DATA TYPES IN COMPUTATIONAL PHONOLOGY

Ewan Klein
University of Edinburgh, Centre for Cognitive Science
2 Buccleuch Place, Edinburgh EH8 9LW, Scotland
Email: klein@ed.ac.uk

ABSTRACT

This paper examines certain aspects of phonological structure from the viewpoint of abstract data types. Our immediate goal is to find a format for phonological representation which will be reasonably faithful to the concerns of theoretical phonology while being rigorous enough to admit a computational interpretation. The longer term goal is to incorporate such representations into an appropriate general framework for natural language processing. [1]

1 Introduction

One of the dominant paradigms in current computational linguistics is provided by unification-based grammar formalisms. Such formalisms (cf. (Shieber 1986; Kasper & Rounds 1986)) describe hierarchical feature structures, which in many ways would appear to be an ideal setting for formal phonological analyses. Feature bundles have long been used by phonologists, and more recent work on so-called feature geometry (e.g. (Clements 1985; Sagey 1986)) has introduced hierarchy into such representations. Nevertheless, there are reasons to step back from standard feature-based approaches, and instead to adopt the algebraic perspective of abstract data types (ADT) which has been widely adopted in computer science. One general motivation, which we shall not explore here, is that the activity of grammar writing, viewed as a process of programme specification, should be amenable to *stepwise refinement* in which the set of (not necessarily isomorphic) models admitted by a loose specification is gradually narrowed down to a unique algebra (cf. (Sannella & Tarlecki 1987) for an overview, and (Newton *in prep.*) for the application to grammar writing). A second motivation, discussed in detail by (Beierle & Pletat 1988; Beierle & Pletat 1989; Beierle et al. 1988), is to use equational ADTs to provide a mathematical foundation for feature structures. A third motivation, dominant in this paper, is to use the ADT approach to provide a richer array of explicit data types than are readily admitted by 'pure' feature structure approaches. Briefly, in their raw form, feature terms (i.e., formalisms for describing feature structures) do not always provide a perspicuous format for representing structure.

On the ADT approach, complex data types are built up from atomic types by means of **constructor functions**. For example, _._ (where we use the underscore '_' to mark the position of the function's arguments) creates elements of type `List`. A data type may also have **selector functions** for taking data elements apart. Thus, selectors for the type `List` are the functions `first` and `last`. Standard feature-based encoding of lists uses only selectors for the data type; i.e. the feature labels FIRST and LAST in

(1) $\text{FIRST} : \sigma_1 \sqcap \text{LAST} : (\text{FIRST} : \sigma_2 \sqcap \text{LAST} : \mathit{nil})$

However, the list constructor is left implicit. That is, the feature term encoding tells you how lists are pulled apart, but does not say how they are built up. When we confine our attention just to lists, this is not much to worry about. However, the situation becomes less satisfactory when we attempt to encode a larger variety of data structures into one and the same feature term; say, for example, standard lists, associative lists (i.e. strings), constituent structure hierarchy, and autosegmental association. In order to distinguish adequately between elements of such data types, we really need to know the logical properties of their respective constructors, and this is awk-

[1] The work reported in this paper has been carried out as part of the research programmes of the Human Communication Research Centre, supported by the UK Economic and Social Research Council and the project *Computational Phonology: A Constraint-Based Approach*, funded by the UK Science and Engineering Research Council, under grant GR/G-22084. I am grateful to Steven Bird, Kimba Newton and Tony Simon for discussions relating to this work.

ward when the constructors are not made explicit. For computational phonology, it is not an unlikely scenario to be confronted with such a variety of data structures, since one may well wish to study the complex interaction between, say, non-linear temporal relations and prosodic hierarchy. As a vehicle for computational implementation, the uniformity of standard attribute/value notation is extremely useful. As a vehicle for theory development, it can be extraordinarily unperspicuous.

The approach which we present here treats phonological concepts as abstract data types. A particularly convenient development environment is provided by the language OBJ (Goguen & Winkler 1988), which is based on order sorted equational logic, and all the examples given below (except where explicitly indicated to the contrary) run in the version of OBJ3 released by SRI in 1988. The denotational semantics of an OBJ module is an algebra, while its operational semantics is based on order sorted rewriting. § 1.1 and 1.2 give a more detailed introduction into the formal framework, while § 2 and 3 illustrate the approach with some phonological examples.

