the list will result in [$\bot$, **a**, **b**, **c**, **e**, **d**]. Building a subsumption matrix based on this sorting results in an *upper-triangular sparse matrix* meaning that the lower left quadrant of the matrix consists only of zeros as in figures 3.2 and 3.3.

3.4 Zero-Counting by Quadrants

The algorithm used in ALE is a specialized sparse matrix multiplication algorithm for semirings. Sorting of the rows and columns of the matrix takes place through the standard depth-first topological sorting algorithm

provided in the last section. In such matrices,[1] the transitive closure of the matrix can be calculated as:

$$\begin{pmatrix} A & B \\ 0 & C \end{pmatrix}^* = \begin{pmatrix} A^* & A^*BC^* \\ 0 & C^* \end{pmatrix}$$

An algorithm to efficiently calculate this has been specially developed to make use of the sparseness property of these matrices.

Two functions are used to recursively divide a matrix evenly into quadrants. For any i and n such that $1 \leq i \leq n$, let $d_n(i)$ be defined such that:

$$d_n(i) = \begin{cases} 0 & i = 1 \\ d_{\lceil n/2 \rceil}(i) + 1 & 1 < i \leq \lceil n/2 \rceil \\ d_{\lfloor n/2 \rfloor}(i - \lceil n/2 \rceil) + 1 & i > \lceil n/2 \rceil \end{cases}$$

In addition, for any $d \geq d_n(i)$, let $q_n^d(i)$ be defined such that:

$$q_n^d(i) = \begin{cases} n & d = 0 \text{ (thus } i = 1) \\ q_{\lceil n/2 \rceil}^{d-1}(i) & d > 0, i \leq \lceil n/2 \rceil \\ q_{\lfloor n/2 \rfloor}^{d-1}(i - \lceil n/2 \rceil) & d > 0, i > \lceil n/2 \rceil \end{cases}$$

One way of looking at them is as measures defined on a balanced binary tree with n leaves. Given the ith leaf from the left, there will be some subtrees for which i is the leftmost leaf. In that case, $d_n(i)$ is the least depth of such a subtree, and $q_n^d(i)$ is the total number of leaves that such a subtree at depth d has.

Given a matrix M, we shall say that a submatrix is *rooted* at $M[i, j]$ iff $[i, j]$ is its leftmost, uppermost coordinate in M.

Given i, j, m and n such that $1 \leq i \leq m$, and $1 \leq j \leq n$, let $v_{\langle m,n \rangle}(i, j) = \max(d_m(i), d_n(j))$. When we divide an $m \times n$ matrix M evenly into quadrants, then the largest quadrant rooted at $M[i, j]$ will be $q_m^{v_{\langle m,n \rangle}(i,j)}(i) \times q_n^{v_{\langle m,n \rangle}(i,j)}(j)$ in size. It can be proven that if m and n differ by no more than 1 in the original matrix, then these two dimensions will differ by no more than 1.

Now, given a Boolean matrix M, there is a unique matrix $Z(M)$ over the non-negative integers such that the value of $Z(M)[i, j]$ is the size of the largest zero-quadrant rooted at $M[i, j]$ in the largest quadrant rooted

[1] Note that the corresponding Boolean algebra for these matrices is a semiring where $1 \oplus 1 = 1$ because as mentioned above, in a Boolean semiring, the additive operator $\oplus$ corresponds to OR.

at $M[i,j]$. If this largest zero-quadrant is not square, then we use the larger of its dimensions as the value. As a simple example, consider the 4×4 identity matrix as M:

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad Z(M) = \begin{pmatrix} 0 & 1 & 2 & 1 \\ 1 & 0 & 1 & 1 \\ 2 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Note that the 1s in M are replaced by 0s in $Z(M)$—there are no zero-quadrants rooted at those coordinates. Also notice that many values of $Z(M)$ can be inferred from other values. The fact that $Z(M)[1,3]$ is 2, for example, tells us that $Z(M)[1,4]$, $Z(M)[2,3]$, and $Z(M)[2,4]$ must be 1, and *vice versa*. It is perhaps useful to conventionally write $Z(M)$ with as few values as can be used to infer the rest of the matrix:

$$Z(M) = \begin{pmatrix} 0 & 1 & 2 & - \\ 1 & 0 & - & - \\ 2 & - & 0 & 1 \\ - & - & 1 & 0 \end{pmatrix}$$

This convention accentuates the sparseness of the original matrix M.

3.5 Prolog Representation of ZCQ Matrices

In Prolog, we represent the Zero-Counting-Quadrant (ZCQ) Matrices discussed above using the data structure presented in this section.

- `zcm(A,B,D,C)` is the basic data structure. Each argument in `zcm/4` stands for a quadrant in the matrix. The correspondence is shown in Figure 3.4. This is a recursive structure which means that each one of `A`, `B`, `C`, or `D` is itself a `zcm/4` or any of the other data structures presented below.

A	B
D	C

Figure 3.4: Matrix quadrants corresponding to arguments in `zcm(A,B,D,C)`

- The number `0` stands for a sparse matrix of 0s.
- `zcu(A,B,C)` represents an upper-triangular matrix as shown in Figure 3.5. Because it is an upper-triangular matrix, we already know that `D` is 0. The values of `A` and `C` need only be either `zcu/3` themselves or 1. The value of `B` can be `zcm/4` or any of the base cases mentioned below.

A	B
0	C

Figure 3.5: An upper triangular matrix corresponding to `zcu(A,B,C)`

- **Base Cases:**
  - The number `1` stands for a 1×1 matrix of 1.
  - `zc12(A,B)` is a 1×2 matrix shown in Figure 3.6

A	B

Figure 3.6: A 1×2 matrix corresponding to `zc12(A,B)`

  - `zc21(A,D)` stands for a 2×1 matrix shown in Figure 3.7

A
D

Figure 3.7: A 2×1 matrix corresponding to `zc21(A,D)`

3.6 Transitive Closure with ZCQ

To transitively close a matrix in its ZCQ-representation, we first recurse on its diagonal quadrants, as suggested by (*), to obtain A^* and C^*. We then compute A^*BC^* with two matrix multiplications. Matrix multiplication in ZCQ is given by the quadrant-based recursive formulation:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{pmatrix}$$

Summation in ZCQ is given by a coordinate-wise *min* operation. In addition to the size-one base cases, the efficient implementation of ZCQ includes in both summation and multiplication a *sparse case*, in which the value of $Z(M)$ is checked first against the dimensions of the submatrices being multiplied. In this context, the base case of multiplication thus always returns 0 (indicating a non-zero element).

3.7 Compiling Appropriateness Conditions

The closed Boolean semiring, B_{OR}, suffices for processing simple partial orders of types with no features, often called *type hierarchies*. For type signatures with feature appropriateness conditions and value restrictions on those features, this is not enough however.

We can think of 0 and 1 in B_{OR} as constituting a very small type hierarchy, as shown in Figure 3.8. If $\top$ corresponds to 1, and $\bot$ to 0, then unification in this hierarchy corresponds to Boolean OR. We can also write this as in Figure 3.9, in which the trivial type hierarchy, consisting of just $\bot$, has been *bottom-lifted* to add a new bottom, $\underline{\bot}$. Because bottom-lifting preserves meet-semi-latticehood, we can trivially extend unification, $\sqcup$, to any $P \cup \{\underline{\bot}\}$ where P is a finite meet semi-lattice. Now, we need something to correspond to AND:

$$a \underline{\sqcup} b = \begin{cases} \underline{\bot} & \text{if } a = \underline{\bot} \text{ or } b = \underline{\bot} \\ a \sqcup b & \text{otherwise} \end{cases}$$

Figure 3.8: The Boolean type hierarchy

Figure 3.9: The trivial type hierarchy lifted to produce the Boolean hierarchy

Definition 7 *Let $\langle P, \sqsubseteq \rangle$ be a finite meet semi-lattice. Then $Q(P) = \langle P \cup \{\underline{\bot}\}, \sqcup, \underline{\sqcup}, \underline{\bot}, \bot \rangle$, is the* closed semiring *induced by P.*

Notice that we can define this for all P, not just the trivial type hierarchy, because in all type hierarchies, $\sqcup$ and therefore its extension to $P \cup \{\underline{\bot}\}$ and to $\underline{\sqcup}$, are total functions, and there is a least element. As can easily be verified:

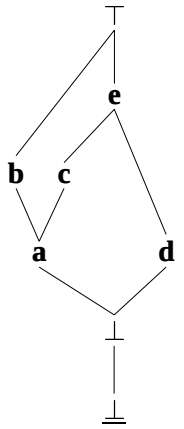

Figure 3.10: The induced closed semiring, $Q(P)$, constructed from Figure 3.1

Proposition 1 *For all finite meet semi-lattices P with a greatest element, $Q(P)$ is a closed semi-ring.*

The existence of a greatest element ensures that $\sqcup$ and $\underline{\sqcup}$ are closed in $P \cup \{\underline{\bot}\}$. Without loss of generality, we can assume that the greatest element, $\top$, does not explicitly appear in P, and that it does not occur anywhere else in the signature, e.g., in appropriateness conditions. $\top$ can be smashed onto any such P, and is typically implemented as type unification failure in the original signature. Figure 3.10 shows the type hierarchy in Figure 3.1 $\underline{\bot}$-lifted and $\top$-smashed to form its induced closed semiring.

The benefit of using $P(Q)$ is that it allows us to generalize to other computations on signatures that require matrices with types in them rather than just 0s and 1s. The subsumption matrix of P can still be constructed using $\underline{\bot}$ and $\bot$ in place of 0 and 1, respectively.

3.7.1 Feature Introduction

In the logic of typed feature structures, appropriateness conditions take the form of a restriction on which types can bear a particular feature, and which types that feature's value can have.

Definition 8 *A* type signature *is a quadruple, $\langle P, \sqsubseteq, F, A\rangle$, where $\langle P, \sqsubseteq\rangle$ is a finite meet semi-lattice, F is a set of features, and $A : F \times P \rightharpoonup P$ is a partial function such that:*

- *(Feature Introduction)*
  for every $\text{F} \in F$*, there is a most general type* $intro(\text{F}) \in P$ *such that* $A(\text{F}, intro(\text{F}))\downarrow$,[2] *and*

- *(Upward Closure / Right Monotonicity)*
  if $A(\text{F}, t)\downarrow$ *and* $t \sqsubseteq u$*, then* $A(\text{F}, u)\downarrow$ *and* $A(\text{F}, t) \sqsubseteq A(\text{F}, u)$.

$A(\text{F}, t)$ is defined on those types, t, that can bear the feature F, and its value is a type that must subsume the type of the value of F at t.

Feature Introduction eliminates a source of disjunction in inferring types from feature names in descriptions. By having a unique introducing type, it is possible to apply appropriateness in the other direction immediately to infer some of the other features that must also exist, along with the types of their values.

In practice, signature declarations specify appropriateness only by declaring (1) where a feature is introduced, along with the type its value must have and (2) where a feature takes on a value whose type cannot be inferred to be the least type that satisfies Upward Closure and/or Right Monotonicity given its value on supertypes. The type to which a feature's value is constrained is called a *value restriction*. Figure 3.11, for example, is Figure 3.1 with appropriateness declarations added. F is appropriate to

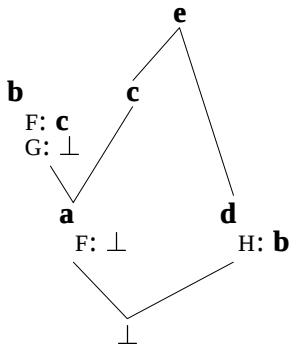

Figure 3.11: An example type signature

a, for example, with value restriction, $\bot$. Because F is appropriate to **a**, it is also appropriate to **b**, **c** and **e**, although **b** refines the value restriction to **c**. **b** has two appropriate features because it also introduces G. **e** has two appropriate features by Upward Closure because H was introduced at **d**.

[2] $\downarrow$ means *defined*.

	F	G	H
$\bot$	$\underline{\bot}$	$\underline{\bot}$	$\underline{\bot}$
a	$\bot$	$\underline{\bot}$	$\underline{\bot}$
b	c	$\bot$	$\underline{\bot}$
c	$\underline{\bot}$	$\underline{\bot}$	$\underline{\bot}$
d	$\underline{\bot}$	$\underline{\bot}$	b
e	$\underline{\bot}$	$\underline{\bot}$	$\underline{\bot}$

Figure 3.12: The value declaration matrix of Figure 3.11

In order to enforce this view of appropriateness conditions on P, we can build a matrix over $Q(P)$ for these declarations:

Definition 9 *Given a set of types, P, a set of features, F, and a partial function of appropriateness declarations $D : F \times P \rightharpoonup P$, the* value declaration matrix *for D over F and P is a $|P| \times |F|$ matrix, V, over $Q(P)$, in which $V_{i,j} = u$ if there exists a $u \in P$ such that $D(\mathrm{F}_j, t_i) = u$, and $V_{i,j} = \underline{\bot}$ if there is no such u.*

The uniqueness of u, when it exists, is guaranteed by the fact that D is a partial function. The value declaration matrix for Figure 3.11 is shown in Figure 3.12. The entry for type d, feature H is b because H is declared as appropriate to d with its value restricted to b.

Notice that $\underline{\bot}$ is being used here as a place-holder for pairs of type, **t** and feature, F, for which F is not appropriate to **t**. We use $\underline{\bot}$ rather than $\top$ so that feature introduction (with a value restriction of $\bot$ or greater) still respects Right Monotonicity.

Definition 10 *The* value restriction matrix *of P is $R = S^T \cdot V$.*

Premultiplying V by the transpose of S closes the appropriateness declarations under subsumption. $V_{i,j}$ is thus something other than $\underline{\bot}$ iff feature j is appropriate to type i. The value restriction matrix of Figure 3.11 is shown in Figure 3.13. Notice that the entry for type **e**, feature H is also **b** because **e** inherits H from **d**. Using the value restriction matrix, we can then express the condition on Feature Introduction:

Definition 11 *$\delta : P \cup \{\underline{\bot}\} \rightarrow \{\underline{\bot}, \bot\}$ is the* characteristic function *for P, such that:*

$$\delta(t) = \begin{cases} \bot & \text{if } t \in P, \\ \underline{\bot} & \text{if } t = \underline{\bot} \end{cases}$$

	F	G	H
$\bot$	$\underline{\bot}$	$\underline{\bot}$	$\underline{\bot}$
a	$\bot$	$\underline{\bot}$	$\underline{\bot}$
b	c	$\bot$	$\underline{\bot}$
c	$\bot$	$\underline{\bot}$	$\underline{\bot}$
d	$\underline{\bot}$	$\underline{\bot}$	b
e	$\bot$	$\underline{\bot}$	b

Figure 3.13: The value restriction matrix of Figure 3.11

	F	G	H
$\bot$	$\underline{\bot}$	$\underline{\bot}$	$\underline{\bot}$
a	$\bot$	$\underline{\bot}$	$\underline{\bot}$
b	$\underline{\bot}$	$\bot$	$\underline{\bot}$
c	$\underline{\bot}$	$\underline{\bot}$	$\underline{\bot}$
d	$\underline{\bot}$	$\underline{\bot}$	b
e	$\underline{\bot}$	$\underline{\bot}$	$\underline{\bot}$

Figure 3.14: The introduction matrix of Figure 3.11

Proposition 2 *R satisfies the feature introduction restriction iff for all i, there exists a j, such that $\vec{\delta}(R_i^T) = S_j$.*

δ projects the elements of $Q(P)$ back onto the closed Boolean semi-ring, according to whether they belong to P. Feature Introduction is satisfied iff, after component-wise projection, every column of R is the same as some row of S. Rows of S encode types as the upward closed sets that they subsume. The columns of R have non-$\underline{\bot}$ values for the types to which a feature, F, is appropriate; and we know that that set is upward-closed, having left-multiplied by S^T. If that set is one of the rows of S, then that row corresponds to a least type, which is $intro(\text{F})$.

Along the way, we can also compute those introducing types:

Definition 12 *The* introduction matrix, *I, of P is a $|P| \times |F|$ matrix in which $I_{j,i} = V_{j,i}$ when j is the j specified for i in Proposition 2, and $\underline{\bot}$ elsewhere.*

The introduction matrix for Figure 3.11 is given in Figure 3.14. The entry for type **b**, feature F is $\underline{\bot}$ because, although **b** places a non-inferable value restriction on F, it does not introduce F.

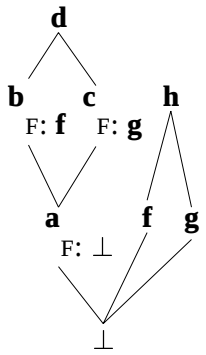

Figure 3.15: A type signature with consistent value restrictions

	F
⊥	⊥
a	⊥
b	f
c	g
d	⊥
f	⊥
g	⊥
h	⊥

Figure 3.16: The value declaration matrix of Figure 3.15

3.7.2 Value Restriction Consistency

Because of Right Monotonicity, join-reducible types can not only multiply inherit features, but also inherit value restrictions on the same feature from two or more different branches; and these must be consistent. Figure 3.15 shows an example of this. Right Monotonicity from **b** and **c** requires F to be appropriate to **d** with a value of both **f** and **g**. In Figure 3.15, this is consistent—the value of F at **d** must be of type **h**. Without **h**, it would not be consistent. We can use value restriction matrices to enforce this consistency check as well.

Proposition 3 *The value restrictions of P are consistent iff there is no i,j for which $R_{i,j} = \top$.*

$\top$ corresponds to inconsistency in the original signature.

The value declaration matrix for Figure 3.15 is given in Figure 3.16. The value restriction matrix of Figure 3.15 is given in Figure 3.17. Without **h**,

	F
$\bot$	$\underline{\bot}$
a	$\bot$
b	f
c	g
d	h
f	$\underline{\bot}$
g	$\underline{\bot}$
h	$\underline{\bot}$

Figure 3.17: The value restriction matrix of Figure 3.15

the entry for **d** would have been $\top$.

