Phonological Data Types[⋆]

Ewan Klein

Centre for Cognitive Science, University of Edinburgh, 2 Buccleuch Place, Edinburgh EH8 9LW, Scotland

Table of Contents

1 Introduction

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.

One of the dominant paradigms in current computational linguistics is provided by unification-based grammar formalisms. Such formalisms (cf. [Shi86, KR86]) describe hierarchical feature structures, which in many ways would be appear to be an ideal setting for formal phonological analyses. Feature bundles

⋆ The work reported in this paper has been carried out as part of the research programmes of the DYANA project (BR 3175), funded by CEC ESPRIT, and of the Human Communication Research Centre, supported by the UK Economic and Social Research Council. It is based on a paper, written jointly with Steven Bird, presented to the Twelfth International Congress of Phonetic Sciences, Aix-en-Provence, August 1991, and I am indebted to Steven for many of the ideas discussed here. I would also like to thank Michael Newton and Jo Calder for comments on earlier versions.

have long been used by phonologists, and more recent work on so-called feature geometry (e.g. [Cle85, Sag86]) has introduced hierarchy into such representations.

However, in their raw form, feature terms (i.e., formalisms for describing feature structures) do not always provide a perspicuous format for representing structure. Compare, for example, the 'dotted pair' representation of a list of syllables with the feature-based one (where '$\sqcap$' represents feature term conjunction):

(1) $(\sigma_1 . \sigma_2 . \text{ nil})$

(2) $\textsc{first} : \sigma_1 \sqcap \textsc{last} : (\textsc{first} : \sigma_2 \sqcup \textsc{last} : \mathit{nil})$

The term in 2 is standardly taken to be satisfied by a feature structure of the sort shown in (3):

(3) $$\begin{bmatrix} \text{FIRST} & \sigma_1 \\ \text{LAST} & \begin{bmatrix} \text{FIRST} & \sigma_2 \\ \text{LAST} & \mathit{nil} \end{bmatrix} \end{bmatrix}$$

In standard approaches to data structures, complex data types are built up from atomic types by means of **constructor functions**. For example, `_._` (where we use the underscore '_' 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`.

It can be seen that the feature-based encoding of lists uses only selectors for the data type; i.e. the feature labels FIRST and LAST in 3. However, the `_._` constructor of (1) 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 awkward when the constructors are not made explicit.

For computational phonology, it is not such an unlikely scenario to be confronted with 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.

This problem has to a certain extent already been encountered in the context of syntactic analysis, and in response various proposals have been made to enrich raw feature term formalisms with recursive type (or sort) specifications ([RM-R87, DE91]) or relational and functional dependencies [Rea91] so as to allow a more transparent encoding of data types. By virtue of their expressiveness, these enrichments typically render the resulting formalisms undecidable. Thus, some care has to be taken to ensure that a given encoding does not introduce computational intractability into the grammar.

As hinted above, in the longer term, it would be sensible to embed phonological analyses within a broader formalism for grammar processing, and enriched feature formalisms of the kind alluded to above seem provide an appropriate setting. However from a heuristic point of view, there seems to be some virtue in being able to explore the complexities of phonological structure without being overly concerned about this embedding into a feature-based formalism. The alternative which we been exploring here treats phonological concepts as abstract data types. A particularly convenient development environment is provided by the language OBJ ([GW88]), which is based on order sorted equational logic. The denotational semantics of an OBJ module is an algebra, while its operational semantics is based on order sorted rewriting.

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. [GTW78]) and much subsequent work (cf. [EM85]) 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

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.

we list all the variables (`var`) which figure in them. As an illustration, we give below a specification of the data type `LIST1`.

(4)
```
obj LIST1 is sorts Elt List  .
  ops x y : -> Elt .
  op nil : -> List .
  op _._ : Elt List -> List .
  op head_ : List -> Elt .
  op tail_ : List -> List .
  var X : Elt .
  var L : List .
  eq head(X . L) = X .
  eq tail(X . L) = L .
endo
```

Th 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 `ops x y : -> Elt` means that `x, y` are constants of sort `Elt`.