1.1 Abstract Data Types

A data type consists of one or more domains of data items, of which certain elements are designated as basic, together with a set of operations on the domains which suffice to generate all data items in the domains from the basic items. A data type is **abstract** if it is independent of any particular representational scheme. A fundamental claim of the ADJ group (cf. (Goguen, Thatcher & Wagner 1976)) and much subsequent work (cf. (Ehrig & Mahr 1985)) is that abstract data types are (to be modelled as) algebras; and moreover, that the models of abstract data types are initial algebras.[2]

The **signature** of a many-sorted algebra is a pair $\Sigma = \langle S, O \rangle$ consisting of a set S of sorts and a set O of constant and operation symbols. A **specification** is a pair $\langle \Sigma, \mathcal{E} \rangle$ consisting of a signature together with a set $\mathcal{E}$ of equations over terms constructed from symbols in O and variables of the sorts in S. A **model** for a specification is an algebra over the signature which satisfies all the equations $\mathcal{E}$. Initial algebras play a special role as the semantics of an algebra. An initial algebra is minimal, in the sense expressed by the principles 'no junk' and 'no confusion'. 'No junk' means that the algebra only contains data which are denoted by variable-free terms built up from operation symbols in the signature. 'No confusion' means that two such terms t and t' denote the same object in the algebra only if the equation $t = t'$ is derivable from the equations of the specification.

Specifications are written in a conventional format consisting of a declaration of `sorts`, operation symbols (`op`), and equations (`eq`). Preceding the equations we list all the variables (`var`) which figure in them. As an illustration, we give below an OBJ specification of the data type `LIST1`.

```
(2) obj LIST1 is sorts Elt List .
      op nil : -> List .
      op _._ : Elt List -> List .
      op head_ : List -> Elt .
      op tail_ : List -> List .
      var X : Elt .
      var L : List .
      eq (X . nil) = X .
      eq head(X . L) = X .
      eq tail(X . L) = L .
    endo
```

The sort list between the `:` and the `->` in an operation declaration is called the **arity** of the operation, while the sort after the `->` is its **value sort**. Together, the arity and value sort constitute the **rank** of an operation. The declaration `op nil : -> Elt` means that `nil` is a constant of sort `Elt`.

The specification (2) fails to guarantee that there are any objects of `Elt`. While we could of course add some constants of this sort, we would like to have a more general solution. In a particular application, we might want to define phonological words as a `List` of syllables (plus other constraints, of course), and phonological phrases as a `List` of words. That is, we need to **parameterize** the type `LIST1` with respect to the class of elements which constitute the lists.

Before turning to parameterization, we will first see how a many-sorted specification language is generalized to an order sorted language by introducing a subsort relation.

Suppose, for example, that we adopt the claim

[2] An initial algebra is characterized uniquely up to isomorphism as the semantics of a specification: there is a unique homomorphism from the initial algebra into every algebra of the specification.

that all syllables have *CV* onsets[3]. Moreover, we wish to divide syllables into the subclasses heavy and light. Obviously we want heavy and light syllables to inherit the properties of the class of all syllables, e.g., they have *CV* onsets. We use `Heavy < Syll` to state that `Heavy` is a subsort of the sort `Syll`. We interpret this to mean that the class of heavy syllables is a subset of the class of all syllables. Now, let `onset_ : Syll -> Mora` be an operation which selects the first mora of a syllable, and let us impose the following constraint (where `Cv` is a subsort of `Mora`):

```
(3) var S : Syll . var CV : Cv .
    eq onset S = CV .
```

Then the framework of order sorted algebra ensures that `onset` is also defined for objects of sort `Heavy`.

Returning to lists, the specification in (4) (slightly simplified from that used by (Goguen & Winkler 1988)) introduces `Elt` and `NeLists` (non-empty lists) as subsorts of `List`, and thereby improves on `LIST1` in a number of respects. In addition, the specification is **parameterized**. That is, it characterizes a list of `X`s, where the parameter `X` can be instantiated to any module which satisfies the condition `TRIV`; the latter is what (Goguen & Winkler 1988) call a 'requirement theory', and in this case simply imposes on any input module that it have a sort which can be mapped to the sort `Elt`.

```
(4) obj LIST[X :: TRIV] is sorts List NeList .
      subsorts Elt < NeList < List .
      op nil : -> List .
      op _._ : List List -> List .
      op _._ : NeList List -> NeList .
      op head_ : NeList -> Elt .
      op tail_ : NeList -> List .
      var X : Elt .
      var L : List .
      eq (X . nil) = X .
      eq head(X . L) = X .
      eq tail(X . L) = L .
    endo
```

Notice that the list constructor `_._` now performs the additional function of *append*, allowing two lists to be concatenated. In addition, the selectors have been made 'safe', in the sense that they only apply to objects (i.e., nonempty lists) for which they give sensible results; for what, in `LIST1`, would have been the meaning of `head(nil)`?