3.7.3 Appropriateness Cycles

All types must have a finite most general satisfier, a finite least informative feature structure of that type that respects appropriateness conditions. This means that appropriateness conditions may not conspire so as to require a feature structure of type **t** to have a proper substructure of type either **t** or a subtype of **t**. Very often, the appropriateness conditions will *allow* a subtype of **t** to occur, but that is not the same thing. Non-empty lists, for example, have a tail appropriate to lists, of which non-empty lists are a subtype. That makes lists a recursive data type, but lists still have finite most general satisfiers. The former kind of *appropriateness cycle* can very naturally be described using R.

Definition 13 *The* convolution matrix *of R, C, is a $|P| \times |P|$ matrix over $Q(P)$ such that $C_{i,j} = \bot$ if there exists a k such that $R_{i,k} = t_j$, and $C_{i,j} = \underline{\bot}$ otherwise.*

Proposition 4 *P has an appropriateness cycle iff there exists an i such that $C^*_{i,i} = \bot$, where C^* is the (non-reflexive) transitive closure of C.*

$C_{i,j} = \bot$ means that type t_j is accessible as a substructure by some feature from structures of type t_i. By transitively closing C, we extend that accessibility to finite paths of features, and so can detect whether t_i is accessible from t_i. Because we convoluted C from R, which was upward closed by left-multiplication with S^T, we detect accessibility to subtypes of t_i as well. The convolution of Figure 3.17 is given in Figure 3.18 (0 and 1 are used for readability). The entry for row b, column f is $\bot$, because f is

	$\perp$	a	b	c	d	f	g	h
$\perp$	0	0	0	0	0	0	0	0
a	1	0	0	0	0	0	0	0
b	0	0	0	0	0	1	0	0
c	0	0	0	0	0	0	1	0
d	0	0	0	0	0	0	0	1
f	0	0	0	0	0	0	0	0
g	0	0	0	0	0	0	0	0
h	0	0	0	0	0	0	0	0

Figure 3.18: The convolution of Figure 3.17

accessible from b along the feature F. In this case, the transitive closure of C is the same as C itself, because all types that occur as value restrictions of features are atomic, i.e., they have no features of their own.

In practice, this transitive closure can be computed directly from R, without explicitly constructing C.

3.7.4 Join Preservation Condition

Not all type signatures are statically typable. This means that there are some signatures for which the result of unifying two feature structures that obey appropriateness must be reverified as obeying appropriateness. Although these signatures are technically valid ones, systems that process with LTFS normally check for static typability because of the computational cost of this run-time verification. Signatures that are not statically typable are those that do not satisfy the following condition:

Definition 14 *A signature, $\langle P, \sqsubseteq, F, A\rangle$, satisfies the* join preservation condition *iff for all consistent $t, u \in P$ and $\mathrm{F} \in F$:*

$$A(\mathrm{F}, t \sqcup u) = \begin{cases} A(\mathrm{F}, t) \sqcup A(\mathrm{F}, u) & \textit{if } A(\mathrm{F}, t)\downarrow \textit{ and } A(\mathrm{F}, u)\downarrow \\ A(\mathrm{F}, t) & \textit{if only } A(\mathrm{F}, t)\downarrow \\ A(\mathrm{F}, u) & \textit{if only } A(\mathrm{F}, u)\downarrow \\ \textit{anything} & \textit{otherwise} \end{cases}$$

Proposition 5 *$\langle P, \sqsubseteq, F, A\rangle$ satisfies the join preservation condition iff for all i, j for which $J_{i,j} = \perp$, and $U_{i,j} = t_k$, $R_i \vec{\sqcup} R_j \vec{\sqcup} (S^T I)_k = R_k$.*

Viewed in terms of R, join preservation is a linear dependence condition among consistent types—recall that in $Q(P)$, $\sqcup$ corresponds to the additive operator. In a join-preserving signature, joins cannot add new information to the system, apart from introducing new features.

3.8 Subtype Covering

As discussed in the user's guide, TRALE assumes that subtypes exhaustively cover their supertypes. To recapitulate, let us review the example provided in the user's guide.

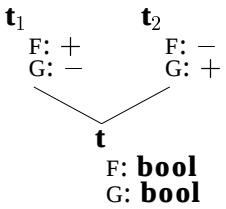

Figure 3.19: A sample deranged type signature

In the hierarchy depicted in Figure 3.19, there are no `t` objects with identically typed F and G values as in the following feature structures:

$$\begin{bmatrix} \text{F} & + \\ \text{G} & + \end{bmatrix} \quad \begin{bmatrix} \text{F} & - \\ \text{G} & - \end{bmatrix}$$

TRALE does not accept such undefined combinations of feature values as valid. However, if TRALE detects that only one subtype's product of values is consistent with a feature structure's product of values, it will promote the product to that consistent subtype's product. Thus, in our example, a feature structure:

$$\begin{bmatrix} \textbf{t} & \\ \text{F} & + \\ \text{G} & \textbf{bool} \end{bmatrix}$$

will be promoted automatically to the following. Note that the type itself will not be promoted to $\mathtt{t}_1$.

$$\begin{bmatrix} \mathbf{t} & \\ \mathrm{F} & + \\ \mathrm{G} & - \end{bmatrix}$$

This section shows how subtype covering is handled in TRALE. Types whose feature values are not exhaustively covered by their subtypes' feature values (such as `t` in the above example) are called *deranged* types.

At the highest level, TRALE performs the following steps during signature compilation in order to compute subtype covering.

1. Find all the subtypes `Sub` of supertype `Super`.
2. Create a base subsumption graph representation of the subtype relations found.
3. Topologically sort the graph and reverse the result so that more specific types come first (see section 3.3).
4. For each type `t`, find the number of maximal types that are subsumed by `t`.
5. For each type `t`, determine the status of every type for the purpose of classifying deranged types (to be discussed below).

These steps can be seen in the following code snippet:

```
compile_deranged_assert :-
  !,findall(Super-Sub,(non_a_type(Super),
                       imm_sub_type(Super,Sub)),SubGEdges),
  vertices_edges_to_ugraph([],SubGEdges,SubG),
  top_sort(SubG,RevSortedTypes),
  reverse(RevSortedTypes,SortedTypes),
  compute_maxcount(SortedTypes,SubG),
  compute_deranged(SortedTypes,SubG).
```

Here we focus on the last two steps, namely, `compute_maxcount/2` and `compute_deranged/2`.

3.8.1 Computing *maxcount* of each type

Maxcount counts the number of maximal types subsumed by every type. The *maxcount* of every maximal type as well as `a_/1` atoms is 1. The

$maxcount$ of a non-maximal type `t` is the cardinality of the union of the sets of maximal subtypes covered by the immediate subtypes of `t`, as given by the base subsumption graph. For example, in the type signature of Figure 3.20, the $maxcount$ of `d`, `e`, and `f` is 1 because they are maximal types. The $maxcount$ of `b` and `c` is 2 as they cover $\{$`d`, `e`$\}$, and $\{$`e`, `f`$\}$ respectively. The $maxcount$ of `a`, then, will be $|\{$`d`, `e`$\} \cup \{$`e`, `f`$\}| = |\{$`d`, `e`, `f`$\}| = 3$.

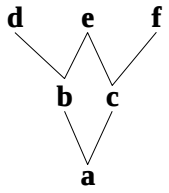

Figure 3.20: A sample type hierarchy with multiple inheritance

3.8.2 Classifying Deranged Types

Types are classified in two categories: *normal*, and *deranged*. Normal types are non-deranged types that can be treated as maximal (even if they are not) when checking a potentially deranged supertype. All maximal types are normal. Atoms, because they are downward-closed, are not deranged. We do not add an entry for atoms (including `a_` atoms) to the database.

As mentioned above, a deranged type is one whose map of appropriate value restrictions is not exhaustively covered by the maps of appropriate value restrictions of its maximally specific subtypes. When checking potentially deranged supertypes, these types must be replaced by a minimal covering of normal subtypes. The predicate `deranged/2` indicates the status of every type for the purpose of classifying deranged types.

A *map* is a product of types, such as that found in a set of appropriate features. $maxcountMap$ counts the number of maximal maps subsumed by every map. A *maximal map* is one that has a maximal type in every dimension (i.e., feature). $MaxcountMap$ of a map is the product of the $maxcount$ of every dimension. For example, type `t` in Figure 3.19 will have a $maxcountMap$ of four corresponding its four possible subtypes, only two of which are present as designated subtypes of `t`.

Then we need to identify which one of the maps cover `a_` atoms, and delete those from the root map. This is because the `a_` atoms have infinite subtypes.

We then replace all deranged types with their non-deranged subtype covers with `normalise_cover/2`.

Chapter 4

Description Compilation

The description compiler in ALE is responsible for compiling feature structure descriptions into Prolog code. Carpenter (1992) defines a description as follows:

Definition 15 (Descriptions) *The set of* descriptions *over the collection* $Type$ *of types and* $Feat$ *of features is the least set* $Desc$ *such that:*

- $\sigma \in Desc$ *if* $\sigma \in Type$
- $\top \in Desc$
- $\pi : \phi \in Desc$ *if* $\pi \in Path$, $\phi \in Desc$
- $\pi_1 \doteq \pi_2 \in Desc$ *if* $\pi_1, \pi_2 \in Path$
- $\phi \wedge \psi, \phi \vee \psi \in Desc$ *if* $\phi, \psi \in Desc$

The most basic kind of description is a simple type σ. Such a description applies to objects which are of the type σ. Since $\bot \in Type$, $\bot$ is a description, and it describes any feature structure while $\top$, which must be explicitly defined to be a description, is not satisfied by any feature structure. In practice, $\top$ is used to denote failure in unification. A description of the form $\pi : \phi$ applies to objects whose value for the path π satisfies the description ϕ. A description of the form $\phi_1 \doteq \phi_2$ is taken to mean that the value of the object that you get by following the path π_1 is structure-shared to the value of the object that you get by following the path π_2. Conjunction and disjunction are interpreted in the usual way. Descriptions are used in various places in a grammar including phrase-structure rules, lexical entries, constraints, macros, and procedural attachments.

Description compilation is performed in four steps as follows:

- Serialization
- Sorting
- Peephole optimization
- Code Generation

These steps are described in the following sections.

4.1 Serialization

The first thing that the compiler needs to do, during description compilation, is to translate a description into a series of instructions that tell the compiler what code needs to be generated. This step is called serialization.

Six kind of instructions are generated in this step. These are instructions for *types*, *variables*, *feature structures*, *inequations*, *functions*, and *disjunctions*. The syntax of these instructions is provided in Table 4.1.

Description	Instruction
Types	`type Path, Type`
Variables	`var  Path, Var`
Feature Structures	`fs   Path, FS`
Inequations	`ineq Path1, Path2`
Functions	`fun  Rel,  Arity, ArgPaths, ResPath`
Disjunctions	`or   Path, (Code1; Code2)`

Table 4.1: The syntax of serialized instructions

`Path` refers to the feature path that leads to the item in question. To illustrate, let us go through an example. Suppose we want to compile the feature structure below:

```
phrase,
  dtrs:(hc_struct,
        hdtr:synsem:local:cat:(head:H
                               subcat:append(NDS,MS)),
        ndtr:synsem:local:cat:subcat:NDS),
  synsem:local:cat:(head:H,
                    subcat:MS)
```

This feature structure will result in the following instructions. The numbers are added for ease of reference.

```
1: type [],phrase
2: type [dtrs],hc_struct
3: var  [head,cat,local,synsem,hdtr,dtrs],H
4: var  0,NDS
5: var  1,MS
6: fun  append,2,[0,1],
        [subcat,cat,local,synsem,hdtr,dtrs]
7: var  [subcat,cat,local,sysem,ndtr],NDS
8: var  [head,cat,local,synsem],H
9: var  [subcat,cat,local,synsem],MS
```

All instructions start with a *terminal path*, which is a path of length 0 refering to the root of a description. The terminal path is written as `[]`. Therefore, the first instruction means that the root of the feature structure being compiled is of type `phrase`. Instruction 2 states that there is a path from the root through `dtrs` that leads to the type `hc_struct`. Instruction 3 says that the path `[head,cat,local,synsem,hdtr,dtrs]` leads to variable `H`. Note that paths are written in the reverse order of what is customary in linguistics. Paths are discussed in more detail in the next section. Instructions 4 and 5 inform the compiler that two variables, namely, `NDS` and `MS` have been encountered and they have been assigned the numbers 0 and 1. In the instruction number 6, we see these variables being used inside a function. This instruction says that the path `[subcat,cat,local,synsem,hdtr,dtrs]` leads to a function called `append` with an arity of two, and that the function makes use of variables 0 and 1 that have been previously seen. The last three instructions show where else the previously seen variables are used.

4.2 Sorting

In this phase, the compiler goes through the instructions and using its knowledge about the type signature on which the grammar is based, it gathers all the necessary information about the description. Before we go any further, we need to define the concept of *mode*.

When compiling a description, ALE needs to keep track of the information that it has about that description at any given time. This information may not refer to just one feature structure sometimes. For instance, if a

description is used in the context of a phrase-structure rule, it may contain variables that have scope over the whole rule. All of the information that ALE has about a description within the current variable scope context (i.e., a phrase-structure rule, complex-antecedent constraints, or simply a line of ALE code) is referred to as *mode*. We can think of *mode* as a partial function from the union of all paths and variables to totally well-typed feature structures.

Definition 16 ***(mode)*** *Given a set of paths* Π*, a set of variables* V*, and a set of totally well-typed feature structures* TTF*, we have* $mode : \Pi \cup V \rightharpoonup TTF$.

In ALE, *mode* is represented as paths for terms. We call this *Mode*. A slightly different kind of *Mode* is used to keep track of the information ALE has about variables.

Definition 17 ***(VMode)*** *A* Variable Mode *(VMode) is a triple* $\langle FS, Flag, CBSafe \rangle$ *where:*

- FS *is a feature structure;*
- $Flag = \{seen, tricky\}$*;*
- $CBSafe = \{true, false\}$.

VMode holds for each variable a feature structure FS that is the most general satisfier of that variable depending on its context, a $Flag$ that says whether the variable is $seen$ or $tricky$ (to be discussed shortly), and a $CBSafe$ flag that says whether it is safe to bind that variable at compile-time or not.

When sorting, ALE starts with a minimal *Mode* (typically $\bot$), and begins going through the instructions adding information as necessary. Let us continue working through our running example from the previous section. The first instruction to process, then, is `type [],phrase`. What we do at this point is add the most general satisfier of `phrase` to *Mode* (in this case $\bot$). This results in the promotion of $\bot$ to `phrase` which also entails adding all the features introduced by `phrase` (in this case, only DTRS; see Figure 4.1). ALE goes through the other instructions similarly. When a path is encountered, as in the second instruction `type [dtrs], hc_struct`, ALE starts at the end of the path (i.e., the terminal path `[]`), and unifies the most general satisfier of the introducer of each feature in the path with the *Mode*. Note that each nonterminal element in the path refers to a feature. In order to access the value of that

feature, ALE needs to make sure that it has access to it, meaning that the *Mode* contains the most general satisfier of the introducer of that feature. That is why we add the most general satisfier of each element to the *Mode*. Each instruction, therefore, potentially changes the *Mode*. If an instruction fails to modify the *Mode*, it is considered redundant and thus eliminated. An example of this is instruction 2. In this case, ALE ends up not adding anything to the *Mode* because (assuming that our type signature includes what is depicted in Figure 4.1) all phrases have a DTRS feature with the value restriction **hc-struct**.

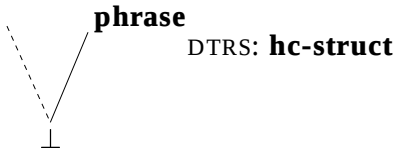

Figure 4.1: A portion of a type signature

After this overview, let us see how each kind of instruction is utilized to potentially modify the *Mode*. A summary of the actions that ALE takes on the face of each kind of instruction is provided in Table 4.2.

The arguments of each function are associated with two type specifications: an input type and an output type. Because ALE is a logic-programming language, each argument in a function can potentially be an input or an output argument. The type declarations for the arguments of functions are written as `InputType-OutputType` pairs. When the user specifies only a single type for the argument of a function, as in `Type`, ALE interprets that as `Type-Type`. Therefore,

```
foo(t,u,v).
```

is equivalent to

```
foo(t-t,u-u,v-v).
```

When executing `fun` instructions, ALE does not add the type of the input *Mode* of the function to the *Mode*. Instead, it just checks to see if the path corresponding to each argument in the function has the input *Mode*. For instance in the case of `append 2,[0,1]`, it only makes sure that the mode corresponding to the variables `0` and `1` has the type *list* in its *Mode*.

Earlier, we mentioned that the initial *Mode* typically starts with ⊥. This is true for most descriptions except type-antecedent constraints and relational calls. In the case of a type-antecedent constraint, `Type cons Cons`,

Instruction	Action
`type Path,Type`	Unify the most general satisfier of `Type` with the *Mode* of `Path`.
`var Path,Var`	Unify the mode of `Var` with the *Mode* of `Path`.
`fs Path,FS`	Unify `FS` with the *Mode* of `Path`.
`ineq Path1,Path2`	Do nothing.
`fun Rel,Arity,ArgPaths,ResPath`	First check `ArgPaths` to see if the most general satisfier of the input type of each argument subsumes the *Mode* of the path of that argument. Then unify the input path with the most general satisfier of the output of that argument.
`or Path,(Code1;Code2)`	Do the above with each disjunct. Then using finite-state automaton intersection and type generalization, combine the output *Mode* of each disjunct to get the output mode of the whole disjunction.

