Abstract Specification of Grammar

Michael Newton

Ph.D.
University of Edinburgh
1992

I declare that this thesis has been composed by me and that the work reported here is original except where acknowledged otherwise.

Michael Newton
July 23, 1992

I would primarily like to thank my excellent and conscientious supervisors, Ewan Klein and Don Sannella. I would also like to thank Ingemarie Bethke and Mike Reape, who also acted as supervisors (official and de facto) during the early stages of this work. Thanks for forbearance on the part of those unfortunate enough to have shared an office with me, particularly Lex Holt, TeXie extraordinaire. General thanks to all around the Centre who have taken the time and trouble to point out to me the errors of my various ways. Thankyou to Edinburgh University, the British Council and the Commonwealth Universities Commission, for feeding me for three years. Lastly, thanks to my friends in Edinburgh and elsewhere (including family!) for keeping me sane, and particularly to Ms Deirdre King of Cork City, the realest person I've ever known.

Table of Contents

Chapter 1

Introduction

1.1 The Problem of Abstraction

Science is concerned with modelling observable phenomena. The task of grammar is to model human language. There are many steps and aspects to be considered for any modelling task, including fixing an interpretative framework, working out a specific syntax sufficient for the task, isolating conditions in this syntax which all models should satisfy, demonstrating (preferably constructively) the existence of such models, describing a system of deduction for the system and showing this to be (at least) sound. In a simple task many of these steps may be trivial or void; in a more complex case one can rapidly be overwhelmed with detail. A level of informality that may cause no problem when dealing with a simple task may hide all sorts of errors and omissions in a more complex one. I would argue that an important step in avoiding such problems, in grammar as elsewhere, is to employ a (preferably formal) language of description which explicitly reflects the many distinct tasks and levels, and their interaction. The theory of algebraic specification languages offers the promise of a single formal setting in which to consider all of the above steps. In this thesis, I aim to demonstrate that, in the investigation of grammar, good use can be made of some of the central ideas in specification, including refinement, modularisation, parameterisation, and explicit

levels of abstraction. Primarily, I want to illustrate the importance of the distinction between abstract descriptions and concrete representations, and the interplay between them.

There are many grammar-writing frameworks in which one can define models for the syntax of natural language: GB, LFG, GPSG, HPSG, various categorial frameworks, and so on. Although they approach their tasks in different ways and at different levels of formality, they all employ some notional domain, whose objects (most typically derived from tree diagrams) are meant to correspond to utterances, with particular features of the objects corresponding to particular observable features of such utterances. In model theory, we devise a syntax to describe these features (a *signature*), and try to formally describe both the domain of interpretation, and the correspondence between the syntax and the elements of that domain. Thus the objects are said to *model* the signature (in the formal, model-theoretic sense), and the precise correspondence between the utterances and the domain of interpretation is called a *model.* Sometimes, less formally, the domain itself is referred to as a model.

Very often a linguist seeking to account for (part of) a particular language will also find her work suggesting extensions or refinements to the defining framework in which her work is cast. In practice, working either on or within a formalism may "feel" very much like programming (perhaps the former is "system programming"), and this is very far from coincidence. Indeed very often it is programming: a model which can be demonstrated running on a computer is at least a step toward psychological plausibility. In algebraic specification, a program is considered as an algebra, and an algebra is just a certain kind of model. We consider a program purely in terms of its input/output behaviour, thus abstracting away from the procedural details of just what algorithm is used to produce this behaviour.

Programming, according to one view, is a discipline precisely concerned with the breaking down of modelling tasks into subtasks trivial to the point of being routine. In programming as well as grammar writing one most often has to consider a vast, if not infinite, range of data. This very often makes real "proof" that what you have is a correct model impossible, but nevertheless some effort has

been put in by computer scientists to move toward this goal. Some of this work might profitably be considered for use in the process of producing natural language grammars. At least one advantage might be to give a rigorous and unified way of working on frameworks. It may also lead ultimately to more meaningful inter-framework comparison, and perhaps even to eliminating altogether the need for distinct frameworks: such frameworks invariably incorporate a range of individual hypotheses and arbitrary design decisions, which, in a less parochial setting, could more easily be considered individually on their merits for incorporation or rejection.

It is interesting to note that much of theoretical computer science has been involved with moving *away* from the imperative notion of computation, toward declarative characterisation of information systems (Backus 1978), and to some extent, this spirit has carried over to the computational linguistics community. One motivation for this trend has been the provision of abstract specifications for programs (or modules), against which putative implementations could be measured, allowing implementations to be freely interchanged within a larger system. So one notion of specification has been to give input and output conditions required of any program which purports to model the specified task. If the effect of every construct in a programming language can be specified in terms of pre- and post-conditions, one might then proceed to show of a particular program whether, given the input conditions, the output conditions hold. To produce such a "proof" for an existing program can be a very difficult task, but an alternative is to attempt to integrate the production of the proof with that of the program, or even make it the driving force (Gries 1981). Another tack is the attempt to produce *declarative* programming languages, such as (pure) Prolog, in which the program can be viewed as describing, in logical terms, how input and output are related, but the interpreter imposes a particular notion of execution upon this description. These approaches might be said to meet in so-called *wide-spectrum languages* (Sannella 1986). Such a language can be viewed as an enrichment of an executable (probably declarative) language by non-executable forms of description, in order to make easier the initial description of required behaviour. Such a

specification will admit many eventual implementations — if it is *loose* they need not even all be isomorphic. It also comes equipped with a notion of *refinement*, by which another, very similar, specification is shown to admit no models not admitted by the first. By a sequence of such refinements, embodying a sequence of design decisions, we may eventually move to an executable specification, which we know to satisfy the original specification also. The term "wide-spectrum" comes from this ability to express all stages in the spectrum from very loose, abstract specification, to executable program.

Experience suggests that the question of which comes first, specification or program (module), is not a particularly enlightening one. The point is simply that the specification says what we think is important about the program. Then the program gives us a tool for re-examining that notion of what is important: to what extent does the program capture the behaviour we expected and what other behaviour would we like to see, does this behaviour (or lack of it) arise from the specification, or merely from this implementation of it, and how can the specification be changed to reflect our newly acquired knowledge of what really is important? That the "program" be running on a computer is not really central to this process: we see the same process at work whenever we choose to examine a particular model in order to gain insight into a wider problem. To put it in terms of stepwise refinement, this amounts to saying

1. we don't always necessarily make the *right* design decisions at each stage (sometimes we have to backtrack) and

2. sometimes it can be useful, in order to investigate the design we have so far, to make a refinement that goes all the way to executability, even if the refinements that get us there are rather dubious or unmotivated (prototyping!).

So abstract specification *starts* from the need for powerful modularisation and parameterisation facilities (data types, modules, etc.), and on top of that requires a vocabulary for specifying what is important about a model. There are two central ideas behind this. One is the ability to make statements which, while

not by themselves admitting of interpretation as programs, can be seen as giving partial or incomplete descriptions of programs, and thus can also be given a precise semantics, namely the *class* of models to which they apply. The second is to have a sufficiently powerful technique for expressing levels of abstraction. Very often it is difficult or impossible to describe some required behaviour without giving an example of how it is to be achieved, for example, using auxiliary operations, even though there may be many other ways of achieving the same result. We need to be able to indicate that these details are unimportant: what is important is the result. It is tempting to suppose that this second is *all* you need: here's a program, here's the level of abstraction at which it is to be considered. I think where this fails is in not allowing you to capture every stage of the development process explicitly: in particular, there would be no way to say what is importantly *wrong* about a program: what you fail to mention at all is not necessarily wrong, just unimportant (as far as you know anyway).

This should be quite a familiar phenomenon from the construction of grammar fragments. In a simple rewrite grammar R a statement like "S $\longrightarrow$ NP VP" will be interpreted as saying that a noun phrase and a verb phrase can be put together to form a sentence. If we decide that this is an important abstract feature of grammars of English, we can form from R, using an abstraction operation, a specification which requires this feature, without saying anything about any other details of implementation which may have occurred in R. If in investigating such a system, however, we realise that it is not adequate to deal with agreement between subject and verb (we may have *I eat* but not *I eats*), we can then form an abstract specification which insists that subject and verb agree, but which need not spell out the full details of a grammatical fragment, down to the last lexical entry. Thus we have a method of expressing what we learnt from the simple (but wrong) implementation. Having discovered that the failure to mention agreement *is* important, we can deal with it in a way that does not depend on any particular implementation.

The aim of this thesis is to produce some specifications which precisely and plausibly (if not completely) describe some taxonomy and postulates used in mod-

ern linguistics. For instance, one might begin with the specification of some gross phenomenon such as constituent structure, one refinement of which could be via a construction utilising a combination of immediate dominance structure and linear precedence ordering (the IDLP format of GPSG (Gazdar et al. 1985)). This is a refinement precisely because it narrows the class of models. Specifications supply a vocabulary, and restrictions on the interpretation of that vocabulary.

If we aim to produce a specification of the range of natural language grammars (*universal grammar*), we can employ the discipline of *stepwise refinement*, whereby a series of specifications is produced, each more constrained than the last, gradually winnowing away grammars which never occur in natural language, with each constraining step representing some claim about universal grammar. In order to make such a claim (e.g. "all natural languages have an IDLP presentation" — c.f. Section 3.4), we need to be able to relate these vocabularies one to another. If UG is to define the range of natural languages, and IDLP defines grammars with an IDLP presentation, we require $\mathsf{UG} \sqsubseteq \mathsf{IDLP}$ (UG refines IDLP). However, a specification X such that $\mathsf{X} \sqsubseteq \mathsf{Y}$ can also be used to describe some subproblem of the problem addressed in Y. Thus we can use specifications to set up taxonomies in which a phenomenon can be described, and refinement either to make claims about "universal grammar", or to focus on particular languages or implementation techniques. I hope that specifications from different grammatical frameworks will turn out to involve subspecifications amenable to generalisations which will concretely identify the same idea across different frameworks.