[3]Here, the term ONSET refers to the inital mora of a syllable in Hyman's (1984) version of the moraic theory.

2 Metrical Trees

As a further illustration, we give below a specification of the data type `BINTREE`. This module has two parameters, both of whose requirement theories are `TRIV`.[4]

```
(5) obj BINTREE[NONTERM TERM :: TRIV] is
      sorts  Tree Netree .
      subsorts Elt.TERM Netree < Tree .
      op _[_,_] : Elt.NONTERM Tree Tree -> Netree .
      op _[_] : Elt.NONTERM Elt.TERM -> Tree .
      op label_ : Tree -> Elt.NONTERM .
      op left_ : Netree -> Tree .
      op right_ : Netree -> Tree .
      vars E1 E2 : Tree .
      vars A : Elt.NONTERM .
      eq label (A [ E1 , E2 ]) = A .
      eq label (A [ E1 ]) = A .
      eq left (A [ E1 , E2 ]) = E1 .
      eq right (A [ E1 , E2 ]) = E2 .
      endo
```

We can now instantiate the formal parameters of the module in (5) with input modules which supply appropriate sets of nonterminal and terminal symbols. Let us use uppercase quoted identifiers (elements of the OBJ module `QID`) for nonterminals, and lower case for terminals. The specification in (5) allows us to treat terminals as trees, so that a binary tree, rooted in a node `'A`, can have terminals as its daughters. However, we also allow terminals to be directly dominated by a non-branching mother node. Both possibilities occur in the examples below. (6) illustrates the instantiation of formal parameters by an actual module, namely `QID`, using the **make** construct.

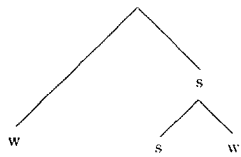
```
(6) make BINTREE-QID is BINTREE[QID,QID] endm
```

The next example shows some reductions in this module, obtained by treating the equations as rewrite rules applying from left to right.

[4]The notation `Elt.NONTERM`, `Elt.TERM` utilizes a **qualification** of the sort `Elt` by the input module's parameter label; this is simply to allow disambiguation.

```
(7)  left ('A['a,'b]) .
         ~> 'a
     left ('A['B['a],'C['b]]) .
         ~> 'B['a]
     left ('A['B['a,'b],'c]) .
         ~> 'B['a,'b]
     right(left ('A[('B['a,'b]),'c])) .
         ~> 'b
     label ('A['a,'b]) .
         ~> 'A
     label(right ('A['a,'B['b,'c]])) .
         ~> 'B
```

Suppose we now wish to modify the definition of binary trees to obtain metrical trees, These are binary trees whose branches are ordered according to whether they are labelled 's' (strong) or 'w' (weak).



In addition, all trees have a distinguished leaf node called the 'designated terminal element' (dte), which is connected to the root of the tree by a path of 's' nodes.

Let us define 's' and 'w' to be our nonterminals:

```
(8)  obj MET is
       sorts Label .
       ops s w : -> Label .
       endo
```

In order to build the data type of metrical trees on top of binary trees, we can **import** the module **BINTREE**, suitably instantiated, using OBJ's **extending** construct. Notice that we use MET to instantiate the parameter which fixes BINTREE's set of nonterminal symbols.[5]

```
(9)  obj METTREE is extending
              BINTREE[MET,QID]*(sort Id to Leaf) .
       op dte_ : Tree -> Leaf .
       var L : Leaf .
       vars T1 T2 : Tree .
       vars X : Label .
       eq dte ( X [ L ] ) = L .
       ceq dte ( X [ T1 , T2 ]) = dte T1
                 if label T1 == s .
       ceq dte ( X [ T1 , T2 ]) = dte T2
                 if label T2 == s .
       endo
```

[5] The * construct tells us that the principal sort of QID, namely Id, is mapped (by a signature morphism) to the sort Leaf in METTREE. ceq signals the presence of a conditional equation. == is a built-in polymorphic equality operation in OBJ.