Table 4.2: Summary of actions taken by ALE, for each kind of instruction

the initial *Mode* is the most general satisfier of `Type`. In the case of a relational call, on the other hand, the initial *Mode* is the most general satisfier of the input *Mode*.

If during sorting adding a type to the *Mode* fails at some point, an error message is displayed to the user. If the failure happens within a disjunction however, we wait to see if the other disjunction fails as well. If it does, we complain to user; and if it does not, we will simply eliminate the dis-

junct that fails together with the disjunction operator.

4.3 Peephole Optimization

After all the instructions have been generated and all the missing information has been added to *Mode*, it is time to go through the instructions again and eliminate all the redundancies before any code is generated. This is achieved via a process called *peephole optimization* as follows. Starting at the top, ALE looks at pairs of instructions and try to either eliminate one or merge the two together. Then it moves down by one instruction and does the same to the next pair until it has considered all instructions.

- **Merging:**

  ```
  type Path, Type1
  type Path, Type2   ⟹   type Path, (Type1 ⊔ Type2)
  ```

  Given two instructions `type Path,Type1` and `type Path,Type2`, we know that the two types `Type1` and `Type2` must be unifiable because they share the same path; therefore, we can merge the two instructions into one: `type Path, (Type1 ⊔ Type2)`.

- **Deletion:**

  ```
  type Path, Type
  ...Path ...        ⟹   ...Path ...
  ```

  Given the instruction `type Path, Type` and another instruction containing `Path` to a feature `F`, we can eliminate the first instruction because we know `Type` $\sqsubseteq$ `intro(F)`. This means that when trying to get the value of `F` by a post-order traversal of `Path`, ALE has to process the instruction `type Path, Type` anyway.

4.4 Code Generation

The last step in description compilation is to generate Prolog code based on the instructions prepared during the previous steps. In the following, we present the kind of code that is generated based on each class of instructions.

- `type`:
  These instructions are translated into a series of `add_to_type` statements depending on the length of the path. For example, `type [dtrs], hc_struct` will generate `add_to_type(phrase,dtrs,hc_struct)`. Remember that ALE finds the most general satisfier of a description by first finding the most general satisfier of its terminal path, in this case `phrase`. The above `add_to_type` instruction says that we need to add the feature `dtrs` with the value restriction `hc_struct` to the type `phrase`.

  Two issues need to be pointed out at this point. As mentioned before, this specific example will not find its way to the Prolog code because `phrase` types already come with the feature `dtrs` with the value restriction `hc_struct`. The use of the example here is only for expository purposes. Second, in practice, ALE appends the type name with `add_to_type_` for efficiency. Therefore, the above example would have actually appeared as `add_to_type_phrase(dtrs,hc_struct)` in the generated Prolog code.

- `var`:
  Variables in an instruction are either seen or unseen. If a variable is seen, then a Prolog equality is generated. Some variables are *tricky*, however. This means that in a disjunction `(A;B)`, the variable `Var` has been used in `A`, and at compile time, it is not possible for ALE to determine whether `Var` is seen in `B` or not. In such cases, ALE generates a shallow cut as in

  ```
  var(Var) -> Var=FS; ud(Var,FS)
  ```

  in `B` to determine whether or not `Var` is seen at run time; if it is unseen it will then equate it with the feature structure that it refers to; otherwise, it will have to make a call to `ud/2`, which is unification with dereferencing.

  If a variable is unseen, then it is either *CBSafe* or not. Recall that a *CBSafe* variable is safe to be bound at compile time. In this case, ALE simply performs a unification at compile time using `ud/2`. If the variable is not *CBSafe*, however, it will use Prolog equality.

- `fs`:
  Feature structures are treated like seen variables, and therefore, they are handled with `ud/2` as well.

- `ineq`:
  Inequations are handled with Prolog's `dif/2` predicate, which constrains its arguments to represent non-unifiable values.

- `fun`:
  Function calls are directly transferred to Prolog as function calls.

- `or`:
  Disjunctions are handled as Prolog disjunctions.

4.5 ALE Lexical Rule Compilation

Lexical rules are formulated in the form of `lex_rule/2` predicates. These rules, which represent redundancies in the lexicon, operate on a subset of the items in the lexicon modifying their categories and/or their phonology/spelling. There are two lexical rule packages: one that belongs to ALE, and another that comes with the TRALE extension. This section deals with ALE lexical rules, which have the following format:

```
<RuleName> lex_rule <RewriteRule> morphs <MorphologicalRule>.
```

For more information on the syntax of lexical rules, refer to the user's guide.

During compilation, the predicate `lex_rule/8` is called, when appropriate, to close the lexicon under the available lexical rules. This process simply involves threading together the input and output descriptions with procedural attachments and the phonological mapping rules if they are specified. Phonological mapping rules are handled by `morph/3`. More specifically, `morph(Morphs,WordIn,WordOut)` converts `WordIn` to a list of characters; then it calls `morph_chars/3` to apply a pattern specified in `Morphs`, and finally it converts the resulting character list back to a word unifying it with `WordOut`. If `Morphs` contains more than one pattern, `morph_chars/3` uses the first one that applies and succeeds.

Chapter 5

Logical Variable Macro Compilation

As discussed in the User's Guide, TRALE's logical-variable macros (unlike ALE macros) assume structure-sharing of their variables if they are used more than once in the definition of the macro. What TRALE does during compilation of logical-variable macros is simply translate into ALE macros such that structure-sharing between variables is preserved. For example, the following logical-variable macro

```
foo(X,Y) := f:X, g:X, h:Y.
```

is translated into

```
foo(X,Y) macro f:(X,_A), g:(X,_A), h:(Y,_B).
```

The added variables `_A` and `_B` are there to enforce structure sharing among multiple occurrences of the logical-variable macros `X` and `Y`. If a guard declaration is given in the definition of the logical-variable macro to restrict the type of the variables, then that is translated as a description as in the following example:

```
foo(X-a,Y-b) := f:X, g:X, h:Y

foo(X,Y) macro f:(X,a,_A), g:(X,a,_A), h:(Y,b,_B)
```

To reiterate, the added variables `_A` and `_B` make ALE treat the variables in the right-hand side not as logical variables but as normal Prolog variables.

Let us now see how TRALE performs this translation. First, TRALE gathers all `:=/2` predicates (logical-variable macros) in a list and hands them

to `compute_macros/2` which in turn goes through each macro one by one generating left-hand side arguments for a corresponding `macro/2` definition with the same predicate name and arity as the left-hand side of the original `:=/2`. The descriptions used as arguments in the new left-hand side, however, are provided by `add_lv_macro_descs/4`.

Chapter 6

Co-routining

SICStus Prolog follows a more complicated rule for computation than traditional implementations of Prolog. Instead of simply evaluating the leftmost rule in its search path, it evaluates the leftmost *unblocked* rule in its search path. This mechanism resembles co-routines in imperative programming languages, i.e., suspending and resuming different threads of control. SICStus Prolog provides some built-in co-routining predicates some of which, that are relevant to our discussion here, are as follows:

- `when(Condition,Goal)`: Block `Goal` until `Condition` is true, where `Condition` is a Prolog goal with the restricted syntax:

  ```
  nonvar(X)
  ?=(X,Y)
  Condition,Condition
  Condition;Condition
  ```

SICStus `when/2` suspends a goal until (i) a variable is instantiated or (ii) two variables are equated or inequated. The latter case corresponds directly to structure-sharing and inequations in typed feature structures. The former, if it were to be carried over directly, would correspond to delaying until a variable was promoted to a type more specific than $\bot$. There are degrees of instantiation in the logic of typed feature structures, however, corresponding to subsumption chains that terminate in a most specific type. With signature in which this chain is of length more than two, a more general and useful primitive is then suspending until a variable is promoted to a particular type. As discussed in the User's Guide, ALE has its own version of `when/2` which is designed for work with typed-feature structures. Table 6.1 shows how various `when/2` calls in ALE are

compiled. Conjunction and disjunction of conditionals are compiled as conjunctions and disjunctions respectively.

Input	**Output**
`when((FS=F:Desc), Goal)`	`when_type(FIntro,FS,DescGoal)` where `FIntro` is the introducer of `F`; and `DescGoal` is the result of compiling `Goal`.
`when(FS=(Path1==Path2), Goal)`	Treated as `when(Y^(FS=(Path1:Y, Path2:Y)), Goal)`.
`when(FS=Var,Goal)`	`when_eq(FS,Var,Goal)` if `Var` not seen; otherwise, unify `FS` and `Var` and call `Goal`.
`when(FS=Type,Goal)`	If `Type` is $\perp$, then call `Goal`; otherwise, `when_type(Type,FS,Goal)`.

Table 6.1: Compilation of `when/2`

`when_type(Type,FS,Goal)` is used to delay `Goal` until variable `FS` reaches type `Type`. To implement `when_type(Type,FS,Goal)`, ALE actually uses the underlying Prolog implementation of `when/2`. Types with no appropriate features constitute the base case, in which only delaying on the internal structure of the type's representation (which can be as simple as the type's name or a compound term encoding) is necessary. For types with appropriate features, we must ensure that our suspended `Goal` does not fire in between the time when the internal representation of a feature structure's type is updated, and when the appropriateness conditions for that type are enforced on the representation. To do otherwise would risk passing a non-well-typed or even non-well-formed representation in `FS` to the delayed `Goal`. What `when_type/3` must do is thus delay on the entire structure of the (well-typed) most general satisfier of its `Type` argument, relative just to appropriateness and the type system, i.e., without considering any implicational constraints. That most general satisfier is guaranteed not to have any structure-sharing since appropriateness is not able to require this. As a result, only recursively delaying on the value restrictions of its appropriate features is necessary.

More specifically, `when_type/3` delays the execution of `Goal` until such time that `FS` is not a variable and `Type` subsumes the type of `FS`. If it does not subsume the type of `FS`, then the execution of `Goal` is suspended until `Type` is an appropriate value restriction for the type of `FS`. If `Type` and the type of `FS` are not unifiable, then the `when_type/3` declaration is abon-

doned.

Another co-routining predicate used in ALE is `when_eq/3`. Basically, `when_eq(FS1,FS2,Goal)` suspends `Goal` until `FS1 == FS2`. This is implemented using Prolog `when/2` together with `?=/2`.

A similar co-routining predicate, `when_a_(X,FS,Goal)`, delays the execution of `Goal` until the feature structure `FS` becomes the argument of the `a_` atom used in `X`. `when_a_chk/3` is a very similar predicate, but rather than using `==/2` as in `when_a_/3`, it uses `subsumes_chk/2` to see whether `X` subsumes the argument of `a_`. This is necessary for checking appropriateness conditions with `a_/1` value restrictions, in which token-identity of variables has no significance.

Delaying on appropriateness restrictions is performed by `when_approp(Type,SVs,Goal)` which, as stated above, delays `Goal` until, `Type` becomes an appropriate value restriction for `SVs`.

Chapter 7

Complex-Antecedent Constraint Compilation

Unlike ALE that uses only type constraints (i.e., only types are allowed as the antecedent of constraints), TRALE employs *complex antecedent constraints*. These constraints allows arbitrary function-free and inequation-free antecedents. For a detailed discussion of these constraints, refer to the User's Guide.

Compilation of complex antecedent constraints is done by `compile_complex_assert/0` in three steps as follows:

1. Separate all existential quantifiers from the antecedent. Recall that TRALE interprets the scope of existentially quantified variables in complex antecedent constraints loosely. This step is necessary to identify such variables.
2. Find a *trigger* for the antecedent (defined below).
3. Replace the constraint with one that has the trigger found in the previous step as the antecedent.

The predicate that is responsible for separating existential quantifiers from the antecedent of a constraint is `find_narrow_vars/3`. For example, consider the following call to `find_narrow_vars/3`.

```
find_narrow_vars(A^B^C(A;(B,C)),Antec,Vars)
```

This will unify `Antec` with `A;(B,C)`, and `Vars` with `[C,B,A]`.

The next step is to find a trigger for the complex antecedent. The trigger is the most specific type whose most general satisfier subsumes the most general satisfier of the antecedent. The predicate `find_trigger_type/3` is responsible for finding the trigger. There are five possibilities, as follows:

- $trigger(\tau) = \tau$ where $\tau \in Type$
- $trigger(x) = \bot$ where x is a variable
- $trigger(F) = intro(F)$ where $F \in Feat$ and $intro(F)$ is the introducer of F
- $trigger(\phi_1 \wedge \phi_2) = trigger(\phi_1) \sqcup trigger(\phi_2)$
- $trigger(\phi_1 \vee \phi_2) = trigger(\phi_1) \sqcap trigger(\phi_2)$

The final step is to assert a kind of type-antecedent constraint for the trigger. The difference between this and a regular type-antecedent constraint is that in the former, the constraint only applies to a subset of objects of the specified type: those that match the description given in the original complex-antecedent.

Let us suppose there is a predicate, `whenfs(X=Desc,Goal)`, which delays `Goal` until `X` was subsumed by the most general satisfier of description `Desc`. Thus all principles $\alpha \Longrightarrow (\gamma \wedge \rho)$ could be converted into:

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

This typically involves compiling unifications of feature structures into unifications of their term encodings plus a type check to see whether one or more constraint consequents need to be enforced.

For example, if we formulate the Finiteness Marking Principle (FMP) for German as the following complex-antecedent constraint (assuming the type signature depicted in Figure 7.1):

```
synsem:loc:cat:(head:verb,marking:fin)
     *> synsem:loc:cat:head:vform:bse
```

then we can observe that the antecedent is a feature value description (F:ϕ), so the trigger is *intro*(SYNSEM), the unique introducer of the SYNSEM feature, which happens to be the type *sign*. We can then transform this constraint, as follows:

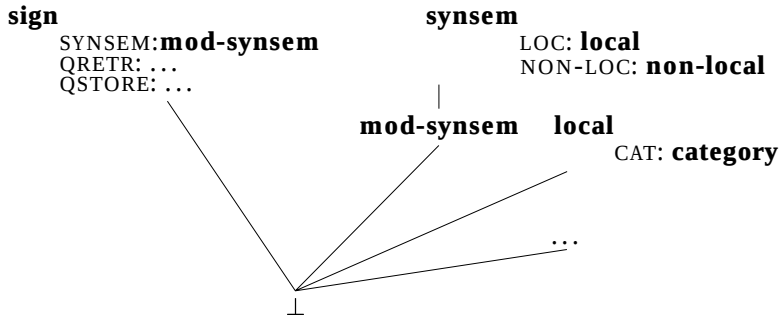

Figure 7.1: Part of the signature underlying the FMP constraint

```
sign cons X goal whenfs((X=synsem:loc:cat:(head:verb,
                                           marking:fin)),
                        (X=synsem:loc:cat:head:vform:bse)).
```

The effect of the hypothetical `whenfs/3` predicate is achieved by ALE's coroutining predicate, `when_type/3` (see Chapter 6), and `farg/3`. Given `when_type(Type,FS,Goal)`, ALE delays `Goal` until `FS` is of type `Type`. `farg(F,X,FVal)` binds `FVal` to the argument position of term encoding `X` that corresponds to the feature `F` once `X` has been instantiated to a type for which `F` is appropriate.

The corresponding type-antecedent constraint generated for our example is thus as follows:

```
sign cons X
     goal (farg(synsem,X,SynVal),farg(loc,SynVal,LocVal),
           farg(cat,LocVal,CatVal),farg(head,CatVal,HdVal),
           when_type(verb,HdVal,(farg(marking,CatVal,MkVal),
           when_type(fin,MkVal,
                     (X=synsem:loc:cat:head:vform:bse))))).
```

The above constraint now waits until the value of the HEAD feature of SYNSEM|LOC|CAT in a sign is `verb` and the value of its MARKING feature is `fin` before it enforces the constraint, which is unifying `X` with `synsem:loc:cat:head:vform:bse`.

Chapter 8

TRALE Lexical Rule Compiler

Vanessa Metcalf, Detmar Meurers and Markus Dickinson, Ohio State University [1]

T8.1 Introduction

T8.1.1 Background

In the framework of Head-Driven Phrase Structure Grammar (HPSG, Pollard and Sag, 1994) and other current linguistic architectures, linguistic generalizations are often conceptualized at the lexical level. Among the mechanisms for expressing such generalizations, so-called *lexical rules* are used for expressing so-called horizontal generalizations (cf. Meurers, 2001, and references cited therein).

Generally speaking, horizontal generalizations relate one word-class (e.g., passive verbs) to a second one (e.g., their active counterpart). A lexical rule specifies that for every word matching the description on the left-hand side of the rule (the input) there must also be a word matching the description on the right-hand side of the rule (the output). Lexical rules in HPSG are generally interpreted based on the understanding that properties not specified in the output description are assumed to be identical to the input of a lexical rule, which is sometimes referred to as framing. Lexical rules can be thought of in a static sense, as capturing generalizations about the lexicon, or in a dynamic sense, as licensing derived lexical entries.

The lexical rule compiler implemented for the TRALE system encodes the treatment of lexical rules proposed in Meurers and Minnen (1997). The underlying idea of the approach is to make computational use of the generalizations encoded in lexical rules, instead of expanding out the lexicon at compile time, as in the common approach also available in the TRALE system. The lexical rule compiler of Meurers and Minnen (1997) takes a basic lexicon and a set of lexical rules to produce a set of lexical entries which is modified to license not only the base lexical entries, but also all possible derivations. The lexical rule compiler approach thus has two advantages over an approach that complies out the lexicon: the resulting lexicon is much smaller, and it allows for lexical rules that apply recursively, and thus can result in infinitely large lexicons.

T8.1.2 The lexical rule compiler

The lexical rule compiler consists of the four steps shown in figure 8.1. A full discussion of this setup is provided in Meurers and Minnen (1997); we provide a brief summary of the overall approach here, to be followed by a discussion of our implementation of each step in the following sections.