Although we have specified (4 as a type of lists of elements `x, y`, this is obviously rather limiting. 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. We will see how this can be done in the next section.

1.2 Inheritance

We have briefly examined the idea that data can be structured in terms of sorts and operations on items of specific sorts. Another approach is to organise data into a hierarchy of classes and subclasses, where data at one level in the hierarchy inherits all the attributes of data higher up in the hierarchy. Inheritance hierarchies provide a succinct and attractive method for expressing a wide variety of linguistic generalizations. Suppose, for example, that we adopt the claim that all syllables have *CV* onsets. 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.

In order to deal with inheritance, we need to generalise the many-sorted specification language to an order sorted language by introducing a subsort relation.[3] Thus, 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 See [Car88] for a general discussion of inheritance between record structures in programming languages, and [SA89] for an account of inheritance within the framework of order sorted equational logic.

(5)
```
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`.

In general, let $\sigma$and σ'be sorts such that $\sigma' < \sigma$, let f be an operator of rank $\sigma \rightarrow \tau$, and let t be a term of sort σ'. Then f is defined not just for terms of sort σ, but also for t of subsort σ', and $f(t)$ is a term of sort τ. From a semantic point of view, we are saying that if a function assigns values to members of particular set X, then it will also assign values to members of any subset X' of X.

Returning to lists, the specification in (6) (due to [GW88]) introduces `elt` and `nelist` (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 list of `X`s, where the parameter `X` can be instantiated to any module which satisfies the condition `TRIV`; the latter is what [GW88] 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`.

(6)
```
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 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)`?

2 More Examples: Metrical Trees

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

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)
```
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
```

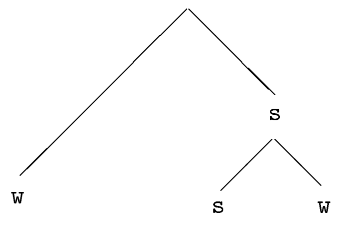
We can now instantiate the formal parameters of the module `BINTREE[NONTERM TERM :: TRIV]` with input modules which supply appropriate sets of nonterminal and terminal symbols. Let us use uppercase quoted identifiers (elements of the OBJ module `QID`) for nonterminals, and lower case for terminals. The specification in ([**?**]) 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. ([**?**]) illustrates the instantiation of formal parameters by an actual module, namely `QID`.

(8)
```
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.

(9)
```
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).

(10)


In addition, all trees have a distinguished leaf node called the 'highest terminal element', 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:

```
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]

(11)
```
obj METTREE is
extending BINTREE[MET,QID] * (sort Id to Leaf) .
op hte_ : Tree -> Leaf .
var L : Leaf .
vars T1 T2 : Tree .
vars X : Label .
eq hte ( X [ L ] ) = L .
ceq hte ( X [ T1 , T2 ]) = hte T1 if label T1 == s .
ceq hte ( X [ T1 , T2 ]) = hte T2 if label T2 == s .
endo
```

These allows reductions of the following kind:

(12)
```
hte(s['a]) .                       ⇝ 'a
label(right (s[s['a],w['b]])) .    ⇝ w
hte (s[s['a],w['b]]) .             ⇝ 'a
hte (s[s[w['a],s['b]],w['c]]) .    ⇝ 'b
```

The specification `METTREE` 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. However, this seems to be a weakness of the original formulation of metrical trees, and we will not elaborate here on various solutions that come to mind.

3 Feature Geometry

The particular feature geometry we shall specify here is based on the articulatory structure defined in [BC89].[6]. The five active articulators are grouped into a

5 The `*` construct tells us that the principal sort of `QID`, namely `Id`, is mapped to the sort `Leaf` in `METTREE`. The `==` is a built-in polymorphic equality operation in OBJ.

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 as its phonological role is somewhat dubious.

hierarchical structure involving a tongue node and an oral node, as shown in the following diagram.

This structure is specified below. The nine sorts and the first three operations describe the desired tree structure, using an approach which should be familiar by now. However, in contrast with our previous specifications, this specification permits ternary branching: the third constructor takes something of sort `glottal` and something of sort `velic` and combines them with something of sort `oral` to build an object of sort `root`.