1.2 Programs and Specifications

The first use of logical axioms in the specification of programs was in so-called *Floyd-Hoare assertions*. Gries (1981) represents perhaps the apogee of this style. This system is designed for use with traditional *imperative* programming languages, in which program statements are primarily step-by-step instructions to the machine, rather than just definitions or descriptions. Such languages include Fortran,

Algol, and their many descendants. In this system, the specification of a program consists of two logical statements (or *assertions*), the *precondition*, giving the properties expected of the program's input variables, and the *postcondition*, giving the required properties of its output variables. Associated with each type of program command is a *rule of deduction*, using which one may, from the state of the variables before the execution of such a command, deduce their state thereafter. We view the statements of the program as subprograms. For each of these subprograms, we prove that certain input conditions imply certain output conditions. We then compose these subprograms, with the output conditions of one statement becoming the input conditions of the next. Thus a program can be proved to be *correct* with respect to a specification, by showing that if the precondition is true before execution of the program, the postcondition must be true afterward. The idea here was that a specification provides a standard, against which different implementations can be measured.

Gries (1981) describes a discipline in which proof and program are developed hand-in-hand, starting from a Floyd-Hoare specification. The driving force is the breakdown of the proof task into smaller and smaller subproofs, with the choice of how to make that breakdown governed by the available rules of inference. The construction of the program becomes no more than a record of the rules used at each step of the proof, that is, a syntactic representation of the proof. This identification of proof and program is a familiar one in constructive type theory (Martin-Löf 1982).

On another tack, the automation of proof construction has long been a central concern in Artificial Intelligence. If the particular inference rules used in a theorem prover could be viewed as the operations of a programming language, then the theorem prover could in turn be viewed a system which automatically produces programs from specifications. Since (theoretically!) only the theorem prover need know about these inference rules, the precise sequence of instantiated rules used (which constitute the program for Gries or Martin-Löf) becomes irrelevant to the user. Since the specification itself can be directly executed, we may as well say that it *is* the program. This is the perspective taken in the discipline of

logic programming (in which a program consists purely of logical statements). In particular, viewing the deduction rule of Robinson (1965) (called *resolution*) as a programming construct gives us the programming language *Prolog* (Colmerauer et al. 1973, Kowalski 1974), the dominant language in logic programming, and one which has also found widespread acceptance in the computational linguistics community.

We are now in a position to write down a few logical axioms, and have Prolog interpret these as a program. What more could we ask? As anyone who has done very much work in the language will know, the answer is "quite a bit". I will only consider its deficiencies as a specification language.

1. Limitation to the Horn clause fragment of predicate logic means one cannot write specifications which use axioms like $\forall v \,.\, \texttt{finite}(v) \rightarrow (\texttt{present}(v) \vee \texttt{past}(v))$ or $\forall xy \,.\, x + y = y + x$, yet such statements are often the clearest way of expressing a concept. It would be useful to have a formal language in which specifications using such axioms have a precise meaning, and can be explicitly related to programs (in Prolog or anything else) which implement them.

2. It is quite possible to write a logically complete description of a system in the Prolog language which nevertheless fails as a program because of the particular control strategy employed by the Prolog interpreter. For instance, it may loop indefinitely without ever finding an answer. It would be useful if we could say that this was correct as a specification (that is, a complete description of the system being modelled), but not as an implementation, and to have a formal language in which it has a precise meaning, which makes explicit its relationship to any reformulation (possibly even in a different language) which does run correctly.

3. Because the semantics of Prolog is tied to the *initial* model of a description (more of this below), there is no way to make loose, or partial, specifications, which describe some features of a system, but make no claim to be complete.

> In a formal specification language which makes explicit the relationship between such specifications and implementations of them, more and more such specifications can be gradually (and explicitly) accreted as investigation of the system proceeds, until the description is complete.

The key to solving the second problem is the use of a richer system of specification, which contains the programming language (or languages) as a subset. For the first problem, it will also be necessary to abandon the restriction to predicative Horn clauses. For most of this thesis, I will be content to allow myself arbitrary first order sentences, possibly involving sorts, equality and partial operations. Later, in Chapter 4, we will see some machinery for combining specifications written in different logics. In order to address the third problem, we need to be able to produce (at least) two different specifications from the same axioms, one of which is used to refer to the class of *all* models which satisfy the axioms, and the other of which is used to refer to a particular model (the term model) or subclass of models (the initial models). It is time that I introduced the algebraic concepts on which such specifications will be based.

1.3 Universal Algebra

In mathematics, it is common to characterise a system in terms of a few simple statements, or *axioms*. Perhaps the prototypical example is Euclid's axiomatisation of geometry. At that time, statements in natural language were used, but today, it is probable that some formal language, such as predicate logic, might be employed. In Universal Algebra (Burris & Sankappanavar, 1981), a class of *unrestricted algebras* is presented in the abstract by giving a set name A and a number of *operation* or *function* symbols upon that set. These names are said to form the *signature* of the class of algebras. An algebra of that class is simply a model interpreting the syntax supplied by the signature. That is, it must provide an actual set (*carrier*) to correspond to the name A, and functions on that set to correspond to each operation symbol.

More generally, to present any abstract algebra, we simply add to the above *equations* (also known as *sentences*) which equate terms built from the operators, and possibly some variables. These equations are interpreted in a particular algebra of that class by insisting that for any instantiation of the variables, the object which interprets the left hand side term is the same as that which interprets the right (so all equations are effectively universally quantified). The algebras for which this holds are said to *satisfy* the equations.

For instance, we might specify a set name A, a *nullary* operator (constant) e, and an infix binary operator $\cdot$ and equations

$$x \cdot \mathsf{e} = \mathsf{e} \cdot x = x \tag{1.1}$$
$$(x \cdot y) \cdot z = x \cdot (y \cdot z)$$

This defines the *monoids*, the most general interpretation of which is to suppose A consists of strings over some alphabet, $\cdot$ is concatenation, and e is the empty string. But there are many other interpretations: for instance the integers form a monoid if we interpret $\cdot$ as addition, and e as zero.

One way we might generalise this is to allow other forms of sentence. For instance, instead of only allowing sentences which are implicitly universally quantified, we might make this quantification explicit and allow existential quantification too. Then in the above we need not have supplied the e operator, but could instead have replaced the sentences there by

$$\exists i \forall x (x \cdot i = i \cdot x = x) \tag{1.2}$$
$$\forall x \forall y \forall z ((x \cdot y) \cdot z = x \cdot (y \cdot z))$$

1.4 First Order Logic

First order formulae can be defined from a set of atomic formulae by closure under the following rules.

1. If ϕ is a first order formula, then $\neg\phi$ ("not ϕ") is a first order formula.

2. If ϕ and ψ are first order formulae, $\phi \wedge \psi$ ("ϕ and ψ"), is a first order formula.

3. If ϕ is a first order formula, and x is a variable, then $\forall x \,.\, \phi$ ("for every x, ϕ"), is a first order formula.

We may as well suppose that there is a fixed stock of variables, $\mathcal{X}$ say (so $x \in \mathcal{X}$). Fixed syntax, like "$\neg$", "$\wedge$" and "$\forall$" , is called *logical syntax.* There are many different first order logics, corresponding to different notions of atomic formula. To each such notion, there corresponds a different definition of *signature*, which describes the variable, or *non-logical*, part of the syntax.

A *sentence* is a formula which is *closed* with respect to each variable. $\forall x.\phi$ is closed with respect to x, and with respect to any variable y with respect to which ϕ is closed; $\neg\phi$ and $\phi_1 \wedge \phi_2$ are closed with respect to x if and only if their constituent subformulae (ϕ, and ϕ_1 and ϕ_2, respectively) are closed with respect to x. An atomic formula ϕ is closed with respect to x only if x does not occur in ϕ.

In *pure predicate logic*, a signature Σ is a collection of *predicate* names, divided into subclasses Σ_i, for $i = 0, 1, 2 \ldots$. If $p \in \Sigma_i$, p is said to be of *arity* i, and for any sequence $x_1, \ldots x_i$ of variables (not necessarily distinct), $p(x_1, \ldots x_i)$ is an atomic formula. We get sentences like $\forall x \,.\, \neg(\texttt{lexical}(x) \wedge \texttt{phrasal}(x))$.

In a *(functional) predicate logic*, a signature may also include *function* (or *operation*) names of the various arities. The definition of atomic formulae is generalised by replacing the sequence $x_1, \ldots x_i$ of variables by a sequence $t_1, \ldots t_i$ of *terms*,

where a term is either a variable, or has the form $f(u_1, \ldots u_j)$, where f is an operation of arity j, and $u_1, \ldots u_j$ is a sequence of j terms. The language of Prolog is a subset of such a language. We get sentences like $\forall x \,.\, \texttt{precedes}(x, \texttt{hd}(x))$.

In a language *with equality*, we always have the binary predicate $\_ = \_$, which we may as well consider part of the fixed (logical) syntax. (The underscores, _, are meant to indicate where the arguments should be written, so $\_ = \_$ indicates an infix predicate, producing atomic formulae written $x = y$, instead of $= (x, y)$). In an *equational* logic, this is the *only* predicate. We get sentences like $\forall xy \,.\, x + y = y + x$.

An *algebra* is simply a *model* of a first order signature Σ, that is, a domain of interpretation, and a map which interprets the syntax of Σ in that domain. These define a *semantics* for the syntax of Σ. (When I refer to the semantics of a formal language in this way, I take no particular position on whether natural language "semantics" should be seen as an instance of this, as in Montague grammar, or an entirely separate endeavour, as in GB.) A Σ-model M must provide

1. a domain of interpretation, or *carrier*, $[M]$.

2. for every predicate name p of arity i in Σ, a set $[Mp] \subseteq [M]^i$, that is, a set of i-tuples drawn from $[M]$. In a language with equality, $[M =] = \{\langle x, x\rangle | x \in [M]\}$, the diagonal relation on $[M]$.

3. for every operation name f of arity i in Σ, a (total) function $[Mf] : [M]^i \to [M]$, which maps i-tuples over $[M]$ into $[M]$.[1]

Where the name of the model is obvious or unimportant I may just write $[p]$ instead of $[Mp]$, and so on. We must imagine that every variable x names a point $[x] \in [M]$. Then $f(x)$ names the point $[f(x)] = [f]([x])$, where f is an operation of arity 1. More generally, if f is an operation of arity j and the terms $t_1, \ldots t_j$ name the

---

[1]For the interpretation $[Mp]$ of a predicate name p one sometimes sees $[p]^M$, $[p]_M$, $[[p]]^M$, and various other forms besides (similarly for operation names).

points $[t_1], \ldots [t_j]$, then $f(t_1, \ldots t_j)$ names the point $[f(t_1, \ldots t_j)] = [f]([t_1], \ldots [t_j])$. M is said to satisfy an atomic formula $p(t_1, \ldots t_i)$ if $\langle [t_1], \ldots [t_i] \rangle \in [Mp]$. A model satisfies $\neg\phi$ if it does *not* satisfy ϕ. A model satisfies $\phi \wedge \psi$ if it satisfies both ϕ and ψ. A model satisfies $\forall x \,.\, \phi$ if it satisfies ϕ, *independent* of the value $[x]$ associated with x; that is, no matter what value $[x]$ takes. Thus the question of whether a model satisfies a *sentence* is also independent of the values associated with the variables. If M satisfies ϕ, write $M \models \phi$.