The equations state that the **dte** (designated terminal element) of a tree is the **dte** of its strong subtree. Another way of stating this is that the information about **dte** element of a subtree T is **percolated** up to its parent node, just in case T is the 's' branch of that node.

The specification **METTREE** can be criticised on a number of grounds. It has to use conditional equations in a cumbersome way to test which daughter of a binary tree is labelled 's'. Moreover, it fails to capture the restriction that no binary tree can have daughters which are both weak, or both strong. That is, it fails to capture the essential property of metrical trees, namely that metrical strength is a *relational* notion.

What we require is a method for encoding the following information at a node: "my left (or right) daughter is strong". One economical method of doing this is to label (all and only) branching nodes in a binary tree with one of the following two labels: 'sw' (my left daughter is strong), 'ws' (my right daughter is strong). Thus, we replace MET with the following:

```
obj MET2 is
sorts Label
ops sw ws : -> Label .
endo
```

We can now simplify both BINTREE and METREE:

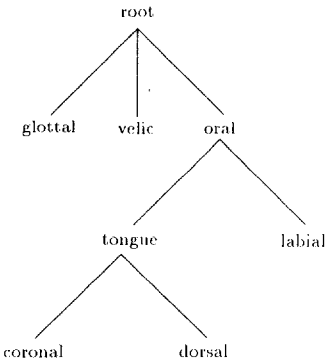
```
obj BINTREE2[NONTERM TERM :: TRIV] is
sorts  Tree Netree .
subsorts Elt.TERM Netree < Tree .
op _[_,_] : Elt.NONTERM Tree Tree -> Netree .
op label_ : Tree -> Elt.NONTERM .
op left_ : Netree -> Tree .
op right_ : Netree -> Tree .
vars E1 E2 : Tree .
vars A : Elt.NONTERM .
eq label (A [ E1 , E2 ]) = A .
eq left (A [ E1 , E2 ]) = E1 .
eq right (A [ E1 , E2 ]) = E2 .
endo

obj METTREE2 is extending
       BINTREE2[MET2,QID]*(sort Id to Leaf) .
op hte_ : Tree -> Leaf .
```

```
var L : Leaf .
vars T1 T2 : Tree .
eq dte L = L .
eq dte T = if label T == sw
                      then dte(left T)
                      else dte(right T) fi .
endo
```

3 Feature Geometry

The particular feature geometry we shall specify here is based on the articulatory structure defined in (Browman & Goldstein 1989).[6] The five active articulators are grouped into a hierarchical structure involving a tongue node and an oral node, as shown in the following diagram.



This structure is specified via term constructors {_,_} and {_,_,_} which give a standard positional encoding of features. Each feature value is expressed as a natural number between 0 and 4, representing the **constriction degree** of the corresponding articulator. For example, the term {4,0}: Tongue is an item of sort Tongue consisting of the value 4 for the feature CORONAL and 0 for the DORSAL; this in turn expresses a situation where there is maximal constriction of the tongue tip, and minimal constriction of the tongue body. Of course, this encoding is rather crude, and possibly sacrifices clarity for concision. However, it suffices as a working example. We will return to constriction degrees below.

The four sorts Gesture, Root, Oral and Tongue in (10) and the first three operators capture the desired tree structure, using an approach which should be familiar by now. For example, the third constructor takes the constriction degrees of Glottal and Velic gestures, and combines them with a complex item of sort Oral to build an item of sort Root. The specification imports the module NAT of natural numbers to provide values for constriction degrees.

(10)
```
obj FEATS is
    extending NAT .
    sorts  Gesture Root Oral Tongue .
    subsorts Nat Root Oral Tongue < Gesture .
    op {_,_} : Nat Nat -> Tongue .
    op {_,_} : Tongue Nat -> Oral .
    op {_,_,_} : Nat Nat Oral -> Root .
    op _!coronal : Tongue -> Nat .
    op _!dorsal : Tongue -> Nat .
    op _!labial : Oral -> Nat .
    op _!tongue : Oral -> Tongue .
    op _!glottal : Root -> Nat .
    op _!velic : Root -> Nat .
    op _!oral : Root -> Oral .
    vars C C1 C2 : Nat .
    vars O : Oral .
    vars T : Tongue .
    eq { C1 , C2 } !coronal = C1 .
    eq { C1 , C2 } !dorsal = C2 .
    eq { T , C } !tongue = T .
    eq { T , C } !labial = C .
    eq { C1 , C2 , O } !glottal = C1 .
    eq { C1 , C2 , O } !velic = C2 .
    eq { C1 , C2 , O } !oral = O .
    endo
```

We adopt the notational convention of prepending a '!' to the name of selectors which correspond directly to features. For example, the !coronal selector is a function defined on complex items of sort Tongue which returns an item of sort Nat, representing the constriction degree value for coronality.