In the first step we translate user-specified lexical rules to an internal Prolog representation. We generate the so-called frame predicates for each lexical rule, which transfer all information from the input to the output which is not specified in the output description. The internal Prolog representations are specified to call the frame predicates. In the second step, we develop a global interaction finite state automaton (global finite-state automaton), based on a follows relation, which is itself obtained from testing which lexical rule inputs can unify with which lexical rule outputs. We then perform the step of actually unifying the feature structures along all paths, propagating the specifications, which may allow the pruning of some arcs already during compile time instead of at runtime. This global finite-state automaton encodes all possible sequences of lexical rule application from a base lexical entry, on the basis of information in the lexical rules alone. In the third step, we take each lexical entry and run it through the global finite-state automaton, applying the lexical rules to the base or derived lexical entry (as the case may be) pruning those arcs where unification fails. In this way we end up with a finite-state automaton for each lexical entry. In the fourth step, we produce interaction predicates based on the automata and modify the lexical entries to call the appropriate interaction predicate, effectively grouping the lexical entries into lexical classes. That is, since each finite-state automaton is

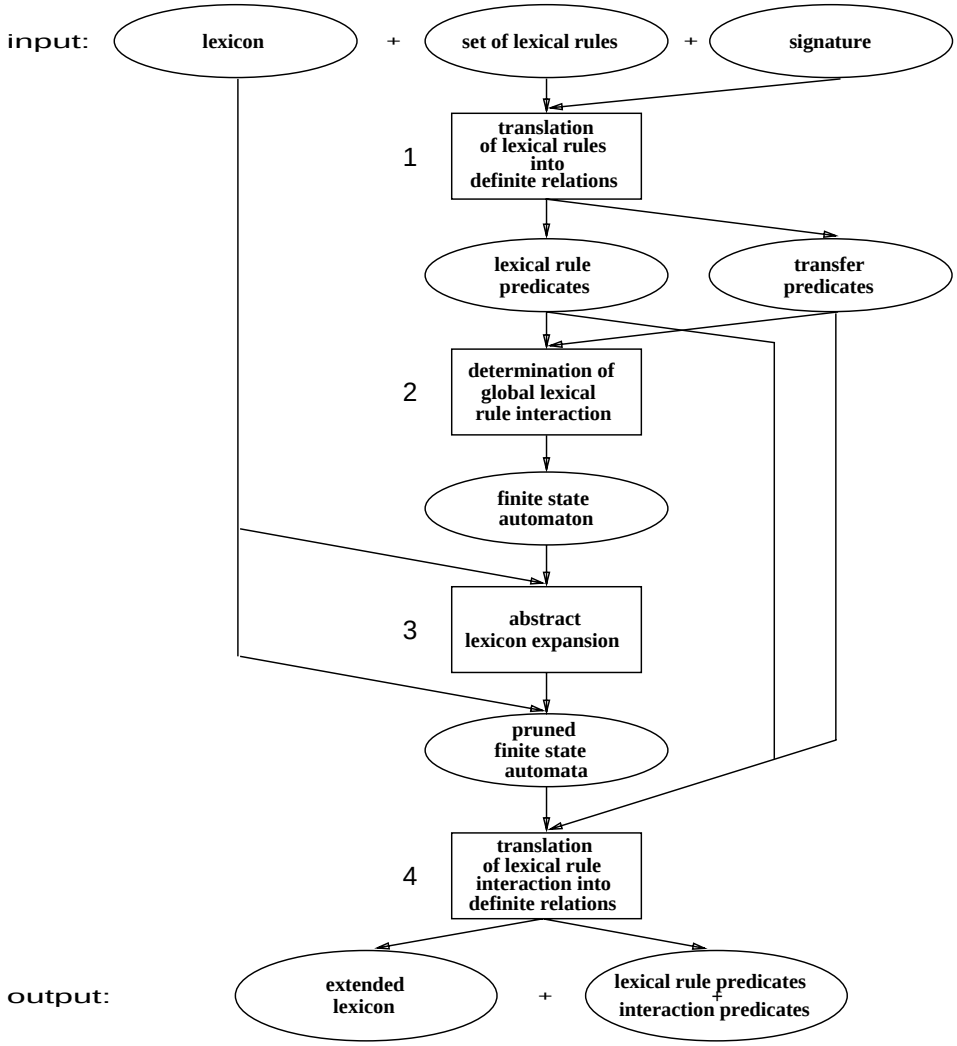

Figure 8.1: Overview of the lexical rule complier

simply a pruned version of the original global finite-state automaton, we can easily check to see if an identical finite-state automaton has already been translated to an interaction predicate. If so, we forgo translating the current finite-state automaton, calling the previously asserted interaction predicate in the current lexical entry instead.

The lexical entries that ultimately result from this process are encodings of base lexical entries which also call the finite state automaton for their word class. This finite-state automaton encodes each possible sequence of lexical rule application as a path through the automaton, where the traversal of an arc results in the application of some lexical rule, the arcs leaving a particular state represent all possible lexical rule applications at that point, and every state is a final state. The start state corresponds to the base lexical entry, and every other state corresponds to some derived lexical entry, which is the output of some lexical rule, and may (possibly) serve as input to others. Thus, the lexical entry, as a whole, licenses any word object whose feature specifications unify with those at some point along a path in the finite-state automaton.

While we do provide general descriptions of the current state of implementation, the primary focus of this reference manual is lexical rule interpretation and the generation of frame predicates.

Before we begin discussion of frame generation, we define the terms *feature description* and *feature structure*. By *feature description* we refer to compound terms in ordinary TRALE syntax. Such terms are specified by the user, and may be partial representations with respect to type values and appropriate features. By *feature structure*, we refer to TRALE's internal representation, in which all type values and appropriate feature values are represented. A variety of built-in TRALEpredicates can be called to obtain information about, or add information to, a feature description.

T8.2 Step One: Generating Frames

In this first step of the compiler, we assert lexical rule predicates corresponding to each user-specified lexical rule, in addition to frame predicates for each lexical rule. Generally speaking, frames specify which feature values in the input are equated with the corresponding value in the output, and, when called, the frame predicates achieve the transfer of this information from one lexical entry (the input) to a derived lexical entry (the output).

In terms of input to the actual process of frame generation, we are

working with two feature descriptions (lexical rule input and output), which may or may not be of the same type. When a feature appropriate to the type of both lexical rule input and output is not mentioned in the output description, we ensure transfer of the input specification to the output via a path equality in the frame predicate. The process of frame generation is recursive, since the (possible) need for path equalities to transfer information applies not only at the top level, but along all paths and subpaths mentioned in the output description. At any level, that is, even when we are working with feature structures or descriptions embedded in our original input, we refer to the current input to the process as lexical rule input and output.

The frame predicates have three arguments: the first identifies the associated lexical rule, while the second and third (referred to as IN and OUT, respectively) are feature structures of type *word* (or some maximally specific subtype of *word*). Path equality in TRALE is realized by means of variables. For feature values that demand a path equality (as described above), we assign the same variable as the value of that feature in both IN and OUT. By unifying the input of the lexical rule with IN and the output of the lexical rule with OUT, "transfer" is achieved.

Two aspects of the task of frame generation here make it relatively complicated: disjunction in the descriptions, and types. We discuss issues related to disjunction in section T8.2.1, and the interpretation of types in the input and output of a lexical rule with respect to framing in section T8.2.2.

T8.2.1 Disjunction in the input and output specifications

While our treatment of disjunction in the lexical rule specifications is consistent with ordinary interpretation of disjunction in TRALE, there are several aspects of that interpretation worth discussing in this particular context.

First, it should be noted that disjunction in a description is a compact representation of two distinct feature descriptions, and so disjunction in the input or output descriptions of a lexical rule is actually a compact representation of what could be two distinct lexical rules. Second, TRALE's syntax does not include named disjunction. For example, an input description *a*;*b* with an output description *c*;*d* does not restrict the possible mappings of input to output to *a* to *c* and *b* to *d*. Instead, such a specification allows fours possible mappings of input to output: *a* to *c*, *a* to *d*, *b* to *c*, and *b* to *d*. A disjunction in the input or output of a lexical rule

means that we will need to calculate a frame corresponding to each disjunct. Each disjunction in either input or output multiplies the number of frames needed by two.

Further, TRALE's internal feature structure representation does not include disjunction. If asked to translate the the user-specified description (*a*;*b*) to a feature structure, Prolog will return *a*. The feature structure *b* is obtained by asking for more solutions. The disjunctive possibilities with regard to a feature description can be generated by means of code which utilizes a failure-driven loop calling TRALE's translation predicate and then collects the results, or by means of code which step-wise multiplies out the disjunctions, yielding some number of disjunctionless feature descriptions.

In our implementation of the algorithm, we deal with disjunction in the input and output of a lexical rule differently, based on the kind of representation we want to work with during the compilation process. Because we only make use of the internal representation of the input during processing, and not the original user-specified description, we use a failure-driven loop to generate all possible input feature structures. Because we want to know what the user has specified in the output, which is not possible to know from the internal representation of that description, we work with feature description representations of the output during processing. We have implemented code which takes a feature description possibly containing disjunction as input, and returns a list of disjunctionless descriptions. We then then generate a list of frames for each input-output pair.

T8.2.2 Interpreting lexical rule descriptions

A lexical rule can apply to a variety of lexical entities. While each of these lexical entities must be described by the input of the lexical rule in order for the rule to apply, other properties not specified by the lexical rule can and will vary between lexical entries. Feature structures corresponding to lexical entities undergoing the lexical rule therefore may differ in terms of type value and appropriate features. Frames carrying over properties not changed by the lexical rule need to take into account different feature geometries. Since framing utilizes structure sharing between input and output, we only need to be concerned with the different kinds of objects that can undergo a lexical rule with regard to the paths and subpaths mentioned in the output description. Specifically, when the objects undergoing lexical rule application differ with regard to type value along

some path mentioned in the output description, we may need to take into account additional appropriate attributes in framing. Each such possibility will demand its own frame.

We provide a truthful procedural realization of the formal interpretation of lexical rules defined in Meurers (2001). Generally speaking, the input description of a lexical rule specifies enough information to capture the class of lexical entries intended by the user to serve as inputs. The output description, on the other hand, specifies what should change in the derivation. All other specifications of the input are supposed to stay the same in the output.

In the spirit of preserving as much information as possible from input to output, we generate frames on the basis of species (=most specific type) pairs; that is, we generate a frame (an IN-OUT pair) on the basis of a maximally specific input type, and a maximally specific output type, subtypes of those specified in, or inferred from, the lexical rule description. In this way we maintain tight control over which derivations we license, and we guarantee that all possible information is transferred, since the appropriate feature list we use is that of a maximally specific type. We create a pair of skeleton feature structures for the species pair, and it is to this pair of feature structures that we add path equalities. We determine the appropriate list of species pairs on the basis of the types of the input and output descriptions.

The first step in this process is determining the types of the input and output of the lexical rule. We then obtain the list of species of the input type, and the list of species of the output type. We refer to these as the input species list, and the output species list, and their members as input and output species. At this point it will be helpful to have an example to work with. Consider the type hierarchy in figure 8.2.

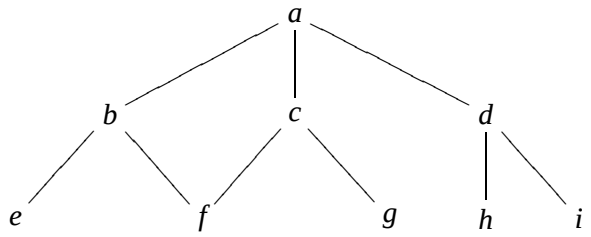

Figure 8.2: An example hierarchy illustrating the interpretation

We can couch the relationship between the input and output types in terms of type unification, or in terms of species set relations. In terms of

unification, there are four possibilities: the result of unification may be the input type, the output type, something else, or unification may fail. In the first case the input type is at least as or more specific, and the input species will be a subset of the output species. In the second case the output is more specific and the output species will be a subset of the input species. In the third case the input and output types have a common subtype, and the intersection of input and output species is nonempty. In the fourth case the input and output types are incompatible, and the intersection of their species sets is empty.

Given in figure 8.3 are examples of all four cases, using the example signature, showing input and output types, the result of unification, their species lists, and the species-pairs licensed by the algorithm described below. Calling these separate "cases" is misleading, however, since the algorithm is the same in every case.

	Case 1	Case 2	Case 3	Case 4
Input type	*c*	*a*	*b*	*c*
Output type	*a*	*c*	*c*	*d*
Unify to	*c*	*c*	*f*	fail
Input species	*f, g*	*e, f, g, h, i*	*e, f*	*f, g*
Output species	*e, f, g ,h, i*	*f, g,*	*f, g*	*h, i*
Species pairs	*f-f, g-g*	*e-f, e-g, f-f, g-g, h-f, h-g, i-f, i-g*	*e-f, e-g, f-f*	*f-h, f-i, g-h, g-i*

Figure 8.3: Examples for the four cases of mappings

If a (maximally specific) type value can be maintained in the output, it is. Otherwise, we map that input species to all output species. In terms of set membership, given a set of input species X, a set of output species Y, the set of species pairs P thus can be defined as:

$$P = \{\langle x, x\rangle \mid x \in X \wedge x \in Y\} \cup \{\langle x, y\rangle \mid x \in X, x \notin Y \wedge y \in Y\}$$

This definition translates to a simple algorithm in the code itself:

- For an input species *x*, if *x* is a member of the output species list, then calculate a frame for *x*-*x*.
- Otherwise, calculate a frame for every *x*-*y* pair, where *y* is an output species.

T8.2.3 Overview of code for frame generation

Taking the input and output descriptions for a particular lexical rule, we first generate a list of possible feature structures corresponding to the input description. (As discussed in section T8.2.1 there will be multiple feature structures if there is disjunction in the description.) We then derive from the output description a list of disjunctionless descriptions, and obtain the internal representation of each.[2] For each input-output pairing (where "output" is itself a description-feature structure pair), we obtain the list of species pairs for which we want to generate a frame predicate. In a species pair, we refer to the input species as IN and the output species as OUT.

There may in fact be more than one frame necessary per species pair because of embedded feature structures which themselves require multiple "small" frames. For each of these, a copy of the larger frame (an IN-OUT pair) is created, and the small IN and OUT are assigned as feature values in the larger frame. Therefore, we obtain a list of frames for each species pair.

Given a particular species pair, we create skeleton feature structures for the IN and OUT types that constitute the pair, and obtain a list of appropriate features for the OUT type. We then walk through the output description, removing mentioned features from the appropriate features list. Concomitantly, when we find a feature with a non-atomic value during the walk-through, we call the process of frame generation for the description which is the value of that feature in the output, generating a list of "small" frames, which are incorporated into the larger one (as mentioned in the previous paragraph). This is accomplished by obtaining the embedded feature structures from the frames and unifying the small frames with these. This process of walk-through, removing mentioned features from an appropriate-feature list and incorporating small frames into large ones, applies recursively along every path and subpath in the output description.

Once we have walked through the entire output description at a particular level of embedding, path equalities are added for features which are appropriate for the input type and have not been removed from the

[2] We actually translate a copy of each description into a feature structure, since desc2fs/3 adds type information to Prolog variables, thereby making them unrecognizable as such to var/1. We rely on Prolog variables in the descriptions to signal the presence of a path equality in the lexical rule descriptions. We keep track of this information in order to avoid creating undesirable path equalities in the frames.

appropriate features list of the output.

T8.3 Step Two: Global Lexical Rule Interaction

In this step, we construct a global interaction finite-state automaton (global finite-state automaton) encoding all possible sequences of lexical rule application, taking into account only the information specified in the lexical rules themselves. This step of the algorithm has three components. We first calculate and assert the follows relation. The follows relation associates each lexical rule with the list of lexical rules that can apply to the output of the first. We calculate the follows relation by test-unifying the output specifications of one rule with the input specifications of all lexical rules, and associating the first lexical rule id with a list of ids where unification succeeded.[3]

Based on the follows relation, we build a finite-state automaton with lexical rule identifiers labelling the arcs. Arcs corresponding to each lexical rule originate at the start state, and for any state, the lexical rule labelling the arc terminating at that state determines the arcs that will originate there, according to the follows relation. In the case that a lexical rule may apply to its own output, we include an arc whose start state and end state are the same, and flag the existence of a cycle in the representation of the arc. Every state is ultimately intended to be a final state, though this is not encoded in the representation at this stage of the compilation process.

The final component of this step involves the propogation of specifications. That is, we prune unusable arcs based on the actual unification of output and input specifications along each of the paths through the automaton.

We use arc(Arc_Label,Rule_Label,Prefix_List,Follow_List,Cycle_Flag) to build the automaton because there exists a convenient relationship between this representation and the follows relation definitions. Specifically, given the Rule_Label argument for a particular arc, we can find the follows/2 clause which has that Rule_Label as its first argument. We can then directly incorporate the follows list from the follows/2 clause

[3]To obtain the correct follows relation for a lexical rule, however, information from the input specification should to be transferred to the output, ideally via a generalized frame predicate. Currently, we include a lexical rule on the follows list if unification succeeds utilizing at least one of the clauses defining the frame predicate for the lexical rule in question.

as the fourth argument in arc/5, and then recursively work through that list to assert arc/5 clauses for each member. However, for the stage of processing that involves actual unification of feature structures along all paths, we associate feature structures with particular states, and store this information in a table. At this point, it becomes more convenient to work with a state-based representation of the automaton. Therefore, we translate from the arc representation given above to arc(StateIn_Label,StateOut_Label,Rule_Label,Cycle_Flag). The automaton that serves as input to step three is defined via arc/4.