```
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
```

```
     3,4,4,1,1 oral . ⇝                     4,1,1
(13) 3,4,4,1,1 oral tongue . ⇝               4,1
     3,4,4,1,1 oral tongue coronal . ⇝      4
```

The selectors (e.g. `coronal`) occupy most of the above specification. Notice how each selector mentioned in the `ops` section appears again in the `eqs` section. Consider the `coronal` selector. Its `ops` specification states that it is a function defined on objects of sort `tongue` which returns something of sort `coronal`. The corresponding equation states that $\langle C, D\rangle$ `coronal` $= C$. Now C has the sort `coronal` and D has the sort `dorsal`. By the definition of the first constructor, $\langle C, D\rangle$ has the sort `tongue`. Furthermore, by the definition of the `coronal` selector, $\langle C, D\rangle$ `coronal` has the sort `coronal`. So the equation $\langle C, D\rangle$ `coronal` $= C$ respects the sort definitions.

Selectors can be used to implement structure-sharing (or re-entrancy). Suppose that two segments $\mathtt{S}_1$ and $\mathtt{S}_2$ share a voicing specification. We can write this as follows: $\mathtt{S}_1$ `glottal` $=$ $\mathtt{S}_2$ `glottal` . This structure sharing is consistent with one of the main motivating factors behind autosegmental phonology, namely, the undesirability of rules such as $[\alpha \text{ voice}] \rightarrow [\alpha \text{ nasal}]$. The equation S `glottal` $= S$ `velic` is illsorted.

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 $\mathtt{S}_1, \mathtt{S}_2$: `root`, where $\mathtt{S}_1$ is the stem-final segment and $\mathtt{S}_2$ is the suffix. The rule must be able to access the coronal node of $\mathtt{S}_1$. Making use of the selectors, this is simply $\mathtt{S}_1$`oral tongue coronal` (a notation reminiscent of paths in feature logic, [10]). The rule must test whether this coronal node contains a fricative specification. This necessitates an extension to our specification, which will now be described.

Browman & Goldstein [4:234ff] 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. These settings will be 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 we employ our notion of percolation again. The definition of `max` and `min` are as follows:

```
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
```

```
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
```

```
     3,0,4,1,1 oral tongue cd .    ⇝ 4
     3,0,4,1,1 oral cd .           ⇝ 4
     3,0,4,1,1 cd .                ⇝ 3
(14) 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
```

There are five basic constriction degrees (`clo`, `crit`, `narrow`, `mid`, and `wide`), and these are grouped into two sorts `obs` and `open`.

Using the above extension, the condition on the English voicing assimilation rule could be expressed as follows[7], where *Crit* : `crit`:

7 A proviso is necessary here. Just because there is a critical CD at the tongue tip

$\mathtt{S}_1$ `oral tongue coronal cd` $\neq$ `crit`

If this condition is met, the effect of the rule would be:

$\mathtt{S}_1$ `glottal cd` $=$ $\mathtt{S}_2$ `glottal cd`

This is how we say that $\mathtt{S}_1$ and $\mathtt{S}_2$ have the same voicing.

Now the manner features can be expressed as follows (omitting `strident` and `lateral`).