Some illustrative reductions in the FEATS module are given below.

(11)
```
{3,4,{{4,1},1}} !oral .
    ~> {{4,1},1}
{3,4,{{4,1},1}} !oral !tongue .
    ~> {4,1}
{3,4,{{4,1},1}} !oral !tongue !coronal .
    ~> 4
```

In the ADT approach to feature structures, reentrancy is represented by equating the values of selectors. Thus, suppose that two segments S1, S2 share a voicing specification. We can write this as follows:

[6] For space reasons we have omitted any discussion of Browman & Goldstein's constriction location (CL) and constriction shape (CS) parameters. We also have omitted the supralaryngeal node, since its phonological role is somewhat dubious.

(12) `S1 !glottal = S2 !glottal`

This structure sharing is consistent with one of the main motivating factors behind autosegmental phonology, namely, the undesirability of rules such as [α voice] $\rightarrow$ [α nasal].

Now we can illustrate the function of selectors in phonological rules. Consider the case of English regular plural formation (–s), where the voicing of the suffix segment agrees with that of the immediately preceding segment, unless it is a coronal fricative (in which case there must be an intervening vowel). Suppose we introduce the variables `S1 S2 : Root`, where S1 is the stem-final segment and S2 is the suffix. The rule must also be able to access the coronal node of S1. Making use of the selectors, this is simply `S2 !oral !tongue !coronal` (a notation reminiscent of paths in feature logic, (Kasper & Rounds 1986)). The rule must test whether this coronal node contains a fricative specification. This necessitates an extension to our specification, which is described below.

Browman & Goldstein (1989, 234ff) define 'constriction degree percolation', based on what they call 'tube geometry'. The vocal tract can be viewed as an interconnected set of tubes, and the articulators correspond to valves which have a number of settings ranging from fully open to fully closed. As already mentioned, these settings are called constriction degrees (`!cds`), where fully closed is the maximal constriction and fully open is the minimal constriction.

The net constriction degree of the oral cavity may be expressed as the maximum of the constriction degrees of the lips, tongue tip and tongue body. The net constriction degree of the oral and nasal cavities together is simply the minimum of the two component constriction degrees. To recast this in the present framework is straightforward. However, we need to first define the operations `max` and `min` over pairs of natural numbers:

(13)
```
obj MINMAX
    is protecting NAT .
    ops min max : Nat Nat -> Nat .
    vars M N : Nat .
    eq min(M,N) = if M <= N then M else N fi .
    eq max(M,N) = if M >= N then M else N fi .
    endo
```

(14)
```
obj CD is
    extending FEATS + MINMAX .
    op _!cd : Gesture -> Nat .
    ops clo crit narrow
            mid wide obs open : Gesture -> Bool .
    var G : Gesture .
    var N N1 N2 : Nat .
    vars O : Oral .
    vars T : Tongue .
    eq N !cd = N .
    eq {N1,N2} !cd = max(N1,N2) .
    eq {T,N} !cd = max(T !cd,N) .
    eq {N1,N2,O} !cd = max(N1,min(N2,O !cd)) .
    eq clo(G) = G !cd == 4 .
    eq crit(G) = G !cd == 3 .
    eq narrow(G) = G !cd == 2 .
    eq mid(G) = G !cd == 1 .
    eq wide(G) = G !cd == 0 .
    eq obs(G) = G !cd > 2 .
    eq open(G) = G !cd < 3 .
    endo
```

The specification `CD` allows classification into five basic constriction degrees (`clo`, `crit`, `narrow`, `mid`, and `wide`) by means of corresponding one-place predicates, i.e. boolean-valued operations over gestures. For example, the fifth equation above states that `G` has the constriction degree `clo` (i.e. `clo(G)` is true) if and only if `G !cd == 4`.