"$\neg$" binds most tightly, and "$\forall$" least, so for instance $\neg\phi \wedge \psi$ means $(\neg\phi) \wedge \psi$, and $\forall x \,.\, \phi \wedge \psi$ means $\forall x \,.\, (\phi \wedge \psi)$. There are various metasyntactic definitions we can make, in order to produce shorter or more easily understood sentences. We write

1. $\phi \vee \psi$ ("ϕ or ψ") for $\neg(\neg\phi \wedge \neg\psi)$,

2. $\phi \rightarrow \psi$ ("ϕ implies ψ") for $\neg\phi \vee \psi$,

3. $\phi \leftrightarrow \psi$ ("ϕ if and only if ψ") for $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$,

4. $\exists x \,.\, \phi$ ("there exists x such that ϕ") for $\neg\forall x \,.\, \neg\phi$,

5. $\forall x_1 x_2 \ldots x_n \,.\, \phi$ for $\forall x_1 \,.\, \forall x_2 \ldots \forall x_n \,.\, \phi$, and

6. $\exists x_1 x_2 \ldots x_n \,.\, \phi$ for $\exists x_1 \,.\, \exists x_2 \ldots \exists x_n \,.\, \phi$.

1.5 Initiality

We might try to use the following set of statements in predicate logic to characterise addition in successor arithmetic.

$$\begin{aligned} \mathsf{E} \;=\; & \forall x \,.\, \mathtt{p}(\mathtt{o}, x, x) \\ & \forall xyz \,.\, \mathtt{p}(x, y, z) \rightarrow \mathtt{p}(\mathtt{s}(x), y, \mathtt{s}(z)) \end{aligned}$$

The following Prolog implementation is just a syntactic variant of E.

```
p(o,X,X).
p(s(X),Y,s(Z)) :- p(X,Y,Z).
```

We have a operation of arity zero (that is, a constant) o, intended to represent zero, a unary operation s, intended to represent successor (the successor of 0 is 1, the successor of 1 is 2, and so on), and a three-place predicate p, where $\mathtt{p}(t_1, t_2, t_3)$ is intended to mean that $[t_1]$ plus $[t_2]$ is $[t_3]$. Thus we might describe the intended model I as follows. The carrier $[I]$ consists of the digit strings 0,1,2,3,4,5,6,7,8,9,10,11 and so on. $[I\mathtt{o}] = 0$, and $[I\mathtt{s}]$ is the function which maps 0 to 1, 1 to 2, 2 to 3, and so on. $[I\mathtt{p}]$ should be a predicative representation of addition, containing all triples $\langle x, y, x + y\rangle$ (so $\langle 0, 3, 3\rangle, \langle 2, 2, 4\rangle$, and so on).

The model (T say) used by Prolog is the *least Herbrand* model (Lloyd 1984), also called the *freely generated* or *(canonical) term* model. In this model, the elements of the carrier are just the ground terms (i.e., the terms containing no variables), to wit, o,s(o),s(s(o)), and so on. $[T\mathtt{o}] = \mathtt{o}$, and $[T\mathtt{s}]$ is the function which maps any ground term t to the ground term $\mathtt{s}(t)$. $[T\mathtt{p}]$ contains triples of ground terms such that the number of occurrences of s in the first element, plus the number in the second, is equal to the number in the third (for example $\langle \mathtt{o}, \mathtt{s}(\mathtt{s}(\mathtt{o}))), \mathtt{s}(\mathtt{s}(\mathtt{s}(\mathtt{o})))\rangle$, $\langle \mathtt{s}(\mathtt{s}(\mathtt{o})), \mathtt{s}(\mathtt{s}(\mathtt{o})), \mathtt{s}(\mathtt{s}(\mathtt{s}(\mathtt{s}(\mathtt{o})))))\rangle$, and so on).

How can T be called an implementation of addition, if it is not the same as the intended model I? If we imagine the objects of the carrier as dots on a blackboard, with arrows to show how the syntax of the signature is interpreted (something like Figure 1–1), we can look on 0,1,2,3... as being just labels for the objects, which help us explain what arrows should point where. The only difference in the blackboard diagram of T would be a one-to-one relabelling, of 0 to o, 1 to s(o), and so on. The two models are called *isomorphic*. It is impossible to distinguish them using only the syntax of the signature: there is no sentence in that syntax which is satisfied by one but not by the other. As models of E, there is really nothing to choose between them. There are many other models of E which are also isomorphic to I. They are all one-to-one re-labellings. We might

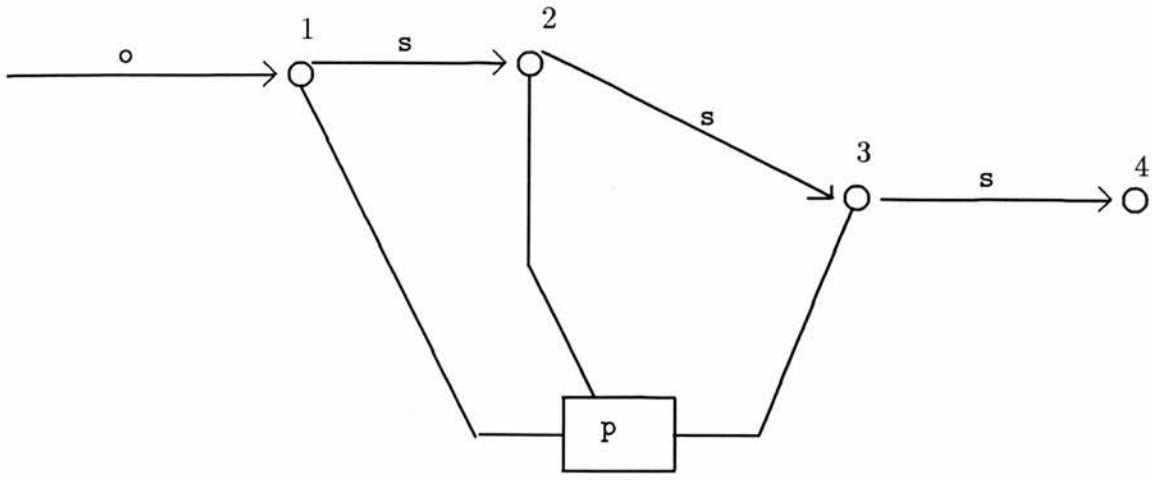

Figure 1–1: Blackboard model of I

label the points by 0,1,10,11,100,101..., or by 1,11,111,1111..., or by any other such sequence.

Isomorphism is an *equivalence relation* on models, which is to say, it is reflexive (each M isomorphic to M), symmetric (if M is isomorphic to M', M' is isomorphic to M), and transitive (if M is isomorphic to M' and M' is isomorphic to M'', M is isomorphic to M''). Thus any collection of models over the same signature can be sorted into distinct *isomorphism classes* of mutually isomorphic models. The isomorphism class of I and T is a very special one, the class of *initial* models. Often we speak of *the* initial model, since isomorphic models are not distinguishable in the logic anyway. There are, however, non-initial models of the axioms in E. The simplest of these, O say, has only one object, call it 0, with $[\mathsf{o}] = 0$, $[\mathsf{s}]0 = 0$, and $[\mathsf{p}] = \{\langle 0, 0, 0\rangle\}$. This was clearly not the sort of model we had in mind in writing E. The initial models are the ones we were after. What distinguishes the initial from the non-initial models?

We have already seen one characterisation, which is to say that an initial model is isomorphic to the canonical term model. To build the term model in a language without equality, as we have seen, we set the carrier to be the collection of terms, and then put into the interpretation of the predicates those terms whose value *must* be there, in virtue of the supplied axioms. For instance, the first axiom of E tells

us that $[T\mathsf{p}]$ must contain, for every ground term t, the tuple $\langle \mathsf{o}, t, t \rangle$. Sometimes, this procedure is not well-defined. For instance, if we have an axiom $\mathsf{p}(\mathsf{o}) \vee \mathsf{q}(\mathsf{o})$, we know we should be trying to put the term o into either $[\mathsf{p}]$, or $[\mathsf{q}]$, but we have no way of telling which. In such cases, there is no canonical term model, and there are no initial models. The reason for the restriction to Horn clauses in Prolog is that a collection of Horn clauses is guaranteed to have an initial model.

Another characterisation of the initial models of E is that they have the property of satisfying exactly those sentences over the signature of E which are satisfied by *every* model of E. I, T, O and every other model of E must satisfy sentences like $\mathsf{p}(\mathsf{s}(\mathsf{o}), \mathsf{s}(\mathsf{o}), \mathsf{s}(\mathsf{s}(\mathsf{o})))$. In order that a model fail to be isomorphic to the term model, it must be the case that it identifies the values of some terms, and/or contains some objects which are not *reachable* (are not the value of any ground term). For any such model it is possible to devise a sentence which distinguishes it from the initial models. For O, one such sentence is $\mathsf{p}(\mathsf{s}(\mathsf{o}), \mathsf{o}, \mathsf{o})$. Thus any non-initial model satisfies sentences not satisfied by all models. One way to describe the operation of Prolog when faced with a query ϕ given E, is to say that it tries to prove that ϕ is true in every model of E. It does this by reference to an initial model.

The situation is slightly more complicated in languages with equality. In that case, the objects of the model are not terms, but equivalence classes of terms. If the axioms insist that two terms be equal (say $\mathsf{o} + \mathsf{o} = \mathsf{o}$), then those terms are held to be equivalent. The equivalence classes are closed under symmetry and transitivity. A term is interpreted by its equivalence class. Thus both o and $\mathsf{o} + \mathsf{o}$ are interpreted by the class containing both of them ($[\mathsf{o} + \mathsf{o}] = [\mathsf{o}]$), and the sentence $\mathsf{o} + \mathsf{o} = \mathsf{o}$ is satisfied. Assignment of tuples to predicates proceeds much as before, except that the objects in the tuples are equivalence classes.