```
obj MANNER is
protecting CD .
ops son cont cons nas : Root -> Bool .
var R : Root .
eq son(R) = open(R) .
eq cont(R) = clo(R oral) == false .
eq cons(R) = obs(R oral) .
eq nas(R) = open(R velic) and obs(R oral).
endo
```

(15)
```
son(3,0,4,1,1) .  ⤳ false
cont(3,0,4,1,1) . ⤳ false
cons(3,0,4,1,1) . ⤳ true
nas(3,0,4,1,1) .  ⤳ true
```

It follows directly from the above definitions that the collection of noncontinuants is a subset of the set of consonants (since `clo` $<$ `obs`). Similarly, the collection of nasals is a subset of the set of consonants. Note also that these definitions permit manner specification independently of place specification, which is often important in phonological description.

4 Conclusions

We began this article by pointing out the difficulty of defining and using complex phonological structures. In addressing this problem we have used a strategy from computer science known as abstract specification. We believe this brings us a step further towards our goal of developing a computational phonology.

This approach contrasts with the finite state approach to computational phonology [1,6]. Finite state grammars have employed a rigid format for expressing phonological information, and have not hitherto been able to represent the complex hierarchical structures that phonologists are interested in. Our approach has been to view phonological structures as abstract data types, and to obtain a rich variety of methods for structuring those objects and for expressing constraints on their behaviour.

does not mean that a fricative is being produced. For example, the lips might be closed. We can get around this problem with the use of CD percolation (as already defined) and the equation $\mathtt{S}_1$`oral` $=$ `crit`. Further discussion of this option may be found in [2].

We have briefly examined the idea that data can be structured in terms of sorts and operations on items of specific sorts. We also explored the organization of data into a hierarchy of classes and subclasses, where data at one level in the hierarchy inherits all the attributes of data higher up in the hierarchy. Inheritance hierarchies provide a succinct and attractive method for expressing a wide variety of linguistic generalizations. A useful extension would be to incorporate **default inheritance** into this system.

Further exploration of these proposals, we believe, will ultimately enable the mechanical testing of predictions made by phonological systems and the incorporation of phonological components into existing computational grammars.

5 References

References

[SA89] Antworth, E. L. (1990). *P*C-KIMMO: A Two-level Processor for Morphological Analysis. Dallas: Summer Institute of Linguistics.

[BP88] Beierle, C. & U. Pletat (1988). Feature Graphs and Abstract Data Types: A Unifying Approach. Proceedings of COLING '88

[Bir90] Bird, S. (1990). *Constraint-Based Phonology.* PhD Thesis. University of Edinburgh.

[BK90] Bird, S. & E. Klein (1990). Phonological events. *Journal of Linguistics*, *26*, 33–56.

[BC89] Browman, C. & L. Goldstein (1989). Articulatory gestures as phonological units. *Phonology*, *6*, 201–251.

[Car88] Cardelli, L. (1988) A Semantics of Multiple Inheritance. *Information and Computation* 76, pp138–164.

[Cle85] Clements, G.N.. (1985) The Geometry of Phonological Features. *Phonology Yearbook* 2, pp225–252.

[DKK*87] Dalrymple, M., R. Kaplan, L. Karttunen, K. Koskenniemi, S. Shaio & M. Wescoat (1987). *Tools for Morphological Analysis.* CSLI–87–108. CSLI, Stanford.

[DE91] Dörre, J. & A. Eisele (1991). A Comprehensive Unification-Based Grammar Formalism. Deliverable R3.1.B, DYANA—ESPRIT Basic Research Action BR3175, January 1991.

[EM85] Ehrig, H. & B. Mahr (1985) *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, Berlin: Springer Verlag.

[GW88] Goguen, J.A., & T. Winkler (1988) 'Introducing OBJ3'. Technical Report SRI-CSL-88-9, SRI International, Computer Science Laboratory, Menlo Park, CA.

[GTW78] 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.

[KR86] 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.

[Rea91] Reape, M. (1991). Foundations of Unification-Based Grammar Formalism. Deliverable R3.2.A, DYANA—ESPRIT Basic Research Action BR3175, July 1991.

[RM-R87] 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, pp96.

[Sag86] Sagey, E. (1986). *The Representation of Features and Relations in Non-Linear Phonology*. PhD Thesis, MIT, Cambridge, Mass.

[Shi86] Shieber, S. (1986). *An Introduction to Unification-Based Approaches to Grammmar*. CSLI Lecture Note Series, University of Chicago Press, Chicago.

[SA89] Smolka, G. and H. Ait-Kaci (1989) 'Inheritance Hierarchies: Semantics and Unification'. *Journal of Symbolic Computation* 7, pp343–370.

This article was processed using the LaTeX macro package with ICM style