T8.4 Step Three: Word Class Specialization

In this step, we associate every lexical entry with a unique identifier, and taking the global finite-state automaton which is the output of step two as input, create a local finite-state automaton for each lexical entry. This step is similar in execution to the second component of step two, since we unify the lexical feature structure with the lexical rule specifications along all paths in the automaton, prune arcs which cannot apply to that lexical entry and its derivations, and create a table in which states are associated with feature structures. We alter the automaton representation to incorporate the lexical identifier, resulting in output of the form lex_arc(Lex_ID,StateIn_Label,StateOut_Label,Rule_Label,Cycle_Flag).

T8.5 Step Four: Definite Relations

In the final step of the lexical rule compilation process, we convert the output of step three into the Prolog definite relations referred to as interaction predicates. We then alter the lexical entries to call the interaction predicates, which in turn call the lexical rule predicates. In order to avoid storing redundant interaction predicates, we collect lexical entries into word classes. That is, we determine which lexical entries are associated with the same local finite-state automaton, and alter those lexical entries to call a single interaction predicate. To do this, we translate the local finite-state automata into compound terms. This allows us to find identical finite-state automata via simple term unification.

When we assert an interaction predicate, we also assert its canonical representation. This gives us an inventory of finite-state automata represented as compound terms, which we can check any finite-state automa-

ton against. If the local finite-state automaton associated with a particular lexical entry has already been translated to an interaction predicate (and asserted as such), we do not translate the finite-state automaton in question to an interaction predicate, retracting it instead. If, on the other hand, none of the previously asserted compound terms unify with the canonical representation of the finite-state automaton in question, we assert the compound term. Additionally, we translate the arcs of that local finite-state automaton to clauses defining an interaction predicate, asserting each clause in turn. In either case, we assert a new representation of the current lexical entry, modified to call the preexisting or newly translated interaction predicate, as the case may be.

Chapter 9

Bottom-up Parsing

ALE's parser makes use of the EFD-closure parsing algorithm introduced in Penn and Munteanu (2003). Unlike the textbook chart parsing algorithm and Carpenter's algorithm employed in earlier versions of ALE (Carpenter and Penn, 1996), this algorithm reduces the number of copying operations for edges to a constant number of two per non-empty edge. Penn and Munteanu (2003) show that this algorithm performs better than standard bottom-up Prolog parsers on a variety of grammar shapes.

9.1 The Algorithm

EFD stands for *Empty First Daughter*. The EFD-closure algorithm assumes that a grammar is "EFD-closed" meaning that the first daughter of all the grammatical rules in the grammar are non-empty. The definition of an EFD-closed grammar is provided below:

Definition 18 *An (extended) context-free grammar, G, is* empty-first-daughter-closed *(*EFD-closed*) iff, for every derivation $N \Rightarrow^* w$, there is a derivation of $N \Rightarrow^* w$ in which no left-corner of any non-preterminal subtree is ϵ.*

At compile time, ALE transforms any phrase-structure grammar to an EFD-closed grammar. The EFD-closure process is explained in section 9.2 below. In this section, we step through the parsing of a sentence based on the simple grammar presented below. For ease of exposition, atomic categories are used instead of feature structures.

```
s    ===> [np,vp].
```

```
vp   ===> [vbar].
vp   ===> [vbar,pp].
vbar ===> [vt,np].
np   ===> [det,nbar].
nbar ===> [n].
nbar ===> [n,pp].
pp   ===> [p,np].

lex(john,np).
lex(nudged,vt).
lex(a,det).
lex(the,det).
lex(man,n).
lex(cane,n).
lex(with,p).
```

The sentence that we are parsing is "John nudged the man with a cane." The top-level predicate of the parser is `rec/2` which takes a list of words in its first argument and returns a feature structure corresponding to that sentence in its second argument. Therefore, to parse the above sentence ALE calls `rec/2` as follows:

```
rec([john,nudged,the,man,with,a,cane],FS)
```

This algorithm proceeds breadth-first, right-to-left through the input string at each step applying the grammar rules depth-first, matching daughter categories left-to-right. The first step is then to reverse the input string, and compute its length (performed by `reverse_count/5`) and initialize the chart:[1]

```
rec(Ws,FS):-
  retractall(edge(_,_,_)),
  reverse_count(Ws,[],WsRev,0,Length),
  CLength is Length - 1,
  functor(Chart,chart,CLength),
  build(WsRev,Length,Chart),
  edge(0,Length,FS).
```

`retractall(edge(_,_,_))` resets the chart by removing all asserted edges. `reverse_count(Ws,[],WsRev,0,Length)` reverses the input list and computes its length. In the case of our example, it will return this:

[1] Note that for ease of exposition, we are showing a simplified version of the code here.

```
reverse_count([john,nudged,the,man,with,a,cane],[],
              [cane,a,with,man,the,nudged,john],0,7)
```

Two copies of the chart are used in this presentation. One is represented by a term `chart(E1,...,EL)`, where the i^{th} argument holds the list of edges whose left node is i. Edges at the beginning of the chart (left node 0) do not need to be stored in this copy, nor do edges beginning at the end of the chart (especially, empty categories with left node and right node `Length`). This is called the *term copy* of the chart. The other copy is kept in a dynamic predicate, `edge/3`, as a textbook Prolog chart parser would. This is called the *asserted copy* of the chart.

Neither copy of the chart stores empty categories. These are assumed to be available in a separate predicate, `empty_cat/1`. Since the grammar is EFD-closed, no grammar rule can produce a new empty category. As you may have noticed already, lexical items are assumed to be available in the predicate `lex/2`.

In our example, `CLength` evaluates to 6, which means the corresponding term copy of the chart gets initiated as `chart(_,_,_,_,_,_)`. The predicate, `build/3`, then actually builds the chart. The definition of `build/3` is shown below:

```
build([W|Ws],R,Chart):-
  RMinus1 is R-1,
  (lex(W,FS),
   add_edge(RMinus1,R,FS,Chart)
  ; ( RMinus1 =:= 0 -> true
    ; rebuild_edges(RMinus1,Edges),
      arg(RMinus1,Chart,Edges),
      build(Ws,RMinus1,Chart)
    )
  ).

build([],_,_).
```

The precondition upon each call to `build(Ws,R,Chart)` is that `Chart` contains the complete term copy of the non-loop edges of the parsing chart from node `R` to the end, while `Ws` contains the (reversed) input string from node `R` to the beginning. Each pass through the first clause of `build/3` then decrements `Right`, and seeds the chart with every category for the lexical item that spans from `R-1` to `R`. The predicate `add_edge/4` actually adds the lexical edge to the asserted copy of the chart, and then

closes the chart depth-first under rule applications in a failure-driven loop. `add_edge/4` initiates a loop because it calls `rule/4`, which in turn calls `match_rest/5`, which then calls `add_edge/4` again. It is also a failure-driven loop because the number of rules with any given leftmost daughter is finite causing `rule/4` to fail after all the relevant rules have been processed. When it has finished, if `Ws` is not empty (`RMinus1` is not 0), then `build/3` retracts all of the new edges from the asserted copy of the chart (with `rebuild_edges/2`, described below) and adds them to the `R`-1st argument of the term copy before continuing to the next word.

`add_edge/4` matches non-leftmost daughter descriptions from either the term copy of the chart, thus eliminating the need for additional copying of non-empty edges, or from `empty_cat/1`. Whenever it *adds* an edge, however, it adds it to the asserted copy of the chart. This is necessary because `add_edge/4` works in a failure-driven loop, and any edges added to the term copy of the chart would be removed during backtracking:

```
add_edge(Left,Right,FS,Chart):-
  assert(edge(Left,Right,FS)),
  rule(FS,Left,Right,Chart).

rule(FS,L,R,Chart):-
  (Mother ===> [FS|DtrsRest]), % PS rule
  match_rest(DtrsRest,R,Chart,Mother,L).

match_rest([],R,Chart,Mother,L):-
  add_edge(L,R,Mother,Chart).

match_rest([Dtr|DtrsRest],R,Chart,Mother,L):-
  arg(R,Chart,Edges),
  member(edge(NewR,Dtr),Edges),
  match_rest(DtrsRest,NewR,Chart,Mother,L)
  ; empty_cat(Dtr),
    match_rest(DtrsRest,R,Chart,Mother,L).
```

Note that we never need to be concerned with updating the term copy of the chart during the operation of `add_edge/4` because EFD-closure guarantees that all non-leftmost daughters must have left nodes strictly greater than the `Left` passed as the first argument to `add_edge/4`.

Moving new edges from the asserted copy to the term copy is straightforwardly achieved by `rebuild_edges/2`:

```
rebuild_edges(Left,Edges):-
  retract(edge(Left,R,FS))
  -> Edges = [edge(R,FS)|EdgesRest],
        rebuild_edges(Left,EdgesRest)
  ;     Edges = [].
```

The two copies required by this algorithm are thus: 1) copying a new edge to the asserted copy of the chart by `add_edge/4`, and 2) copying new edges from the asserted copy of the chart to the term copy of the chart by `rebuild_edges/2`. The asserted copy is only being used to protect the term copy from being unwound by backtracking.

The first pass of `build/3` on our example sentence will thus update the term copy of the chart as follows:

```
chart(_,_,_,_,_,[edge(7,n),edge(7,nbar)])
```

This means that the parser has found an `n` and an `nbar` that span from the edge 6 (known from the argument number in the chart) to the edge 7 (recorded by `edge/2`). By the time the parser reaches the first word in the input string, `Chart` will contain the following data:

```
chart([edge(2,vt),edge(4,vbar),edge(4,vp),
       edge(7,vp),edge(7,vbar),edge(7,vp)],
      [edge(3,det),edge(4,np),edge(7,np)],
      [edge(4,n),edge(4,nbar),edge(7,nbar)],
      [edge(5,p),edge(7,pp)],
      [edge(6,det),edge(7,np)],
      [edge(7,n),edge(7,nbar)])
```

At the end of the parsing operation, the asserted copy of the chart only has a record of the edges whose left corner is 0. This is because all the other edges have been retracted by `rebuild_edges/4` while updating the term copy of the chart, and because there is no need to copy the asserted copy of the edges with a left corner 0 onto the term copy. `rec/2` uses this to retrieve the edge that spans the whole input string.

9.2 EFD-closure

To convert an (extended) context-free grammar to one in which EFD-closure holds, we must partially evaluate those rules for which empty categories could be the first daughter over the available empty categories. If

all daughters can be empty categories in some rule, then that rule may create new empty categories, over which rules must be partially evaluated again, and so on. The closure algorithm is presented in Figure 9.1 in pseudo-code and assumes the existence of six auxiliary lists:

- `Es`—a list of empty categories over which partial evaluation is to occur,
- `Rs`—a list of rules to be used in partial evaluation,
- `NEs`—new empty categories, created by partial evaluation (when all daughters have matched empty categories),
- `NRs`—new rules, created by partial evaluation (consisting of a rule to the leftmost daughter of which an empty category has applied, with only its non-leftmost daughters remaining),
- `EAs`—an accumulator of empty categories already partially evaluated once on `Rs`, and
- `RAs`—an accumulator of rules already used in partial evaluation once on `Es`.

Each pass through the while-loop attempts to match the empty categories in `Es` against the leftmost daughter description of every rule in `Rs`. If new empty categories are created in the process (because some rule in `Rs` is unary and its daughter matches), they are also attempted—`EAs` holds the others until they are done. Every time a rule's leftmost daughter matches an empty category, this effectively creates a new rule consisting only of the non-leftmost daughters of the old rule. In a unification-based setting, these non-leftmost daughters could also have some of their variables instantiated to information from the matching empty category.

If the while-loop terminates (i.e., if there is a finite number of empty categories derivable), then the rules of `Rs` are stored in an accumulator, `RAs` until the new rules, `NRs`, have had a chance to match their leftmost daughters against all of the empty categories that `Rs` has. Partial evaluation with `NRs` may create new empty categories that `Rs` have never seen and therefore must be applied to. This is taken care of within the while-loop when `RAs` are added back to `Rs` for the second and subsequent passes through the loop.

```
Initialise Es to empty cats of grammar;
initialise Rs to rules of input grammar;
initialise the other four lists to [];

loop:
while Es =\= [] do
  for each E in Es do
    for each R in Rs do
      unify E with the leftmost unmatched category
        description of R;
      if it does not match, continue;
      if the leftmost category was rightmost (unary
        rule),
      then add the new empty category to NEs
      otherwise, add the new rule (with leftmost
        category marked as matched) to NRs;
    od
  od;
  EAs := append(Es,EAs); Rs := append(Rs,RAs);
  RAs := [];  Es := NEs; NEs := [];
od;
if NRs := [],
then end: EAs are the closed empty cats,
          Rs are the closed rules
else
  Es := EAs; EAs := []; RAs := Rs;
  Rs := NRs; NRs := [];
  go to loop
```

Figure 9.1: The off-line EFD-closure algorithm

9.3 Parsing Complexity

All textbook bottom-up Prolog parsers copy edges out of the chart once for every attempt to match an edge to a daughter category. Carpenter's algorithm—which traverses the string breadth-first, right-to-left and matches rule daughters rule depth-first left-to-right in a failure-driven loop—can in the worst case make $\mathcal{O}(n^{b-1})$ copies, where b is the maximum rule branching factor. The worst-case time complexity of Carpenter's algorithm is $\mathcal{O}(n^{b+1})$. A fixed CFG based on atomic categories can be converted off-line to Chomsky Normal Form in which $b = 2$, thus resulting in the cubic-time recognition, $\mathcal{O}(n^3)$. The EFD-closure algorithm, as mentioned in the beginning of this chapter, reduces the number of copying to 2 per non-empty edge. Like Carpenter's algorithm, however, its worst-case time complexity is $\mathcal{O}(n^{b+1})$.

Chapter 10

Topological Parsing

This chapter presents a new formalism used in ALE for parsing with freer word-order languages. This new formalism, which is based on a CFG-like backbone, does not presuppose contiguity of constituents nor does it presuppose any linear order. Where necessary, however, these can be explicitly expressed in order to provide the grammar with sufficient constraints to allow for more efficient parsing.

This is achieved by departing from the traditional notion of constituency where a single region (namely conventional grammatical categories such as NP, VP, etc.) is used to define aspects of interpretation (e.g. thematic role assignment), linear order, and contiguity. In the topological-fields (TF) approach, two types of constituency are involved. One type consists of the regions used to capture aspects of interpretation only, and the other involves the regions that capture aspects of linear order and contiguity. This idea was first introduced by Curry (1961) but was not completely formalised. In Curry's terms, the regions that capture interpretation belong to the domain of *tectogrammar* and those responsible for linear order and contiguity belong to the domain of *phenogrammar*. Traditional grammatical categories stripped of their linear-order and contiguity assumptions correspond to tectogrammatical categories. As for phenogrammatical categories, we extend the notion of topological fields used by traditional German grammarians to define contiguous regions with a fixed order of constituents. The TF approach has three properties:

- **Topological Linearity:** All word order constraints are defined on some linear region.

- **Topological Locality:** All discontiguities are characterised in some local region.
- **Qualified Isomorphy:** Regions that captures linear order and those that capture contiguity are the same (i.e., topological fields).

10.1 Phenogrammar

The advantage of using topological fields, which express linear precedence (LP) constraints, is that they provide first-class access to units that may not correspond to syntactic subtrees or may occur in a similar position but not form any natural syntactic and/or semantic class. An example of the latter is the *Left Sentence-Bracket (CF)* in German. This position, which is the second traditionally identified topological field in the clause is filled by such categories as finite verbs, complementisers, and subordinating conjunctions. These do not form a natural syntactic class but occupy the same linear position. Moreover, without recourse to topological fields it is very difficult to define this second position. Another example is the first (topic) position, the *Vorfeld (VF)*, which holds a variety of categories and sometimes strings of words that are only parts of syntactic categories. An example of the traditional topological fields identified for German is given in 4 below:[1]

(4)

VF	CF	MF	VC	NF
Dem Mann	*habe*	*ich das Buch*	*gegeben*	*das ich gelesen habe.*
the man	have	I the book	given	that I read have

I gave the man the book that I've read.

These five fields are said to describe the topology of the *clause* region. In addition to the clause region, we assume that other regions can be found upon which one can define a topology. In the case of German for example, one can also talk about a *noun-phrase regions (NPR)* that contains its own fields. NPRs may or may not correspond to complete NPs in tectogrammar depending on whether or not they are contiguous. This way, we are extending the rather flat structure of regions and giving it a recursive structure. The need for this is made more apparent by certain facts: for example, a NF in a clause region itself being able to contain another (embedded) clause.

[1] VF stands for Vorfeld, CF for Complementiser Field (left sentence-bracket), MF for Mittelfeld, VC for Verbal Complex (right sentence-bracket), and NF for Nachfeld

An extended CFG is used to formalise this grammar. The infix operator `topo/2` is used to define the topology of regions. For example, a topological rule for the German clause region which is written as:

```
clause topo [vf,cf,mf*,{vc},{nf}]
```

means that a clause region contains exactly one VF followed by exactly one CF, followed by zero or more MFs, followed by an optional VC and an optional NF. A topological rule for the German NPR can be written as follows:

```
npr topo [{sprf},adjf*,nounf,postmodf*]
```

The above rule states that a German NPR is made up of an optional *specifier field (SPRF)*, followed by zero or more *adjective fields (ADJF)*, followed by exactly one *noun field (NounF)*, and finally zero more *post-modifier fields (PostModF)*.