The central idea of algebraic specification is to abstract away from details of algorithm by identifying any program with the algebra defined by its functionality. Early work in the area, such as Goguen et al. (1975), constructed specifications from a signature and a set of equations, and associated these with the initial algebra. In Section 1.2 we saw some reasons why it would be useful to be able to refer to other models too, and we shall consider this question further in the

following section. Specifications which refer to models not all mutually isomorphic are called *loose*. ASL (Sannella and Wirsing 1983), the specification language used in this thesis, allows such specifications.

1.6 Specification and Implementation

Suppose that we wish to model some observable phenomena. We begin by fixing some vocabulary in which to make our observations: a signature. We proceed to construct logical sentences using this vocabulary, which describe the regularities we observe (or seem to observe) in the system. In order to sharpen our understanding of the system, and to see to what extent our description is correct or adequate, we need to produce a model. If the sentences we produce all happen to have a positive conditional form ($\phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_n \rightarrow \phi_0$ for some $n \geq 0$ — the $n = 0$ case is simply ϕ_0 — where for each i, ϕ_i is atomic), then they define an initial model, which has the useful property of satisfying only those sentences satisfied by *all* models of our description.

In general, however, there is no reason we should suppose that our observations need be restricted to this class of sentences. For instance, the sentence $\forall x \,.\, \neg(\texttt{lexical}(x) \wedge \texttt{phrasal}(x))$ may well be the clearest way of expressing some observed property. A description including such a sentence will not, in general, admit an initial model. In a specification language in which the only specifications are collections of sentences, and in which these are always interpreted initially, a set of observations including such a sentence has no formal meaning. We must strive to reformulate our description in such a way that it does conform to the conditional sublogic; but having done so, there is no way of giving formal expression to the question of whether the model so specified conforms to our original observation, because that original observation *has* no formal status, even though it may be the clearest and most natural expression of the properties we require.

Thus it is natural to require (at least) two *different* ways of specifying a class of models: one to be interpreted by the class of *all* models of the system (in which

we may use any axiom), and another to be interpreted by the isomorphism class of *initial* models of the axioms (for which case we must restrict ourselves to the conditional form). A specification which admits non-isomorphic models is called *loose*. These two alternatives can be enshrined in the definition of two distinct *specification-building operations.*

As mentioned, it is usually useful in trying to understand a system to have a concrete model. The abstraction of algebraic specification is to identify implementations (i.e. programs) with single models (or collections of isomorphic models). In (software) engineering, the production of an implementation at an early stage, for investigative purposes, is called *prototyping.* In order to move from a loose specification to a concrete model, it may be necessary to introduce some constructive syntax which does not correspond directly to anything observable in the system. We would want to say that the meaning of such a specification resides in its behaviour with respect to the syntax corresponding to observable features of the system. Thus we require the means to *hide* particular details of construction: a third specification-building operation.

In general the move from loose specification to model may involve many such more-or-less arbitrary design decisions. Specifications will be more flexible and understandable if we break this process up, describing each design decision by a *separate* specification. The overall progress toward description occurs by *stepwise refinement*, each step consisting of producing a refined specification, reflecting either a new, independent observation of the system, or a constructive design decision. Thus these design decisions are motivated, not by observation of the physical system, but by the need to decide *something* in order to produce a program, and (by the way) by the desire to keep that program simple and comprehensible. Thus we have two, mutually informative, processes of refinement: one which gradually refines our abstract understanding of the system, and, taking off where that leaves off, another which produces actual models of the abstract specifications, for the purpose of further investigating the correctness and adequacy of the abstraction.

A useful facility in this process will be the ability to put two specifications together in such a way that models of the resulting specification must be usable in

interpreting either one of the two constituent specifications. Breaking up complex specifications by such means into simpler subspecifications again makes them easier to understand, and more flexible, in that the same subspecifications may be a useful component of many, very different, larger constructions. At the level of a concrete model, a particular construction may only rely on fairly trivial details of a subconstruction. In that case, we can define a *parameterised* model construction, which, supplied with any suitable value for the subconstruction, will complete the larger construction on top of it.

Sannella (1989) describes a discipline for producing programs using stepwise refinement, in the Extended ML specification language. This thesis employs the specification language ASL (Sannella and Wirsing 1983), and in the next section, I describe ASL specification-building operations for performing all the sorts of tasks just outlined. ASL was designed as a *kernel* language, in terms of which higher-level languages may be defined (for instance, Sannella and Tarlecki (1986) represents an early attempt to define the semantics of Extended ML in terms of ASL). I chose this language partly because its semantics is quite simple and easy to understand, though the obverse of this is that specifications quickly get syntactically complex. Another reason for the choice was the hope that it might become apparent in the course of this work what kind of higher-level operations would be particularly useful in linguistic applications, but unfortunately this has not become any clearer to me. I think that the choice of a language which is not tied to initial semantics was a good one, for the sorts of reasons outlined above, and I hope that such will become evident as the thesis progresses. Extended ML would have offered the advantage that it is a true *wide-spectrum* language, designed to use the same logic as the ML programming language, so that, just as Horn clause descriptions in predicate logic can be run as Prolog programs, Extended ML descriptions of a certain form can be run as ML programs. On the other hand, I am not convinced that it is wise to associate one's specification language too closely with a particular programming language, since it may incline one toward descriptions nearer to those that can be run, at the expense of a perhaps clearer description, which may be more easily implemented in another

programming language altogether. (For instance, ML is a functional programming language, but in many ways a logic programming language such as Prolog is more suited to computational linguistics. Sannella and Wallen (1987) describes how ML-style modules may be added to Prolog, but at the time of writing this is only just being implemented (Paxton 1992)). There are many subsets of ASL, even restricted to only one logic, which could be given procedural interpretations — unfortunately, running implementations do not exist. However in Chapter 4, we see some theory which allows programs written in existing programming languages (such as Prolog) to be incorporated into ASL specifications.

1.7 Sorts and ASL Specification Operations

Algebraic specification is usually done in *multisorted* algebra. A standard algebra begins with some domain of interpretation. In multisorted algebra there may be several different domains of interpretation. A multisorted algebra, then, consists of the various domains of interpretation, together with functions upon them. ASL-style specifications talk about classes of algebras. Specification-building operations, by which specifications may be combined to produce compound specifications, will map classes of algebras to classes of algebras.

1.7.1 Sorts and basic specifications

In multisorted logic, signatures include a set of *sort* names, and arities cannot be described just by numbers, but must employ sort names. For instance, if we use the sort name `Nat` to represent natural numbers, we might describe the arities of the operations in E by

$$\textbf{opn } \mathtt{o} :\rightarrow \mathtt{Nat}$$

$$\textbf{opn } \mathtt{s} : \mathtt{Nat} \rightarrow \mathtt{Nat}$$

$$\textbf{pred } \mathtt{p} \subseteq \mathtt{Nat} \times \mathtt{Nat} \times \mathtt{Nat}$$

So `o` is a constant (nullary operation) of sort `Nat`, `s` is a unary operation of sort `Nat`, and `p` is a three-place predicate over `Nat`. In a model M, sort names are each assigned a domain of interpretation, or *carrier*, and function and predicate names are assigned functions and predicates of appropriate arity: so `Nat` is assigned some set $[M\texttt{Nat}]$, and `s` is assigned a function $[M\texttt{s}] : [M\texttt{Nat}] \to [M\texttt{Nat}]$. Variables must now be assumed to be associated with a particular sort, and then terms of the various sorts can recursively be built up. For instance if $\texttt{f} : \texttt{A} \to \texttt{B}$, then $\texttt{f}(t)$ is a valid term if and only if t is of sort `A`, in which case $\texttt{f}(t)$ is of sort `B`. Quantifiers should now specify the sort of their associated variable, for instance $\forall x{:}\texttt{Nat} \,.\, \texttt{p}(\texttt{o}, x, x)$, but may be omitted where obvious. Similarly universal quantifiers "$\forall$" may be omitted at the outermost level. So all the following are equivalent:

axiom $\forall x{:}\texttt{Nat} \,.\, \texttt{p}(\texttt{o}, x, x)$
axiom $\forall x \,.\, \texttt{p}(\texttt{o}, x, x)$
axiom $\texttt{p}(\texttt{o}, x, x)$

As an example, *semigroups* consist of some set $[\texttt{Gpd}]$ (the domain of interpretation corresponding to the sort name `Gpd`, for "groupoid") together with a binary function $[\_ \cdot \_] : [\texttt{Gpd}] \times [\texttt{Gpd}] \to [\texttt{Gpd}]$. The following specification might be used to define what it means to be a semigroup:

(1.3) (Semi-group)

SGP = **sort** `Gpd`
opn $\_ \cdot \_ : \texttt{Gpd} \times \texttt{Gpd} \to \texttt{Gpd}$
axiom $x \cdot (y \cdot z) = (x \cdot y) \cdot z$

SGP is a *basic* specification. It begins by giving *sort names* (here just `Gpd`), each of which is to be given a corresponding *carrier*, or domain of interpretation ($[\texttt{Gpd}]$). In addition we may give *operation names*, corresponding to functions on these domains. Lastly we may say something about how these functions are to behave. For instance, here we require $[\_ \cdot \_]$ to be associative. A *model* for a specification (such as SGP) is an algebra consisting of a carrier set to interpret each sort name (in this case just `Gpd`), and a function to interpret each operation name (in this case

just one function on the set, to interpret $\_ \cdot \_$). If there are additional conditions, such as the associativity in SGP, the models must also satisfy these.

So no matter what values in [Gpd] x, y and z take on, if we are to have a model of SGP, the resulting values of $[x \cdot (y \cdot z)]$ and $[(x \cdot y) \cdot z]$ must be the same. Models will include: strings over various alphabets, using concatenation to interpret $\_ \cdot \_$; equivalence classes of $\_ \cdot \_$-terms over some set of variables under associativity (i.e.: each equivalence class contains all alternative bracketings of a term), using term construction to interpret $\_ \cdot \_$; natural numbers (or integers, or real or complex numbers, or matrices of them) under addition (or multiplication); any collection of functions closed under composition; any group or (small) category (in the sense of group or category theory), and many more. For instance, we may define a model in which the carrier is the set of non-empty strings over $\{0,1\}$, to wit $\{0,1,00,01,10,11,000,001,\ldots\}$, and $[\_ \cdot \_]$ is given by concatenation: so in order to calculate the value of $[\_ \cdot \_]$ given two input strings (101 and 001 say), we simply write the two strings down, in that order, juxtaposed (101001).