The working of these predicates is illustrated below:

(15)
```
{3,0,{{4,1},1}} !oral !tongue !cd .
    ~> 4
{3,0,{{4,1},1}} !oral !cd .
    ~> 4
{3,0,{{4,1},1}} !cd .
    ~> 3
mid({3,0,{{4,1},1}} !oral !labial) .
    ~> true
wide({3,0,{{4,1},1}} !oral !labial) .
    ~> false
open({3,0,{{4,1},1}} !oral !labial) .
    ~> true
clo({3,0,{{4,1},1}} !oral !tongue) .
    ~> true
```

References

Beierle, C. & U. Pletat (1988). Feature Graphs and Abstract Data Types: A Unifying Approach. *Proceedings of the 12th International Conference on Computational Linguistics*, pp40–45, Budapest, Hungary.

Beierle, C. & U. Pletat (1988). The Algebra of Feature Graph Specifications. IWBS Report 94, IBM TR-80.89-029, IBM Germany, Institute for Knowledge Based Systems, Stuttgart.

Beierle, C., U. Pletat & H. Uszkoreit (1988). An Algebraic Characterization of STUF.

LILOG Report 40, IBM Germany, Stuttgart.

Bird, S. (1990). *Constraint-Based Phonology*. PhD Thesis. University of Edinburgh.

Bird, S. & E. Klein (1990). Phonological events. *Journal of Linguistics*, *26*, 33–56.

Browman, C. & L. Goldstein (1989). Articulatory gestures as phonological units. *Phonology*, *6*, 201–251.

Cardelli, L. (1988) A Semantics of Multiple Inheritance. *Information and Computation*. *76*, 138–164.

Clements, G.N. (1985) The Geometry of Phonological Features. *Phonology Yearbook*. *2*. 225–252.

Dörre, J. & A. Eisele (1991). A Comprehensive Unification-Based Grammar Formalism. Deliverable R3.1.B. DYANA-ESPRIT Basic Research Action BR3175, January 1991.

Ehrig, H. & B. Mahr (1985) *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, Berlin: Springer Verlag.

Goguen, J.A., & T. Winkler (1988) 'Introducing OBJ3'. Technical Report SRI-CSL-88-9, SRI International, Computer Science Laboratory, Menlo Park, CA.

Goguen, J.A., J.W. Thatcher and E.G. Wagner (1976) 'An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types'. In R. Yeh (ed.) *Current Trends in Programming Methodology IV: Data Structuring*. pp80–144. Englewood Cliffs, NJ : Prentice Hall.

Hyman, L. M. (1984). On the weightlessness of syllable onsets. In Brugman & Macaulay (eds.) *Proceedings of the Tenth Annual Meeting of the Berkeley Linguistics Society*. University of California, Berkeley.

Kasper, R. & W. Rounds (1986). A Logical Semantics for Feature Structures. *Proceedings of the 24th Annual Meeting of the ACL*, Columbia University, New York, NY, 1986, pp257–265.

Klein, E. (1991). Phonological Data Types. In Klein, E. and F. Veltman (eds) *The Dynamics of Interpretation: Proceedings of a Symposium on Natural Language and Speech, Brussels, November 26/27, 1991*. Springer Verlag.

Newton, M. (in preparation). *Grammars and Specification Languages*. PhD Thesis, Centre for Cognitive Science, University of Edinburgh.

Reape, M. (1991). Foundations of Unification-Based Grammar Formalism. Deliverable R3.2.A. DYANA ESPRIT Basic Research Action BR3175, July 1991.

Rounds, W. & A. Manaster-Ramer (1987). A Logical Version of Functional Grammar. *Proceedings of 25th Annual Meeting of the Association for Computational Linguistics*, 6–9 July 1987, Stanford University, Stanford, CA, 89–96.

Sagey, E. (1986). *The Representation of Features and Relations in Non-Linear Phonology*. PhD Thesis, MIT, Cambridge, Mass.

Sannella D. & A. Tarlecki (1987) Some thoughts on algebraic specification. LFCS Report Series ECS-LFCS-87-21, Laboratory for Foundations of Computer Science, University of Edinburgh.

Shieber, S. (1986). *An Introduction to Unification-Based Approaches to Grammmar*. CSLI Lecture Note Series, University of Chicago Press, Chicago.

Smolka, G. and H. Aït-Kaci (1989) 'Inheritance Hierarchies: Semantics and Unification'. *Journal of Symbolic Computation*, 7, 343–370.