10.1.1 Linkage

Since regions themselves can occur as fields in larger regions, we also need a way of making that explicit in the grammar. A new class of rules called *linking rules* is used for this purpose. These rules are universally quantified on the left-hand side and existentially quantified on the right-hand side. Therefore,

```
r <<-- f
```

states that all regions named `r` occur in some `f`, where `f` is a field name or a disjunction of field names. The rule

```
f -->> r
```

on the other hand, state that all fields named `f` contain some `r`, where `r` is a region name or a disjunction of region names. Two examples of linking rules in German are provided below:

```
npr <<-- (vf;mf;nf)
clause <<-- nf
```

These rules state that all NPRs in German occur in some VF, or MF, or NF, and all clauses in German occur in some NF. Looking back to our example sentence above, we can now see what phenogrammatical tree these rules together generate for that sentence. This is shown in Figure 10.1.

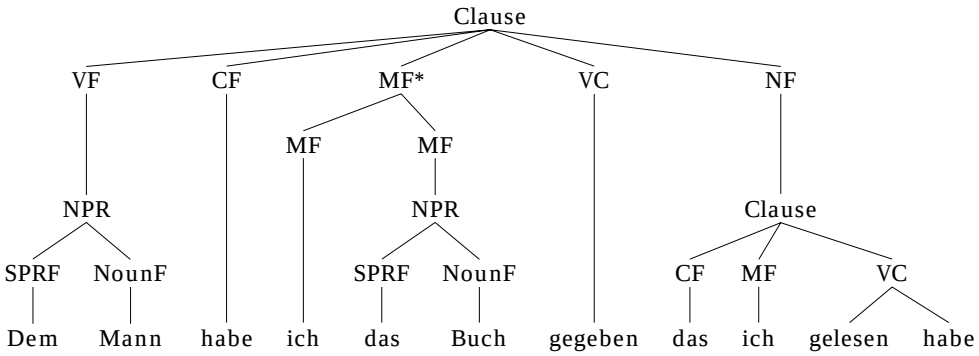

Figure 10.1: A sample phenogrammatical tree

10.2 Tectogrammar

As a result of delegating the tasks of enforcing linear precedence and contiguity constraints to phenogrammar, tectogrammar's only remaining task is to guide semantic interpretation. Thus, our tectogrammatical rules do not assume any order or contiguity in their daughters. However, the formalism allows for specifying such constraints explicitly if need be. Tectogrammatical rules are represented using `*-->/2` infix operator. The category on the left-hand side of a tectogrammatical rule represents the mother node, and the daughters are provided (in no particular order) on the right-hand side. The daughters should be separated by commas. For example, all the following rule is specifying is that a sentence contains a NP and a VP. Nothing is said about the order of these constituents or whether they are contiguous at all.

```
s *--> np, vp
```

One may provide some conditions for the applicability of a tecto-rule much in the same manner as DCGs. The acceptable conditions are local covering, local matching, precedence, immediate precedence, local compaction and inequations or other Prolog goals. The following section discusses various structural constraints on grammar rules.

10.3 Synchronising Phenogrammar and Tectogrammar

Synchronisation between phenogrammar and tectogrammar is achieved through the notion of *covering*. Covering constraints state to what extent and how the yield of a tectogrammatical category corresponds to that of a phenogrammatical category. Two special kinds of covering are also used in this formalism: *matching*, and *splicing*. These are all discussed in the following subsections. Covering constraints can be declared either globally in a grammar file or locally inside tectogrammatical rules. The scope of local constraints is only the rule in which they have been introduces.

10.3.1 Covering

Covering constraints state that the phonological yield of the tectogrammatical category ϕ consumes that of a phenogrammatical category f and possibly more. This situation is depicted in Figure 10.2.

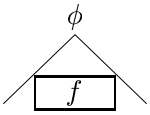

Figure 10.2: Covering

- Global:
  - ϕ `covers` fs: These constraints state that the yield of all ϕ must include all of fs, where fs is a field or region name or a disjunction of field or region names.
  - f `covered_by` Φ: These constraints require that all f be consumed by some Φ, where Φ is a tectogrammatical category or a disjunction of tectogrammatical categories.
- Local: n `covers` fs: This constraint states that the nth daughter mentioned in the rule must include all of fs, where fs is a field name or a disjunction of field names.

As an example, refer to the following hypothetical rule:

```
pp *--> p, nbar,
        {2 covers npr}
```

What the above rule says is that a PP consists of a P and an NBar and that the second daughter in the rule, namely, NBar covers an NPR. In this context the field or region specified will be one that is topologically accessible to the sponsor of the mother node. Therefore, the NPR in the above example has to be topologically accessible to the phenogrammatical edge that resulted in the prediction of that PP. The NBar in this rule could, in principle, be larger than the NPR. For example, it might also contain an extraposed relative clause.

10.3.2 Matching

Matching is a special kind of covering relation. Matching constraints state that a given tectogrammatical category ϕ is identical in its phonological yield to that of a phenogrammatical category f. This situation is depicted in Figure 10.3.

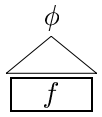

Figure 10.3: Matching

- Global:
  - ϕ `matches` fs: These constraints state that the yield of all ϕ is equivalent to that of some fs.
  - f `matched_by` Φ: These constraints state that the yield of all f is equivalent to that of some Φ.
- Local:
- n `matches` fs: Local matching is a special case of local covering. The only difference is that in matching the daughter covering the field or region cannot be larger than it.

An example of a local `matches` constraint in German grammar is:

```
nbar *--> nbar, rp,
          {   2 matches nf
            ; 2 matches postf
          }
```

This rule states that an NBar consists of another NBar as well as an RP, which matches either a NF (extraposed) or PostF (adjacent to the head noun that it modifies) that is accessible to the sponsor of the mother NBar.

10.3.3 Splicing

Another special case of covering relation is splicing. Splicing refers to a situation where a tectogrammatical category ϕ matches in its phonological yield with more than one phenogrammatical category $f_1 \ldots f_n$. This situation is depicted in Figure 10.4.

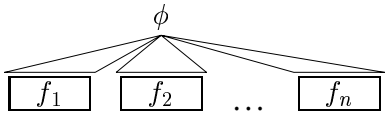

Figure 10.4: Splicing

- Global:
  - ϕ `matches` $f_1 + \ldots + f_n$: These constraints state that the yield of all ϕ is equivalent to that of some $f_1 \ldots f_n$ concatenated.
- Local:
  - n `matches` $f_1 + \ldots + f_n$: These constraints state that the nth daughter of the rule matches in its yield with some $f_1 + \ldots + f_n$ put together.

An example of splicing is given in the following rule:

```
pp *--> p, np,
          {2 matches npr+nf}
```

The above rule states that a PP is made up of a P and a NP, which matches the yield of some NPR and NF put together.

10.3.4 Compaction

Compaction constraints apply to tectogrammatical categories and states that the elements in the list that is its argument be contiguous strings. As a global constraint, compaction applies this condition to all categories that are consistent with any of the elements in the argument list of the constraint.

- Global:
  - `compacts(CatList)` states that all tectogrammatical categories mentioned in `CatList` are always contiguous strings.
- Local:
  - `compacts(N)` states that the nth daughter mentioned in the tectogrammatical rule is a contiguous string. If $n = 0$, the the mother node forms a contiguous string.

For example, the global rule `compacts([s])` says that all Ss form contiguous strings.

10.3.5 Precedence and Immediate Precedence Constraints

Precedence

Precedence constraints mean that the daughter whose index is mentioned on the left-hand side must be entirely located before the daughter whose index is on the right-hand side. Note that this constraint does not make any assumptions about the contiguity of the daughters. As an example, one can provide the following hypothetical rule that states an NP consists of a determiner that precedes but might not be adjacent to an NBar with the same agreement features.

```
np *--> det, nbar,
        {1 < 2}
```

Immediate Precedence

A special kind of precedence is immediate precedence, which means that the two daughters mentioned in the constraint have to be adjacent to one another. This only applies to the rightmost word of the first daughter specified in the constraint and the leftmost word of the second one. This constraint does not assume contiguity of daughters either. The following serves as an example for this and means that an NP consists of a determiner which is immediately followed by an NBar.

```
np *--> det, nbar,
        {1 << 2}
```

10.3.6 Topological Accessibility

The topological parser in ALE is essentially an all-paths chart parser. It performs a phenogrammatical parse of the input string in a bottom-up fashion and once it finds a region that is matched by a tectogrammatical category, it predicts that category and performs a tectogrammatical parse. This continues until all of the input string has been consumed and all parses have been found.

Because this is an all-paths parser, we need to make sure that larger tectogrammatical categories only have access to those smaller categories that have been directly or indirectly predicted by the same entity in phenogrammar. To do this, we have introduced the notion of *sponsorship*. Whenever a phenogrammatical category predicts some tectogrammatical category ϕ by introducing an active edge into the chart, it passes its own ID i to it. We then say that the sponsor of that active edge and all the other active edges introduced in the course of finding ϕ have the same sponsor with the ID i. Active edges only have access to passive edges with the same sponsor. This ensures that clauses and all other categories that can be predicted from phenogrammar are largely parsed independently, which also helps make parsing more efficient. The only time that a category has access to another category with a different sponsor is when it agrees to consume all of the yield of that category and not just parts of it.

For example, in the sentence (10.3.6),

(5) *Ich sagte daβ er den Mann geseben hat.*
I think that he the man seen has
"I think that he saw the man."

once the parser has found the embedded clause, it adds an active edge to the chart for a sentence and passes its own ID to it. This triggers a tectogrammatical parse of the embedded clause. Later on, when parsing the matrix clause, the parser can only gain access to the embedded clause only when it has agreed to take the embedded clause as a whole. It never gains access to the internal subtrees of it as the matrix clause has a different sponsor. This is shown in Figure 10.5.

10.4 Phenogrammatical Parsing

Phenogrammatical parsing (or *pheno-parsing* for short) is performed using a bottom-up right-to-left chart parsing algorithm. As mentioned in

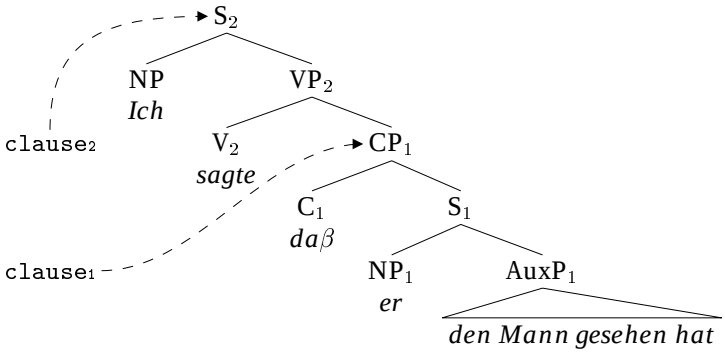

Figure 10.5: Sponsorship and Topological Accessibility

section 9.2, parsing the input string right-to-left and interpreting the rules left-to-right eliminates the need for active edges.

The parser begins by looking up the words in the input stream in its lexicon and finding the grammatical category/categories that the word belongs to. Then using the `matches` constraints, it finds what pheno-category/categories each one of the tecto-categories matches. Using this information, it can then seed the chart with both tecto- and pheno-edges as appropriate. For example, if the third word in the input string is listed as both a pronoun and a marker in the lexicon, and we have the following `matches` rules in the grammar:

```
pron   matches (mf; vf; objf)
marker matches cf
```

then the following pheno-edges will be added to the chart:

```
pheno_edge(2,3,mf,  ID1)
pheno_edge(2,3,vf,  ID2)
pheno_edge(2,3,objf,ID3)
pheno_edge(2,3,cf,  ID4)
```

where `IDn` stands for a unique ID for each edge. The predicate responsible for the addition of pheno-edges to the chart is `add_pheno_edge/5`, which also performs a transitive closure on the accessibility relation between the newly added edge and all its descendents unless one of those descendents has resulted in the prediction of tecto-category. We will talk about tecto-edges in the next section.

Each addition of an edge to the chart triggers a search in the topo rules trying to find a corresponding parse subtree. The search is performed in a failure driven loop using Prolog's call stack as its search space. For instance, if the grammar includes the rule:

```
clause topo [vf,cf,mf,vc,nf]
```

the addition of the edge `pheno_edge(L,R,vf,ID1)` will trigger the following topo rule:

```
topo(L, M, vf, ID1) :-
  pheno_edge(M, N, cf, ID2),
  pheno_edge(N, O, mf, ID3),
  pheno_edge(O, P, vc, ID4),
  pheno_edge(P, R, nf, ID5),
  R>L,
  add_pheno_edge(clause, L, R,
    [ID1-vf,ID2-cf,ID3-mf,ID4-vc,ID5-nf], _).
```

This predicate succeeds if a sequence of correct edges have been found and if at least one of those edges consumes input—i.e., has not been added to the chart because it has been declared optional somewhere in the grammar. The list of `ID-F` pairs is used by `add_pheno_edge/5` for topological accessibility.

In addition, if the grammar contains the following relevant linking and matched-by rules as well:

```
clause <<-- nf
clause matched_by s
```

then the topo predicate mentioned above will have some additional clauses accordingly, as follows:

```
topo(L, M, vf, ID1) :-
  pheno_edge(M, N, cf, ID2),
  pheno_edge(N, O, mf, ID3),
  pheno_edge(O, P, vc, ID4),
  pheno_edge(P, R, nf, ID5),
  R>L,
  add_pheno_edge(clause, L, R,
    [ID1-vf,ID2-cf,ID3-mf,ID4-vc,ID5-nf], ID6),
```

```
bv(L, R, BV),
add_tecto_aedge(s, ID6, [], BV, 0),
add_pheno_edge(nf, L, R, [ID6-link], _).
```

The matched-by clause in the grammar says that the yield of all `clause` regions is matched by the yield of some S. This causes the parser to predict an S whenever it finds a `clause`. In this `topo` predicate, the clause `bv(L, R, BV)` converts the interval spanned by the `clause` region into a bit vector `BV`. We record the yield of tecto-categories in bit vectors because tecto-categories may not always be contiguous. Then it adds an active tecto-edge for an S to the chart. The ID of the *sponsoring* clause is also passed to the active edge. The empty list that is passed as the third argument of `add_tecto_aedge/5` represents the keys that that edge carries (see below). The fourth argument stands for the CanBV of the active edge, which is the parts of the input string that it is allowed to use. The fifth argument stands for the OptBV of the edge, which is the parts of the input string that it is allowed not to use. In this case the OptBV is 0 because the S must consume everything that the `clause` has otherwise the matched-by constraint would not hold. The `add_pheno_edge` has been added because of the linking rule. It adds a new pheno-edge to the chart.

10.5 Tectogrammatical Parsing

10.5.1 Category Graphs and Head Chains

Tectogrammatical parsing (tecto-parsing) is performed after pheno-parsing. The tecto-parsing algorithm used in ALE is a specialised head-corner parsing algorithm. *Head*, in the sense that we require here, must only be a lexical category that is sure to be found inside its phrasal category. For example, one is sure to find a verb in all sentences. In general, this can be a syntactic head, a semantic head, or any other obligatory and lexical category which is computationally advantageous to look for first.

Our tectogrammatical parser is based on the one discussed in Penn and Haji-Abdolhosseini (2002, 2003), and our tectogrammatical rules are those of the Linear Specification Language (LSL, Goetz and Penn, 1997) extended to incorporate topological fields and splicing; LSL itself has a compaction constraint but does not name the resulting contiguous fields/regions, and defines no phenogrammatical structure. Parsing with parallel phenogrammatical and tectogrammatical structures is also very

reminiscent of Ahrenberg's (1989) much earlier proposal to separate linear precedence and contiguity from semantic interpretation in LFG parsing.

Suhre (1999) presented a bottom-up parser for LSL that used an ordered graph to represent the right-hand-sides of rules, and multiple Earley-style "dots" in the graph to track the progress of chart edges through their rules. Daniels and Meurers (2002) re-branded LSL as "Generalised ID/LP" (GIDLP), and proposed a different Earley-style parser. In their parser, only one "dot" is used, and the order in which categories are specified by a grammar-writer (which no longer implicitly states linear precedence assumptions) is used to guide the search for rule daughters. Our parser, although not Earley-style, follows Daniels and Meurers (2002) in using this specification order as a guide. In particular, the heads of a particular category are taken to be those lexical categories accessible by a chain through exclusively leftmost daughters in the specification order. In our view, however, Earley deduction is not sufficiently goal-oriented to be suitable for parsing in a freer word-order language.

We call the chain of head daughters down to a preterminal a *head chain*. For example, a head chain for S in the grammar fragment presented in Figure 10.6 is VP-V. By static analysis of the grammar, ALE can also calculate minimum and maximum *heights* for each chain as well as minimum and maximum *yields* at any given height for each category. *Height* refers to the number of categories below a given node in a syntactic tree. According to the grammar in Figure 10.6 for example, an N occurs at height 1 because there is a lexical item under it and N' occurs at height 2 because there are two nodes under it, namely N and a lexical item. *Yield* or more precisely *phonological yield* refers to the number of words that a given category encompasses. For example in the grammar in Figure 10.6, the yield of an N' can be either 1 (for just a N) or 2 (for a Det and an N). We will discuss this in more detail below.

In order to calculate minimum and maximum heights and yields, given a set of grammar rules, ALE generates a graph for those rules at compile time. There are two types of nodes in each graph: *category nodes* (depicted as circles), and *rule nodes* (depicted as squares). Circular nodes are called category nodes because they could be thought of as representing the left-hand-side of a rule in the grammar. Square nodes are called rule nodes because they could be thought of as the categories in the body (right-hand-side) of a rule. In a category graph, category nodes and rule nodes alternate; i.e., a circular node only points to one or more square node(s), and a square node only points to one or more circular node(s).

Figure 10.6 shows a small piece of grammar and the graph that ALE generates based on it.