(1.4) Call this model N.

1.7.2 Refinement, enrich and derive

SGP is a basic specification. It describes a signature, and some conditions on that signature. Its semantics is the class of all models over that signature which satisfy those conditions. We don't have to have axioms in a basic specification:

$$\begin{array}{lll} \mathsf{BINOP} & = & \textbf{sort } \mathtt{Gpd} \\ & & \textbf{opn } \_ \cdot \_ : \mathtt{Gpd} \times \mathtt{Gpd} \to \mathtt{Gpd} \end{array}$$

is a valid specification. Like SGP, its models must interpret Gpd and $\_ \cdot \_$, but unlike in SGP, $[\_ \cdot \_]$ need not be associative. SGP is a *refinement* of BINOP (SGP $\sqsubseteq$ BINOP), because every model of SGP is a model of BINOP. Basic specifications are the atomic building blocks from which *compound* specifications are built, using *specification-building operations*. For instance, the operation **enrich** may be used to build, from BINOP, a compound specification SGP2:

SGP2 = **enrich** BINOP **by**
axiom $x \cdot (y \cdot z) = (x \cdot y) \cdot z$

SGP2 has exactly the same models as SGP. **enrich** is just one example of many possible specification-building operations which can be useful in building compound specifications. The operations used in this paper are drawn from the ASL specification language (Sannella and Wirsing 1983). For any such specification-building operation, the constituent specification(s) (such as BINOP in SGP2) will be semantically grounded as a collection of models, and the compound specification will also need to be so grounded. Clearly then, the semantics of specification-building operations like **enrich** must consist of maps from collections of models to collections of models. The construction

enrich A **by**
sorts S **opns** F **axioms** E

admits only models which besides having all the attributes of models of A, also interpret S and F and satisfy the conditions in E. For example, the class of monoids (semigroups with identity) is (almost) a refinement of SGP:

(1.5) (Monoid)
MON = **enrich** SGP **by**
opn e :⟶ Gpd
axiom $x \cdot \mathsf{e} = \mathsf{e} \cdot x = x$

This is a refinement *not* because it refers to SGP, but because every model of MON is also a model of SGP, if you ignore e. In fact to be a true refinement this "forgetting" must be said explicitly, because to say every model of B is a model of A implies that they have the same signature. We write Sig[A] to denote the signature of a specification A, so for instance Sig[SGP]=BINOP. So MON is not quite a refinement of SGP, because Sig[SGP] is strictly contained in Sig[MON], where we require them to be equal. However the specification

(1.6) (Hide identity)

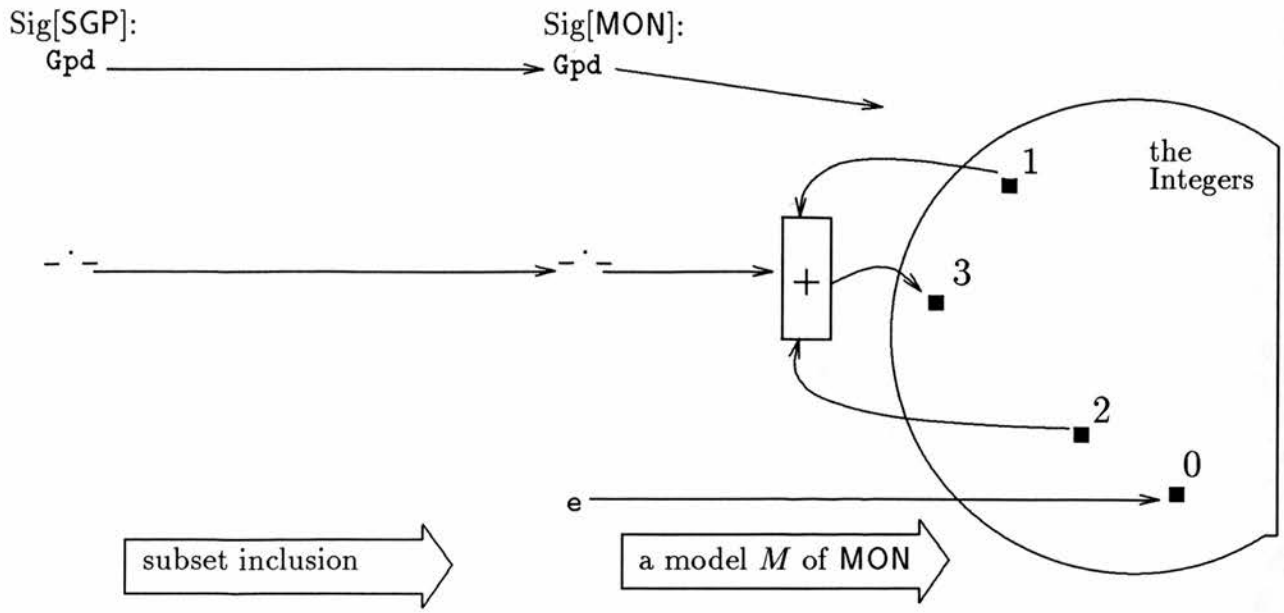

Figure 1–2: The Sig[SGP]-reduct of M

HIDEID = **derive from** MON **by inclusion of** Sig[SGP]

is truly a refinement of SGP, since the effect of deriving by an included signature (Sig[SGP] is a subset of Sig[MON]) on any particular model M of MON, is simply to forget about interpreting the rest of Sig[MON] (namely e). I write **inclusion of** Sig[SGP] to specify the *signature morphism* (map) which maps every piece of syntax in Sig[SGP] to the same piece of syntax in Sig[MON]. Since any Sig[MON]-model M is itself a map from Sig[MON] into some interpretative domain, we may compose the inclusion morphism *to* Sig[MON], with the model map M *from* Sig[MON], to obtain a map from Sig[SGP] into the interpretative domain of M, i.e. a model of Sig[SGP], called the Sig[SGP]-reduct of M, written $M|_{\text{Sig[SGP]}}$ (see Figure 1–2). The carriers of the models of HIDEID still contain their unit elements, but the signature no longer names them. A model consists of carriers, *together* with maps which interpret the signature in terms of those carriers. Then $M|_{\text{Sig[SGP]}}$ no longer so maps the operation name e, despite the fact that the object to which M maps e is still present in the domain of interpretation, just because e is not *in* the signature of $M|_{\text{Sig[SGP]}}$. For instance, the integers under addition, with e interpreted as zero, form a model of MON, and the corresponding

model of HIDEID is exactly the same except that it can no longer interpret any syntax involving e. Zero is still there, it's just that e is no longer mapped to it. In particular, zero will not be reachable, as there is no term in the new signature which names it. So Sig[HIDEID]=Sig[SGP], and also every model of HIDEID is a model of SGP (write Mod[HIDEID]⊆Mod[SGP]). Note that the converse is not true (the inclusion is proper, Mod[HIDEID]⊂Mod[SGP]). There are models of SGP which are *not* models of HIDEID. For instance, N (1.4), the non-empty strings over {0,1}, under concatenation, forms a model of SGP, but not of HIDEID, since there is no unit element in the domain. If we put the empty string into the domain, the result is still a model of SGP, but is now also a model of HIDEID, since it is the same as that model of MON which interprets $\_ \cdot \_$ by concatenation over the same domain, but also interprets e by the empty string.

So refinement is a semantic notion, where the fact that one specification may be a syntactic component of another (as A is of **enrich** A **by**...) is simply a convenience for neatly organising the information, in a way which reflects somewhat ideas on modular program construction. These are separate, albeit related, issues. Note that the **derive** construct may be used with any signature morphism, not only inclusions, as we shall see in Subsection 1.7.4.

In MON we used the **enrich** construction to say that a monoid is a semigroup, with certain additional properties. This would be particularly appropriate if the definition of the additional properties relied heavily on the properties of SGP, but in the case of MON, the property of having an identity is not particularly bound to that of associativity. It might therefore be preferable if we could produce a separate specification of binary operations with identity (which we might be able to use elsewhere, even in the absence of associativity), and say that a monoidal algebra is both a semigroup, and an algebra with (binary) identity.

ID = **enrich** BINOP **by**
opn e :⟶ Gpd
axiom $x \cdot \mathsf{e} = \mathsf{e} \cdot x = x$

MON2 = SGP+ID

Any model of the specification A+B must have all the attributes of models of both A and B, thus MON2 describes exactly the same class of models as MON, but makes clearer the mutual independence of the required properties of monoids (though in such a simple case, it probably makes little difference).

1.7.3 Subsorts, reachable and extend

In single-sort algebra, a model was said to be *reachable* if every object was the value of a ground term. In multisorted algebra, this can be generalised to say that a given set of sorts S is reachable in a model (or the model is reachable *on* those sorts), if every object of those sorts is the value of a term containing no variables of those sorts. That is, the terms which name the objects of those sorts may include variables of sorts *not* in S (we say the objects, and the model, are reachable *from* the sorts not in S). Those variables may then be given any value in the carrier of the appropriate sort. Thus the variables act as slots where inputs (from sorts other than S) can be fitted, and every element of the sorts in S must be expressible by some such choice of inputs. We might also describe this by saying that the sorts S are (in that model) *generated* by the elements of the sorts not in S. Consider the example of a semigroup being generated by some subset of its underlying domain.

$$\begin{array}{lcl} \mathsf{SUB} & = & \textbf{sorts } \mathtt{Sub,Gpd} \\ & & \textbf{axiom } \mathtt{Sub} \subseteq \mathtt{Gpd} \end{array}$$

(1.7) (Generated Semigroup)

$$\begin{array}{lcl} \mathsf{GSGP} & = & \textbf{reachable } \mathsf{SUB} + \mathsf{SGP} \\ & & \textbf{on } \{\mathtt{Gpd}\} \end{array}$$

The axiom $\mathtt{Sub} \subseteq \mathtt{Gpd}$ means that in every model of SUB, every object of [Sub] must also be an object of [Gpd].[2] Models of GSGP must do the jobs of both SUB

[2]Sometimes I will use $x{:}\mathtt{Gpd}$ as an atomic formula, for instance $\forall x{:}\mathtt{Sub}\,.\,x{:}\mathtt{Gpd}$. What should this mean, when the variable x has already been given the sort Sub? In this case,

(two sorts, one within the other), and SGP (associative binary operation), and further, the elements of [Gpd] must be reachable from those of [Sub]. This simply means that every element of the semigroup can be formed from elements of the subset by successive application of the associative operation $\_ \cdot \_$. That is, the *closure* of the subset under $\_ \cdot \_$ is the whole semigroup, and not just some subset of it. For instance, we can construct a model of Sig[GSGP] which is like the model N (1.4) of SGP, but which also interprets Sub by the set {0,1}. This is a model of GSGP, because the non-empty strings over the alphabet {0,1} are generated by that alphabet (under concatenation). Equivalently, any element of [Gpd] may be expressed as a term using $\_ \cdot \_$, where variables take on values in Sub: for instance, $010 = [x \cdot (y \cdot z)]$, where x takes on the value 0, y the value 1, and z the value 0. However the model of Sig[GSGP] in which $[\mathtt{Sub}] = \{0, 1\}$, and which, like N, has $\_ \cdot \_$ interpreted by concatenation, but in which [Gpd] contains all strings over {0,1}, including the empty string, is not a model of GSGP, because it is impossible to produce the empty string from 0's and 1's using concatenation. But if we put the empty string into [Sub], we would again have a model of GSGP.

(1.8) Call this model E.

In like manner, we may describe the reachable monoids:

(1.9) (Generated Monoid)

GMON = **reachable** SUB + MON
on {Gpd}

One important use of reachability is that it gives us enough structure to make recursive definitions. For instance, consider

---

$x : \mathtt{Gpd}$ should be interpreted as stating that the value of x is in the carrier of Gpd. So $\forall x : \mathtt{Sub} \,.\, x : \mathtt{Gpd}$ is the same as $\mathtt{Sub} \subseteq \mathtt{Gpd}$. It can be considered an abbreviation for an expression like $\forall x : \mathtt{Sub} \,.\, \exists y : \mathtt{Gpd} \,.\, x = y$. If I fail explicitly to give a sort for a variable, but the only possibilities are, say, Sub and Gpd, where $\mathtt{Sub} \subseteq \mathtt{Gpd}$ (as, for example, with the variable i in **enrich** SUB+SGP **by** $\exists i \,.\, i \cdot i = i$), one should assume the *most general* sort, that is Gpd, the containing (rather than contained) sort.

FSUB = **sort** Sub
opn f : Sub → Sub

FEND = **enrich** SUB + SGP **by**
opn f : Gpd → Gpd
axiom $\mathtt{f}(x \cdot y) = \mathtt{f}(x) \cdot \mathtt{f}(y)$

If F specifies a single model, F say, of FSUB+GSGP, then F+FEND also has only a single model. F acts as the base case for defining f, and FEND is the recursive case. But if F specified a model of FSUB+SGP, but not FSUB+GSGP (i.e. the model F was not reachable on Gpd), then F+FEND has many models, since for any unreachable $i \in [\mathtt{Gpd}]$, $[\mathtt{f}(i)]$ could take on any value in [Gpd].

The specification

UR1 = **enrich** SUB + SGP **by**
axiom $\exists x{:}\mathtt{Gpd} \,.\, \neg x : \mathtt{Sub} \wedge \forall r{:}\mathtt{Sub}, y{:}\mathtt{Gpd} \,.\, x \neq r \cdot y$

certainly has plenty of models. For instance in the model

[Sub] = $\{0\}$
[Gpd] = $\{0, 1\}$
$[\cdot]$ is given by ordinary multiplication

the object 1 satisfies the requirements for x in the axiom of UR1. But the specification UR1+GSGP is inconsistent, i.e. is satisfied by no model, since in a reachable model, every element of [Gpd] is the value of a $\_ \cdot \_$-term with variables in Sub, for some value of those variables, and since

1. using associativity, any such term involving more than zero $\_ \cdot \_$'s can be rewritten in the form $r \cdot t$, where r is a variable of sort Sub and t is a $\_ \cdot \_$-term with variables in Sub, and

2. any such term involving zero $\_ \cdot \_$'s is a variable of sort Sub.

Note however that although every model of UR1 is unreachable, not every unreachable model of SUB+SGP satisfies the axiom of UR1. For instance in the model

$$\begin{aligned} [\mathtt{Sub}] &= \{0\} \\ [\mathtt{Gpd}] &= \{0,1\} \\ [\cdot] &\text{ is given by addition modulo 2 (so } 1+1=0 \text{ modulo 2)} \end{aligned}$$

the object 1 is the only unreachable element, but $1 = [\cdot](0,1)$, so 1 cannot fill the role of x in the axiom of UR1. In general, one cannot write an axiom which is satisfied by all and only the reachable models. That is why we require an explicit specification-building operation in our metalanguage if we are to be able to limit specifications to reachable models (in order to get enough structure to do things like making recursive definitions as in FEND above), but still have the ability to talk, in other specifications, about the class of all models (or to refer to reachability on different sorts).

Earlier, in Section 1.6 we saw that we might, as well as the class of *all* models of a set of axioms, like to be able to refer to the class of *initial* models of those axioms. In ASL, this class can be picked out using a specification like

(1.10) NAT = **extend** $\emptyset$ **by**
opn o :→ Nat
opn s : Nat → Nat
pred p ⊆ Nat × Nat × Nat
axiom p(o, x, x)
axiom p(x, y, z) → p(s(x), y, s(z))

Here $\emptyset$ stands for the *empty* specification, which has an empty signature and no axioms. It has one (empty) model. The construction **extend** A **by**... specifies the *free extension* of the models of A by the given syntax and axioms. The free extension of a model M is given (up to isomorphism) by a term model with a constant corresponding to every object in M. This construction may be performed

subject to some conditions. Thus we can specify the freely generated (or simply *free*) semigroup over Sub:

(1.11) (Freely generated semigroup)

FGS = **extend sort** Sub
by **sort** Gpd
opn _ · _ : Gpd × Gpd → Gpd
axiom Sub⊆Gpd
axiom $x \cdot (y \cdot z) = (x \cdot y) \cdot z$

Like **derive**, the **extend** operation transforms sets of models by performing a construction, in this case free extension, on every model of the set. Intuitively, the free extension assumes the least objects it can, subject to the given axioms, without equating any new terms except those explicitly identified by axioms. The free extension of a set A by the binary operation _·_ and the axiom $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ is (isomorphic to) the (non-empty) strings over A. In FGS, we extend the specification **sort** Sub, so all models M' of FGS extend some model M consisting of a set $[M\texttt{Sub}]$. The condition Sub ⊆ Gpd means that, in the extensions M', $[M'\texttt{Gpd}]$ contains the objects of $[M\texttt{Sub}] = [M'\texttt{Sub}]$, plus new objects to serve as the values of $[M'\_ \cdot \_]$. The rule is that that these new objects should all be distinct, from each other, and from $[M'\texttt{Sub}]$, except where the axioms imposed in the extension require them to be identified. For instance, if $o \in [M\texttt{Sub}]$, then we require of a free extension properties including $[M'\_ \cdot \_](o, [M'\_ \cdot \_](o, o)) = [M'\_ \cdot \_]([M'\_ \cdot \_](o, o), o)$, and $[M'\_ \cdot \_](o, o) \neq o$. All such models M' are isomorphic, so we speak of *the* free extension of Sub (by the given operations and axioms). Up to isomorphism, FGS yields the non-empty strings over Sub: we can think of the objects of $[M'\texttt{Gpd}]$ as being non-empty finite lists of objects from $[M\texttt{Sub}]$, with $[M'\_ \cdot \_]$ interpreted by concatenation. Intuitively, all properties of a free extension follow from the properties of the model being extended, and the explicitly added structure. Note FGS ⊑ GSGP, but the converse is not true. For instance, the model E (1.8) of GSGP above, in which $[E\texttt{Gpd}]$ contains all strings over {0,1}, $[E\texttt{Sub}]$ contains 0, 1, and the empty string (written ϵ), and $[E\_ \cdot \_]$ is calculated by concatenation,

is not a model of FGS, because it equates some $\_ \cdot \_$-terms formed from elements of $[E\texttt{Sub}]$, e.g. $[E\_ \cdot \_](\epsilon, \epsilon) = \epsilon$. We can similarly specify the freely generated monoid, consisting of all strings over Sub (including the empty string):

(1.12) (Freely generated monoid)

FGM = **extend sort** Sub
by **sort** Gpd
opn $\_ \cdot \_ : \texttt{Gpd} \times \texttt{Gpd} \rightarrow \texttt{Gpd}$
opn $\texttt{e} :\rightarrow \texttt{Gpd}$
axiom $\texttt{Sub} \subseteq \texttt{Gpd}$
axiom $x \cdot (y \cdot z) = (x \cdot y) \cdot z$
axiom $x \cdot \texttt{e} = \texttt{e} \cdot x = x$

Initial models are reachable models in which predicates are only inhabited as required, and terms are only equated as required. Although it may be possible to rewrite collections of axioms in such a way as to restrict from reachable to initial models without the need of a special **extend** operation, in the case of performing a free extension on an existing specification, we would need to think of the meaning of the embedded specification as a set of axioms. We have just seen that a specification **reachable** T **on** S cannot be characterised in this way. Thus the specification-building operation **extend** is required, if we are to be able to refer to free extensions of arbitrary specifications.

Note however that **extend** cannot be used with arbitrary axioms, and it can be quite difficult to tell whether it is well-defined given some particular set of axioms. However conditional axioms of the form $\phi_1 \wedge \ldots \phi_n \rightarrow \phi$, where $\phi, \phi_1 \ldots \phi_n$ are atomic formulae and all variables are universally quantified, are guaranteed to produce a well-defined free extension. Even if it is well-defined, the resulting model can sometimes behave in a most counterintuitive fashion. Fortunately, if we limit ourselves to *persistent* extensions, the result is well-defined and well behaved (Ehrig and Mahr 1985). Persistence of an extension is most easily guaranteed by checking for two other properties, *sufficient completeness*, and *hierarchical consistency*, which may be explained as follows. Because the **extend** operation in FGM

only constrains the new objects it creates to be in the new sort `Gpd`, rather than the existing `Sub`, it does not create new objects of existing sorts. This property is called sufficient completeness. The only pre-existing terms are variables x of sort `Sub`. Then only terms to which an existing term x will be equated are $x \cdot \mathsf{e}$, $\mathsf{e} \cdot x$, or other terms formed using $\_ \cdot \_$ from one occurrence of x and arbitrarily many occurrences of `e`. Because all of these denote the same object, no existing objects will equated, or more generally, invested with any new properties which could have been expressed in the pre-extension syntax. This property is called hierarchical consistency. Together, these two properties guarantee a persistent extension.