S → VP NP
NP → N' S
NP → N'
N' → N Det
N' → N
VP → V NP
VP → V

N → {boy, girl}
Det → {a, the, this}
V → {sees, calls}

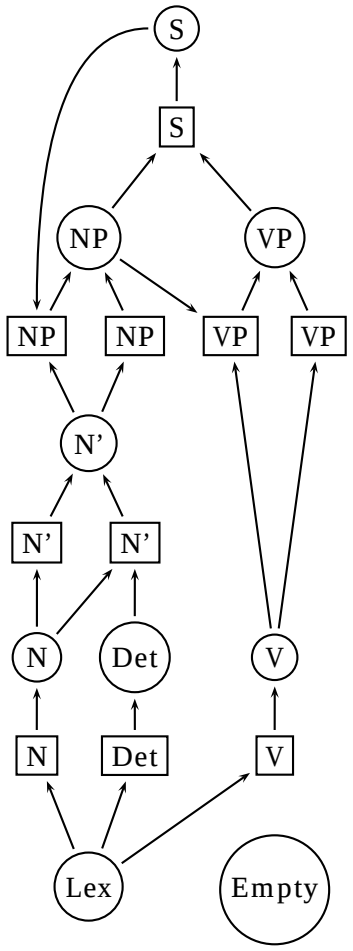

Figure 10.6: Some grammar rules and their graph representation

The maximum yield of category X at a maximum height h is:

$$X_{max}(\leq h) = \max_{0 \leq j \leq h} X_{max}(j)$$

The minimum yield of category X at a maximum height h is:

$$X_{min}(\leq h) = \min_{0 \leq j \leq h} X_{min}(j)$$

These are useful for cases when $X_{max}(h)$ and $X_{min}(h)$ for a given h is not

defined. They also provide a convenient notation for defining the equations derived from tecto-rules.

For every category X, $X_{max}(h)$ is simply defined as the sum of the maximum yields of its daughters at $h-1$ (or lower). It could be, however, that a smaller height tree for X actually has a bigger yield than X at height exactly h because trees at certain multiples of heights could be formed by disjoint combinations of rules. So in the general case, the formula for $X_{max}(h)$ takes the form:

$$\mathrm{S}_{max}(h) = \max \begin{cases} \mathrm{NP}_{max}(h-1) + \mathrm{VP}_{max}(\leq h-1) \\ \mathrm{NP}_{max}(\leq h-1) + \mathrm{VP}_{max}(h-1) \end{cases}$$

This ensures that the tree from which we choose the maximum yield is in fact of height h.

We likewise define $X_{min}(h)$ as the sum of the minimum yields of its daughters (which are at height $h-1$). This is defined as the $X_{min}(h-1)$ of one daughter plus the $X_{min}(\leq h-1)$ of other daughters. To decide for which daughter we need to calculate $X_{min}(h-1)$ and for which we need to calculate $X_{min}(\leq h-1)$, we have to calculate all possibilities and then take the smallest number. For instance, $\mathrm{S}_{min}(h)$ is calculated as follows:

$$\mathrm{S}_{min}(h) = \min \begin{cases} \mathrm{NP}_{min}(h-1) + \mathrm{VP}_{min}(\leq h-1) \\ \mathrm{NP}_{min}(\leq h-1) + \mathrm{VP}_{min}(h-1) \end{cases}$$

Lexical items and empty categories always occur at height 0. Lexical items always have a minimum and maximum yield of 1. Empty categories have a minimum and maximum yield of 0. Lexical categories (preterminals) always occur at height 1 and have minimum and maximum yield of 1. These are the square nodes that directly project from the lexicon. For acyclic rules, calculating X_{min} and X_{max} is very simple because acyclic categories (such as N' in our sample grammar) always have the same minimum and maximum yields and occur at the same height. For example, an N' always occurs at height 2 (lexical items are at height 0, and preterminals are at height 1); and $\mathrm{N'}_{min}(2) = 1$, $\mathrm{N'}_{max}(2) = 2$. This is because we have N' $\rightarrow$ N and N' $\rightarrow$ N Det in this grammar fragment.

Because tecto-rules do not assume order or contiguity, performing bottom-up parsing in the same manner as pheno-parsing is ill-advised. Not using active edges in the parser is also not a very wise approach. Tecto-parsing is performed top-down but in a very goal-oriented fashion in order to prune search. Before an active edge is added to chart, the parser uses head chains in order to make sure that the lexical category

that must necessarily be present inside a category really *is* present. If the parser fails to find the head chain that is required for making the category it is predicting, it will not add the corresponding active edge to the chart. To prune search even further, ALE takes into account the minimum and maximum yields of categories at any given height. If given the number of words and explicit precedence and topological constraints ALE finds out that it can never consume all of the input or it requires more words to build an edge. It will not add that active edge to the chart.

10.5.2 Bit Vector Lattices

As mentioned in the previous section, an active edge contains two bit vectors: CanBV and OptBV. CanBV represents the words that the edge is allowed to take, and OptBV represents the words that it is allowed not to take. At run time, we frequently see that an active edge gets added to the chart that only slightly differs from another active edge that has already been introduced to the chart. For example, let us assume that the chart already includes an active edge with the CanBV 00001111 and the OptBV 11110000, and another edge with the same category and with the CanBV 00001011 and the OptBV 11110100 gets added to the chart. It is obvious that the two edges differ only in one bit (namely the third bit): in the first edge, the bit belongs to the CanBV, but in the second edge, it belongs to the OptBV. In cases like this, when a new active edge differs only slightly with an existing one, we want to make sure that we adjust our search space so that we do not redo the search that we did when the first edge was added to the chart. Table 10.1 shows all the possible basic situations that may arise in such a case. For ease of exposition, we will not use actual bit vectors here and instead use number sets to stand for the word numbers in the input string. For example, $\{1,2,3\}$ under the CanBV column means that the edge is allowed to take the first, second, and third words from the input string (which is equivalent to the bit vector 000111) in this example. The last column of Table 10.1 shows the subsumption condition that must hold in a unification-based setting. Here A stands for the category of the first edge and B for the category of the second edge.

As is shown in Table 10.1, there are six basic situations. The rows marked 'a' represent the bit vectors of the first active edge, and the rows marked 'b' represent the bit vectors of the new active edge. In the first situation, a word moves from the CanBV to the OptBV. In the second situation, a word drops from the CanBV. In the third, a word moves from OptBV to CanBV, and in the fourth, a word drops from OptBV. In the fifth

		CanBV	OptBV	Subsumption Condition
1.	a.	{1,2,3}	{4,5,6}	
	b.	{1,2}	{3,4,5,6}	B $\sqsubseteq$ A
2.	a.	{1,2,3}	{4,5,6}	
	b.	{1,2}	{4,5,6}	A $\sqsubseteq$ B
3.	a.	{1,2,3}	{4,5,6}	
	b.	{1,2,3,4}	{5,6}	A $\sqsubseteq$ B
4.	a.	{1,2,3}	{4,5,6}	
	b.	{1,2,3}	{5,6}	A $\sqsubseteq$ B
5.	a.	{1,2,3}	{4,5,6}	
	b.	{1,2,3}	{4,5,6,7}	B $\sqsubseteq$ A
6.	a.	{1,2,3}	{4,5,6}	
	b.	{1,2,3,7}	{4,5,6}	A $\sqsubseteq$ B

Table 10.1: Possible relationships between the bit vectors of two similar active edges

and sixth situation, a word is added to OptBV and CanBV respectively.

In order to understand the relationships of these situations with respect to the search space that we need to cover, it helps to think of the bit vectors in terms of lattices of word sets. Let the bottom of the lattice stand for the set corresponding to CanBV and the top stand for the set corresponding to CanBV $\vee$ OptBV. In this case, the order relation is defined with set inclusion meaning that $x < y$ iff $x \subset y$. For example, the lattice corresponding to the bit vectors in Table 10.1 item 1a, is shown in Figure 10.7.

What the lattice in Figure 10.7 means is having an active edge with the CanBV of {1,2,3} and OptBV of {4,5,6} entails having already covered the search spaces provided by CanBV {1,2,3,4}, {1,2,3,5}, etc. and the same OptBV. Let us now see what happens in the first situation given in Table 10.1. If one word moves from CanBV to OptBV, it means that we have given our lattice a new bottom, {1,2}. The new lattice corresponding to the new search space is shown in Figure 10.8.

The lattice of Figure 10.7 is a sub-lattice of the one in Figure 10.8, which means that the new active edge overlaps in its search space with that of the old one. What the parser does in this case is to adjust the bit vectors to CanBV {1,2} and OptBV {4,5,6} thus restricting the search space to the part of the lattice shown in dashed lines. This means that case 1 in Table 10.1 effectively reduces to case 2.

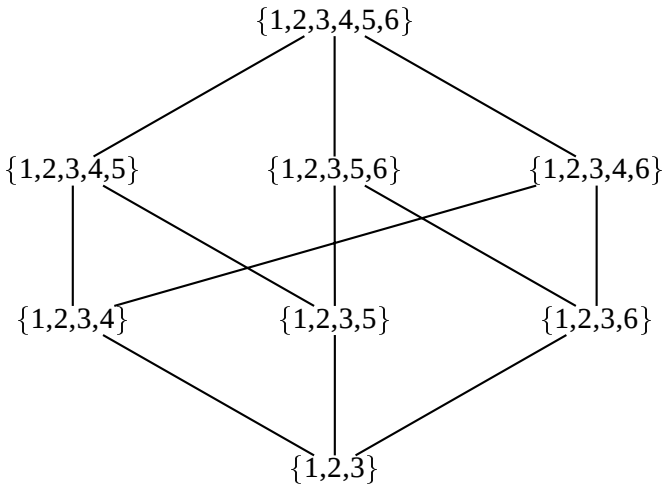

Figure 10.7: The lattice representation of the bit vectors in Table 10.1 item 1a

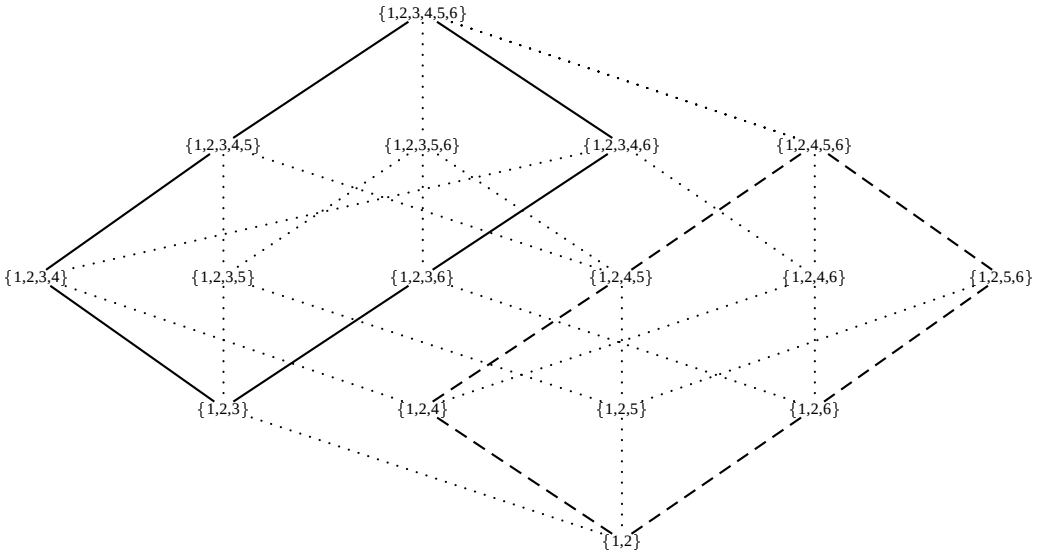

Figure 10.8: The lattice representation of the bit vectors in Table 10.1 item 1a

The third, fourth, and fifth cases are simple. In all these cases, the lattice representation of the new active edge is a sub-lattice of the old one, which means that the search space has already been covered and the parser does not need to do anything. In these cases, the edge will not be added to the chart. It is only in the last case that the parser cannot make use of the existing information and has to perform a full search by adding the edge to the chart intact. This is because the lattice representation of the new edge is completely disjoint from the old one.

Chapter 11

Generation

As pointed out in the User's Guide, ALE uses the Semantic Head-Driven Generation algorithm of Shieber et al. (1990). The application of this algorithm to typed feature structures and ALE has been discussed in Popescu (1996) and Penn and Popescu (1997).

11.1 The Algorithm

Semantic-head-driven generation makes use of the notion of a *semantic head* of a rule, a daughter whose semantics is shared with the mother. In semantic-head-driven generation, there are two kinds of rules: *chain rules*, which have a semantic head, and *non-chain rules*, which lack such a head. These two subsets are processed differently during the generation process.

Given a feature structure, called the *root* goal, to generate a string for, the generator builds a new feature structure that shares its semantic information (using the user-defined semantics predicate with the second argument instantiated) and finds a *pivot* that unifies with it. The pivot is the lowest node in a derivation tree that has the same semantics as the root. The pivot may be either a *lexical entry* or *empty category* (the base cases), or the *mother category of a non-chain rule*. Once a pivot is identified, one can recursively generate top-down from the pivot using non-chain rules. Since the pivot must be the *lowest*, there can be no lower semantic heads, and thus no lower chain-rule applications. Just as in parsing, the daughters of non-chain rules are processed from left to right.

Once top-down generation from the pivot is complete, the pivot is linked to the root bottom-up by chain rules. At each step, the current

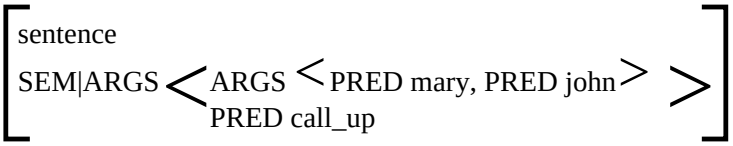

Figure 11.1: The initial root.

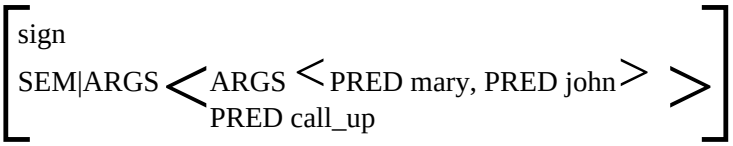

Figure 11.2: The semantics template.

chain node (beginning with the pivot) is unified with the semantic head of a chain-rule, its non-head sisters are generated recursively, and the mother becomes the new chain node. The non-head daughters of chain rules are also processed from left to right. The base case is where the current chain node unifies with the root.

The following example illustrates this better. Suppose that the generator is given the goal description:

```
(sentence,
 sem:(pred:decl,
      args:[(pred:call_up,
             args:[pred:mary,pred:john])]))
```

Figure 11.1 shows the initial root (the most general satisfier of the input description); and Figure 11.2 shows the template that ALE uses to find a pivot. Next (Figure 11.3), a pivot is selected, in this case by unifying the template with the mother category of the non-chain rule, `sentence1`:

```
sentence1 rule
  (sentence,sem:(pred:decl,
                 args:[S]))
  ===>
  cat> (s,form:finite,
          sem:S).
```

We can tell that `sentence1` is a non-chain rule because it lacks a `sem_head>` daughter, unlike, for example, the chain rule `s`:

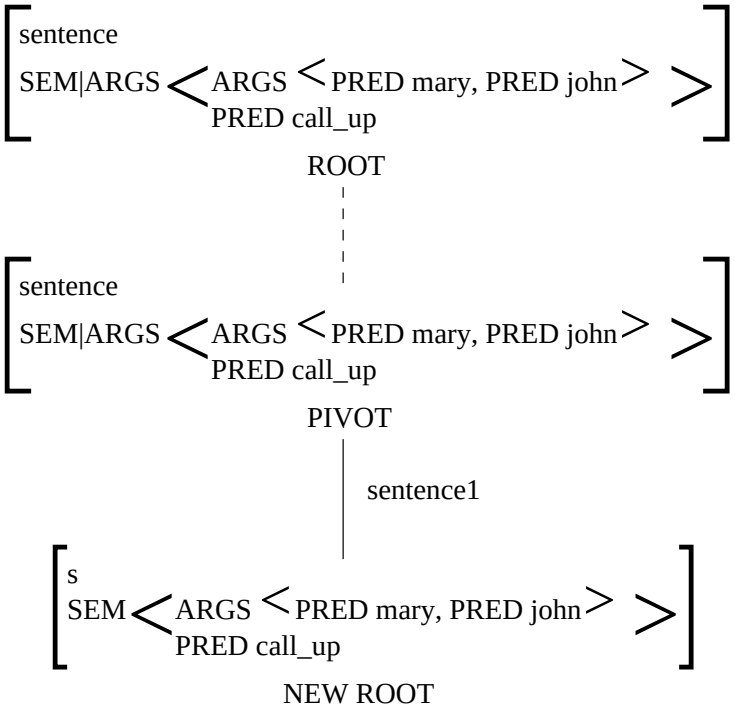

Figure 11.3: The first pivot is found.

```
s rule
(s,form:Form,
   sem:S)
===>
cat> Subj,
sem_head>
  (vp,form:Form,
      subcat:[SUBJ],
             sem:S).
```

The only daughter of `sentence1` becomes the new root.

The pivot chosen for that root is the lexical entry for `calls`, which is obtained by applying the lexical rule, `sg3`, to the first entry for `call` in the grammar. That pivot has no daughters, so it must now be connected by chain rules to the new root in Figure 11.3. The chain rule, `vp1`, is chosen, its semantic head is unified with the entry for `calls`, its one non-head daughter is recursively generated (which simply succeeds by unifying with the lexical entry for `john`), and its mother becomes the new chain node (Figure 11.4).

Again (Figure 11.5), chain rule `vp1` is chosen, its semantic head is unified with the new pivot, its non-head daughter is recursively generated by unifying with the lexical entry for `up`, and its mother becomes the next chain node.