1.7.4 Parameterisation and signature morphisms

The free extension is a very concrete construction. It maps single models to single models. Every model of FGM is specifically constructed from a model of **sort** `Sub`. Any implementation M of **sort** `Sub` (so M has only one model M, up to isomorphism) can be made into an implementation of FGM (the strings over $[M\mathtt{Sub}]$), by just adding FGM, so: M+FGM. Using this fact, and a little lambda (λ) notation, we can construct a *parameterised* implementation.

(1.13) $\quad \mathsf{PFGM} \quad = \quad \lambda\mathcal{X}{:}\mathbf{sort}\ \mathtt{Sub}\ .\ \mathcal{X} + \mathsf{FGM}$

For any specifications A and B, $\mathsf{F} = \lambda\mathcal{X}{:}\mathsf{A}\ .\ \mathsf{B}$ is a function on specifications. It takes any specification $\mathcal{X}$ which refines the specification A and performs the constructions given by the specification B. Semantically, then, it takes a set of models of A to a set of models of (the appropriate instantiation of) B. A gives the *type* of the parameter $\mathcal{X}$. If $\mathsf{A}' \sqsubseteq \mathsf{A}$, then the result of applying F to A' is written $\mathsf{F}(\mathsf{A}')$. If B' is the result of substituting A' for every *free* (i.e. not bound by an embedded λ) occurrence of $\mathcal{X}$ in B, then the models of $\mathsf{F}(\mathsf{A}')$ are exactly the models of B'. So for instance PFGM(M) specifies exactly the same models as M+FGM. If a parameterised specification happens to map single models to single models (as PFGM does), then it can be used as a parameterised implementation, since for any implementation M of **sort** `Sub`, PFGM(M) is an implementation of FGM.

It will be recalled (Section 1.5) that the central idea of algebraic specification is to abstract away from details of algorithm by identifying any program with the algebra defined by its functionality. PFGM may be thought of as a concrete module (or parameterised program) in the programming language sense: linked to an appropriate program module (model) M, it yields a program (i.e. up to isomorphism, a single model). For example if NAT, containing sort `Nat`, is a program implementing the natural numbers, then

NAT* = PFGM(**derive from** NAT **by** [`Sub`$\longmapsto$`Nat`])

references: NAT (1.10)

implements strings of natural numbers. This illustrates the other major use of **derive**, in renaming. **derive from** NAT **by** [`Sub`$\longmapsto$`Nat`] allows us to interpret any model M of NAT as a model of **sort** `Sub` by first mapping `Sub` to `Nat`, and then interpreting `Nat` using M. In this way, we match `Nat` to the signature required of the parameter $\mathcal{X}$ of PFGM.

[`Sub`$\longmapsto$`Nat`] is meant to indicate a *signature morphism* which maps the sort name `Sub` (and that alone) to the sort name `Nat`, and leaves everything else unchanged. A signature morphism is a map on syntax. It maps sort names to sort names and operation names to operation names. There is no standard syntax for specifying signature morphisms. In this thesis, in addition to the syntax just described, I use **inclusion of** Sig[SGP] (as in the specification HIDEID (1.6)) to specify the *inclusion* morphism, which maps all the syntax in Sig[SGP] unchanged into Sig[MON]. Thus the map

inclusion : Sig[SGP] $\rightarrow$ Sig[MON]

differs from the identity map

identity : Sig[SGP] $\rightarrow$ Sig[SGP]

only in that it has a different range, Sig[MON], differing from Sig[SGP] only in containing the constant `e`. It will be useful to have syntax to add to a description

like [Sub$\mapsto$Nat] the information that some piece of syntax should *not* appear in the domain (as with e in the inclusion map), in order to override the default that syntax not mentioned should appear in both signatures ("everything else unchanged"). I shall write this [$\not\mapsto$e]: the morphism which maps everything except e to itself, so e appears in the range, but not the domain, of the morphism. So we could have written [$\not\mapsto$e] instead of **inclusion** in HIDEID, as they would describe the same morphism. A signature morphism specification like [Sub $\mapsto$ Nat, $\not\mapsto$ o, s, p] might be read "rename Nat to be Sub, and forget o, s and p". The signature of

$$\mathsf{DNAT} \quad = \quad \textbf{derive from NAT by } [\mathtt{Sub} \mapsto \mathtt{Nat}, \not\mapsto \mathtt{o}, \mathtt{s}, \mathtt{p}]$$

(the domain signature of the morphism) would thus be the same as Sig[NAT] (the range signature of the morphism), but with Sub instead of Nat, and with o, s and p removed. That is, Sig[DNAT] contains the sort Sub, and nothing else. This is the morphism which ought more properly to have been used in NAT* above, though I shall introduce in Section 1.8 some abbreviatory notation by which the usage in NAT* is well-formed.

By "forgetting" syntax, we can easily make previously reachable elements unreachable. For instance, in NAT (1.10), all of [Nat] was reachable (from $\emptyset$), but in DNAT no element of [Nat] is reachable (can be the value of a term other than a variable of sort Nat). So even though all models of NAT are initial, no model of DNAT is even reachable, let alone initial (regardless of what axioms initiality is considered with respect to).

1.7.5 Constructors, extractors and partial operations

Often it will be convenient to allow ourselves *partial* operations, to be interpreted by partial functions. Whereas an ordinary (total) function $f : A \to B$ associates with every $a \in A$ a unique $f(a) \in B$, if f is partial (write $f : A \dot{\to} B$), then not every (or, indeed, any) $a \in A$ need have an associated value $f(a)$, though where such a value exists, it must be unique. Thus if f : A$\dot{\to}$B names a partial operation

and t is a term of sort A, the term $\mathtt{f}(t)$, even though well-formed, may fail to denote any point in the carrier [A]. We modify the definition of satisfaction of an atomic formula to say that $\mathtt{p}(t_1, \ldots t_i)$ is satisfied if and only if each of $[t_1], \ldots [t_i]$ *does* denote a point in the appropriate carrier (or is *well-defined*, or simply *defined*), and $\langle [t_1], \ldots [t_i] \rangle \in [\mathtt{p}]$. Thus for any term t, the effect of the formula $t = t$ is to assert that t is defined, and nothing more. This we abbreviate $\mathbf{D}\, t$. For partial f, we may take $\mathtt{f}(t)$ to be defined in a free extension, only for those values of t where some axiom of the extension insists it be defined. For instance, in

extend NAT **by**
opn predecessor : Nat$\dot{\rightarrow}$Nat
axiom predecessor(s(x)) = x

predecessor would be defined on [s(t)] for every ground t (so 1,2,3...), but undefined at 0. Partial functions can be useful in so-called *constructor-extractor* implementations.[3] For instance, we can obtain very nearly the effect of FGS (1.11) by adding extractors (and equations) to the constructor _ · _ of GSGP (1.7).

(Non-empty lists)
NEL = **enrich** GSGP **by**
opn hd : Gpd $\rightarrow$ Sub
opn tl : Gpd$\dot{\rightarrow}$Gpd
axiom $\forall r$: Sub.hd(r) = r

---

[3]At the level of programming language, the use of partial operations is often held to be confusing, and, at least in a naive implementation, quite inefficient.Goguen and Meseguer (1987a) and Goguen and Meseguer (1987b) push sort inclusions into the signature, employing an *order-sorted* logic, to overcome these and related problems. This accords with the view, presented in Chapter 4, that different programming languages may be embodied by different *institutions* (logics), and that specifications employing different institutions may be tied together using the **change institution** operation, also presented there. For the purposes of abstract specification, however, it will be simpler to stick with ordinary first order logic.

$$\textbf{axiom } \forall r : \texttt{Sub}, x : \texttt{Gpd}.\texttt{hd}(r \cdot x) = r$$
$$\textbf{axiom } \forall r : \texttt{Sub}, x : \texttt{Gpd}.\texttt{tl}(r \cdot x) = x$$
$$\textbf{axiom } \texttt{hd}(x) = h \wedge \texttt{tl}(x) = t \rightarrow x = h \cdot t$$

$\_ \cdot \_$ is called a *constructor*, because it can be used to construct new objects from old, and the operation `hd` is called an *extractor*, because it can be used to extract the old elements from which a new one is made (Goguen et al. 1975). The use of extractors makes NEL very nearly the same as FGM. The extractor equations make it impossible to equate `Gpd`-terms without also equating `Sub`-terms. For instance in the model E (1.8) described above, we would be trying to set $[\texttt{hd}]([\_ \cdot \_](\epsilon, 0)) = \epsilon$, but this is not consistent with the fact that $[\_ \cdot \_](\epsilon, 0) = 0$, because we would also want $[\texttt{hd}](0) = 0$, so no interpretation of `hd` and `tl` can make E into a model of NEL. For the cases where [Sub] contains at least two elements, the models of NEL are in one-to-one correspondence with the models of FGM, with the correspondence being given by signature inclusion reduct. (When [Sub] has only one element, NEL also admits *cycles*, that is, models of addition modulo some integer n, where $\_ \cdot \_$ is interpreted by addition modulo n, `Sub` by the singleton containing 1, and $[\texttt{tl}(x)]$ is $[x] - 1$ modulo n.) One advantage of directly using reachability (here inherited from GSGP) and constructor/extractor equations, as compared to using **extend** is that the free extension may simply not exist if we start to use axioms involving $\neg$, $\vee$ or $\exists$. But while using **reachable** and/or **derive** with constructors and extractors may get us models in situations where **extend** doesn't, their use still requires considerable care in order to avoid inconsistency. Of course this technique may also give us non-isomorphic classes of enrichments, which **extend** cannot (on a model-by-model basis), but in many cases, it gives classes we can nevertheless successfully reason about. For instance, an axiom $\texttt{f}(1) = 1 \vee \texttt{f}(1) = 0$ is unlikely to produce a well-defined free extension, or indeed an isomorphic class of models, but is likely to produce two isomorphism classes, one for each disjunct, about which we may be able to reason separately.