Finally, the chain rule, `s` is chosen. Its non-head daughter is recursively generated by unifying with the lexical entry for `mary`, and its mother, the new chain node, unifies with the new root (Figure 11.6). With generation below the pivot of Figure 11.3 having been completed, it is linked to its root directly by unification, yielding the solution, `mary calls john up`.

11.2 Pivot Checking

ALE's generator uses a simple depth-first search strategy, displaying solutions as it finds them. As a result, it is not complete. Following the suggestion made in Shieber et al. (1990), ALE also checks whether there are *semantic head* $\rightarrow$ *mother* sequences that could possibly link a potential pivot to the current root before recursively generating its non-head daughters. If not, then the pivot is discarded. Semantic-head-driven generators that do not prune away such bad pivots from the search tree run a greater risk of missing solutions because top-down generation of the bad

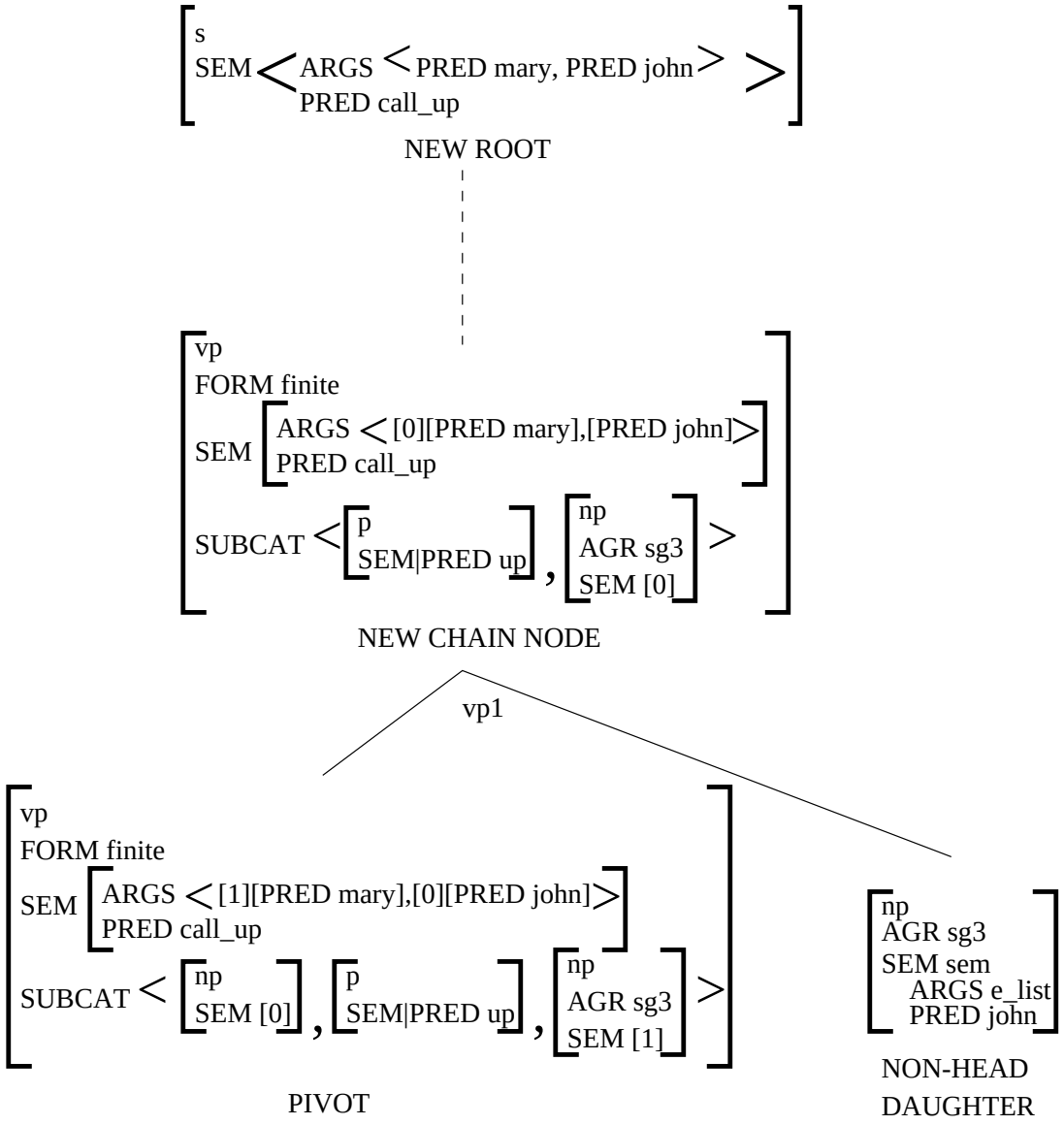

Figure 11.4: First chain rule application.

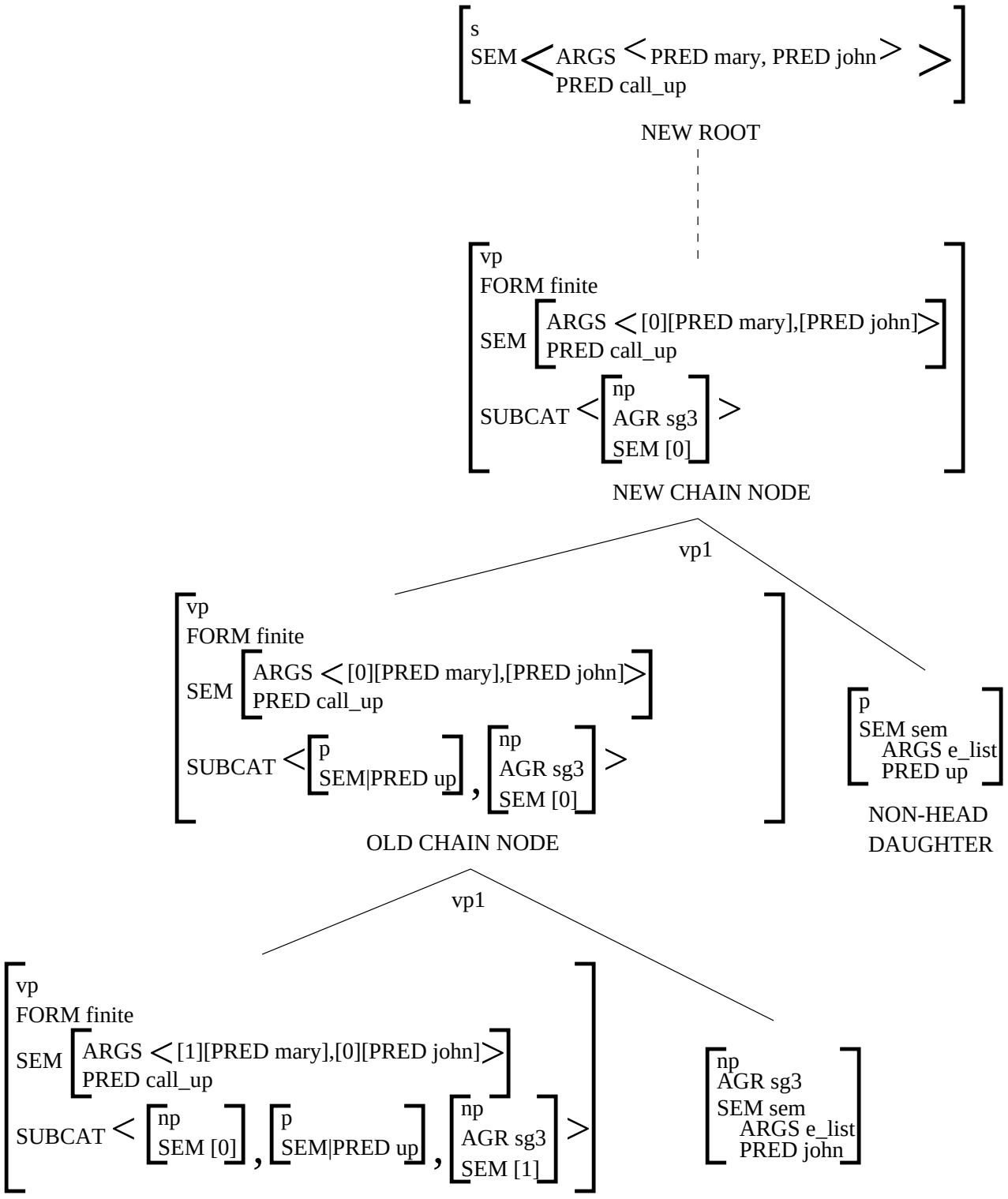

Figure 11.5: Second chain rule application.

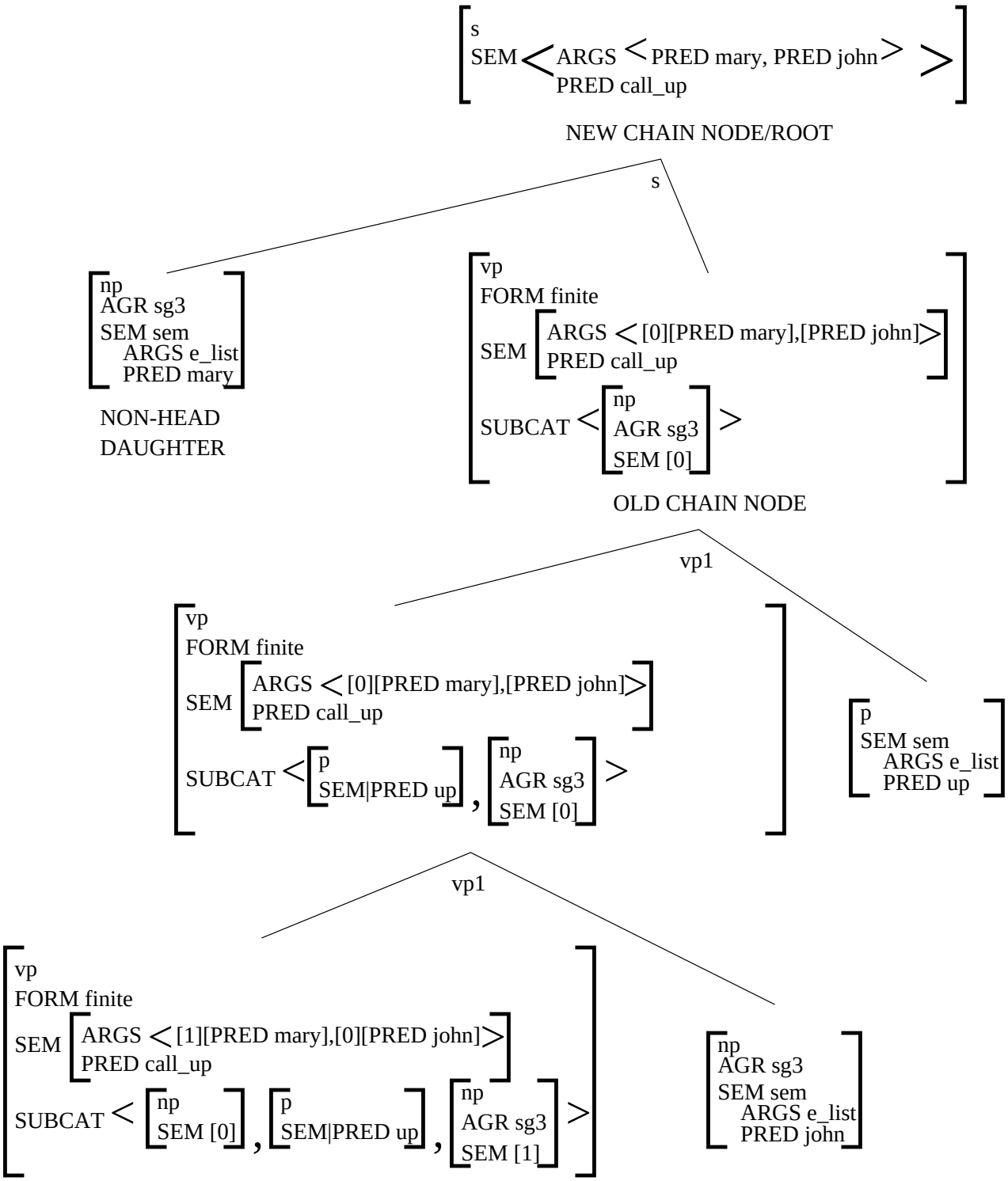

Figure 11.6: Third chain rule application and unification.

pivot's non-head daughters may not terminate, even though it can never yield solutions. This check is valuable because it incorporates *syntactic* information from the mother and semantic head into the otherwise *semantic* prediction step.

It also creates another termination problem, however, namely the potential for infinitely long *semantic head* $\rightarrow$ *mother* sequences in some grammars. To avoid this, ALE requires the user to specify a bound on the length of chain rule sequences at compile-time.

Chapter 12

SLD Resolution

The term *resolution* in logic refers to a mechanical method for proving statements in first-order logic. It is applied to two clauses in a sentence, and by unification, it eliminates a literal that occurs positive in one clause and negative in the other. A literal stated in the antecedent of an implication is negative because an implication $P \rightarrow Q$ is equivalent to $\neg P \vee Q$. Therefore, in

(6) $(man(X) \rightarrow mortal(X)) \wedge man(socrates)$

we can unify $socrates$ with X. This will give us

(7) $(man(socrates) \rightarrow mortal(socrates)) \wedge man(socrates)$

which is equivalent to

(8) $(\neg man(socrates) \vee mortal(socrates) \wedge man(socrates))$

By distribution, we have

(9) $(man(socrates) \wedge \neg man(socrates)) \vee (man(socrates) \wedge mortal(socrates))$

Since $P \wedge \neg P = false$, we have

(10) $man(socrates) \wedge mortal(socrates)$

Through this resolution process, we proved $mortal(socrates)$. Resolution with backtracking is the basic control mechanism in Prolog.[1]

[1]The above information has been adapted from `http://wombat.doc.ic.ac.uk`.

SLD resolution is a term used in logic programming to refer to the control strategy used in such languages to resolve issues of nondeterminism. By definition, SLD resolution is linear resolution with a selection function for definite sentences. A definite sentence has exactly one positive literal in each clause and this literal is selected to be resolved upon, i.e., replaced in the goal clause by the conjunction of negative literals which form the body of the clause.

SLD resolution in ALE is identical to what Prolog does. In fact, in implementation ALE relies on Prolog for SLD resolution. For more information on how Prolog handles nondeterminism, refer to chapter 5 of Sterling and Shapiro (1986).

ALE compiles a clause declaration into instructions, and processes these instructions as follows:

- Enforce the descriptions on the head arguments, in left-to-right order.
- Enforce the descriptions on the arguments of the first body goal, in left-to-right order.
- Call the first body goal.
- Repeat the last two steps on all remaining body goals.

A few built-in operators merit considerationl. `=@/2` is compiled to an equality check (using Prolog `==/2`) on the `Tags` of the ALE data structure. `=/2` is compiled to unification. `when/2` is compiled as stated in Chapter 6. Prolog hooks are compiled directly to Prolog calls. Cut, shallow-cut and negation-by-failure are compiled to their Prolog equivalents.

Bibliography

Aït-Kaći, H., R. Boyer, P. Lincoln, and R. Nasr (1989). Efficient implementation of lattice operations. *Transactions on Programming Languages and Systems 11*(1), 115–146.

Carpenter, B. (1992). *The Logic of Typed Feature Structures: With Applications to Unification Grammars, Logic Programs and Constraint Resolution*. Cambridge Tracts in Theoretical Computer Science 32. Cambridge: Cambridge University Press.

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

Curry, H. B. (1961). Some logical aspects of grammatical structure. In Jacobson (Ed.), *Structure of Language and its Mathematical Aspects: Proceedings of the Twelfth Symposium in Applied Mathematics*, pp. 56–68. American Mathematical Society.

Daniels, M. and W. D. Meurers (2002). Improving the efficiency of parsing with discontinuous constituents. In S. Wintner (Ed.), *Proceedings of NLULP'02: The 7th International Workshop on Natural Language Understanding and Logic Programming*, Number 92 in Datalogiske Skrifter, Copenhagen, pp. 49–68. Roskilde Universitetscenter.

Meurers, W. D. (2001). On expressing lexical generalizations in HPSG. *Nordic Journal of Linguistics 24*, 161–127.

Meurers, W. D. and G. Minnen (1997). A computational treatment of lexical rules in HPSG as covariation in lexical entries. *Computational Linguistics 23*(4), 543–568.

Penn, G. (2000). The algebraic structure of transitive closure and its application to attributed type signatures. *Grammars 3*, 295–312.

Penn, G. and M. Haji-Abdolhosseini (2002). Topological parsing: A linearisation parser specification submitted to project MilCA/A4 of the BMBF. Technical Report.

Penn, G. and M. Haji-Abdolhosseini (2003). Topological parsing. In *Proceedings of the 10th Conference of the European Chapter of the Association for Computational Linguistics*, pp. 283–290.

Penn, G. and C. Munteanu (2003). A tabulation-based parsing method that reduces copying. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, pp. ??–??

Penn, G. and O. Popescu (1997). Head-driven generation and indexing in ALE. In *Proceedings of Workshop on Computational Environments for Grammar Development and Linguistic Engineering (ENVGRAM)*. 35th ACL/8th EACL.

Pollard, C. and I. Sag (1994). *Head-Driven Phrase Structure Grammar*. Studies in Contemporary Linguistics. Chicago: CSLI.

Popescu, O. (1996). Head-driven generation for typed feature structures. Master's thesis, Cargegie Mellon University.

Shieber, S., C. Pereira, G. van Noord, and R. Moore (1990). Semantic-head-driven generation. *Computational Linguistics 16*(1), 30–42.

Sterling, L. and E. Shapiro (1986). *The Art of Prolog: Advanced Programming Techniques*. MIT Press.

Suhre, O. (1999). Computational aspects of a grammar formalism for languages with freer word order. Master's thesis, Eberhard-Karls-Universität Tübingen.