1.8 Syntax and Semantics of ASL

This section loosely follows the presentation in Sannella and Wirsing (1983), though the choice of logic and specification operations is not identical. It would be pedagogically convenient not to have to describe signatures which distinguish between predicates, total operations, and partial operations. A little thought should convince the reader that any basic specification involving predicates and/or total and/or partial operations may be encoded to similar effect by another basic specification involving only predicates, or, equally, by one involving only total operations (and equality), or by one involving only partial operations (and equality). However these encodings will behave differently with respect to **reachable** and **extend**. The choice of partial operations seems give the most satisfactory solution. The least satisfactory part of this solution is the encoding of predicates. One approach is to let definedness encode predicate satisfaction. Wherever $\mathtt{p}(t_1, \ldots t_n)$ appears in a formula, it may be taken as an abbreviation of $\mathbf{D}\mathtt{p}(t_1, \ldots t_n)$, which in turn abbreviates $\mathtt{p}(t_1, \ldots t_n) = \mathtt{p}(t_1, \ldots t_n)$. In order to determine whether a model satisfies $\mathtt{p}(t_1, \ldots t_n)$, instead of seeing whether $\langle t_1, \ldots t_n \rangle$ lies in a set $[p]$, we must see whether the partial function $[p]$ is defined at $\langle t_1, \ldots t_n \rangle$. The problem with this encoding is that of which value $[p]$ is to take, when it is defined. For instance in a free extension, if we say nothing about what this value is to be, then each ground term $\mathtt{p}(t_1, \ldots t_n)$ will come to denote a new object. As a trick to prevent this sort of problem, we can specify that whenever $\mathtt{p}(t_1, \ldots t_n)$ is defined, its value is equal to that denoted by t_1. Of course $\mathtt{p}(t_1, \ldots t_n)$ can only be satisfied when the terms $t_1, \ldots t_n$ are defined, but the converse does not hold: $\mathtt{p}(t_1, \ldots t_n)$ need not be defined just because $t_1, \ldots t_n$ are. Rather, just as we would need to specify anything required to be in a set-of-tuples interpretation of $[p]$, we must specify any tuples at which the partial function $[p]$ must be defined. The declaration

$$\textbf{pred } \mathtt{p} \subseteq \mathtt{X}_1 \times \mathtt{X}_2 \times \ldots \mathtt{X}_n$$

can be taken to abbreviate

opn $\mathtt{p} : \mathtt{X}_1 \times \mathtt{X}_2 \times \ldots \mathtt{X}_n \dot{\rightarrow} \mathtt{X}_1$

axiom $x = \mathtt{p}(x_1, x_2, \ldots x_n) \rightarrow x = x_1$

The declaration of a total function

opn $\mathtt{f} : \mathtt{X}_1 \times \mathtt{X}_2 \times \ldots \mathtt{X}_n \rightarrow \mathtt{X}$

can be taken to abbreviate

opn $\mathtt{f} : \mathtt{X}_1 \times \mathtt{X}_2 \times \ldots \mathtt{X}_n \dot{\rightarrow} \mathtt{X}$

axiom $\forall x_1 x_2 \ldots x_n . \mathbf{Df}(x_1, x_2, \ldots x_n)$

Both of these encoding axioms are well-behaved in free extensions.

I shall therefore just assume here the logic with partial operations and equality. Putting predicates and total functions into the signatures is not difficult, merely tedious. What follows is probably best read as a sketch, which is easily filled out to deal more directly with (for example) predicates. These problems exemplify the fact that the precise notion of logic underlying the specification language may never be exactly what we want for every situation. This can be addressed by parameterising the specification language by an arbitrary *institution* (Sannella and Tarlecki 1985). Institutions are a formalisation of the idea of logical framework. They will be introduced in Chapter 4.

The syntax of ASL specifications may be described as follows:

Spec	::=	Basic | Sum | Reach | Derive | Extend
Basic	::=	**sorts** *sorts* **opns** *partial operations* **axioms** *sentences*
Sum	::=	Spec + Spec
Reach	::=	**reachable** Spec **on** *set of sorts*
Derive	::=	**derive from** Spec **by** *signature morphism*
Extend	::=	**extend** Spec **by** Basic

The semantics of a specification S is given by its *signature* (Sig[S]) and the class of models of that signature which it admits (Mod[S]). A signature Σ is just

a collection of sort and operation names: $\Sigma = \langle \text{sorts}(\Sigma), \text{opns}(\Sigma) \rangle$. Notice the collection of operation names opns(Σ) is indexed by *arity*. The arity of an operation just gives the sorts of its arguments and result, for instance the arity of

$$\textbf{opn } \mathtt{f} : \mathtt{X}_1 \times \mathtt{X}_2 \times \ldots \mathtt{X}_n \dot{\rightarrow} \mathtt{X}$$

above is $\mathtt{X}_1 \times \mathtt{X}_2 \times \ldots \mathtt{X}_n \dot{\rightarrow} \mathtt{X}$. $\mathtt{X} \times \mathtt{X}$ may be abbreviated as $\mathtt{X}^2$, $\mathtt{X} \times \mathtt{X} \times \mathtt{X}$ as $\mathtt{X}^3$, and so on. The signature, Sig[SGP], of SGP (1.3) has sorts(Sig[SGP])={Gpd}, and $\text{opns}(\text{Sig[SGP]})_i = \{\_ \cdot \_\}$ for $i = \mathtt{Gpd} \times \mathtt{Gpd} \dot{\rightarrow} \mathtt{Gpd}$ but empty otherwise. A Σ-*model* M interprets each sort X by a set $[M\mathtt{X}]$, and each operation $\mathtt{f} : \mathtt{X}_1 \times \mathtt{X}_2 \times \ldots \mathtt{X}_n \dot{\rightarrow} \mathtt{X}$ by a (partial) function $[M\mathtt{f}] : [M\mathtt{X}_1] \times [M\mathtt{X}_2] \times \ldots [M\mathtt{X}_n] \dot{\rightarrow} [M\mathtt{X}]$. We say S′ *refines* S if Mod[S']⊆Mod[S]. If also S′ has only *one* model, up to isomorphism, it is said to *implement* S.

A *signature morphism* $\sigma : \Sigma \rightarrow \Sigma'$ is a map which takes every sort name $s \in \text{sorts}(\Sigma)$ to a sort name $\sigma(s) \in \text{sorts}(\Sigma')$, and every operation name $f_i \in \text{opns}(\Sigma)_i$ of arity i to an operation name $\sigma(f_i) \in \text{opns}(\Sigma')_{\sigma(i)}$. Note this definition involves extending σ from sort names to arities, but this is quite straightforward, for instance $\sigma(\mathtt{A} \dot{\rightarrow} \mathtt{B}) = \sigma(\mathtt{A}) \dot{\rightarrow} \sigma(\mathtt{B})$. In the same way σ maps any piece of syntax using Σ to a corresponding piece using Σ'. In this way, a signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ can be used to interpret any Σ' model M' as a Σ model $\sigma M'$ (or $M'|_\sigma$), the σ-*reduct* of M' (as for instance in Figure 1–2). To interpret any piece of syntax built out of Σ via this reduct model, first map that syntax into Σ' using σ, then interpret the result using M'. The simplest examples are when the signature morphism is an inclusion (or the identity), like the inclusion of Sig[SGP] in Sig[MON]. Any model for MON (1.5) can act as a model for SGP (1.3): the extra mechanics to interpret e are simply not used. In such a case, ($\Sigma \subseteq \Sigma'$), we write $M'|_\Sigma$ for the inclusion reduct.

Alg(Σ) denotes the class of all models (algebras) of the signature Σ. When a model M satisfies a sentence ϕ, write $M \models \phi$. The following equations give the semantics of the different constructs.

$$\text{Sig}[\textbf{sorts } S \textbf{ opns } F \textbf{ axioms } E] = \langle S, F \rangle$$

$$\mathrm{Mod}[\textbf{sorts } S \textbf{ opns } F \textbf{ axioms } E] = \{A \in \mathrm{Alg}(\langle S, F\rangle)\, s.t.\, A \models E\}$$

$$\mathrm{Sig}[\mathsf{T} + \mathsf{T}'] = \mathrm{Sig}[\mathsf{T}] \cup \mathrm{Sig}[\mathsf{T}']$$
$$\mathrm{Mod}[\mathsf{T} + \mathsf{T}'] = \{A \in \mathrm{Alg}(\mathrm{Sig}[\mathsf{T} + \mathsf{T}'])\, s.t.\, A|_{\mathrm{Sig}[\mathsf{T}]} \in \mathrm{Mod}[\mathsf{T}] \text{ and } A|_{\mathrm{Sig}[\mathsf{T}']} \in \mathrm{Mod}[\mathsf{T}']\}$$

$$\mathrm{Sig}[\textbf{reachable } \mathsf{T} \textbf{ on } S] = \mathrm{Sig}[\mathsf{T}]$$
$$\mathrm{Mod}[\textbf{reachable } \mathsf{T} \textbf{ on } S] = \{A \in \mathrm{Mod}[\mathsf{T}]\, s.t.\, A \text{ reachable on } S\}$$

$$\mathrm{Sig}[\textbf{derive from } \mathsf{T} \textbf{ by } \sigma] = \Sigma, \text{ where } \sigma : \Sigma \to \mathrm{Sig}[\mathsf{T}]$$
$$\mathrm{Mod}[\textbf{derive from } \mathsf{T} \textbf{ by } \sigma] = \{A|_\sigma\, s.t.\, A \in \mathrm{Mod}[\mathsf{T}]\}$$

$$\mathrm{Sig}[\textbf{extend } \mathsf{T} \textbf{ by sorts } S \textbf{ opns } F \textbf{ axioms } E] = \mathrm{Sig}[\mathsf{T}] \cup \langle S, F\rangle$$
$$\mathrm{Mod}[\textbf{extend } \mathsf{T} \textbf{ by sorts } S \textbf{ opns } F \textbf{ axioms } E] =$$
$$\{A | A \text{ freely extends some } A' \in \mathsf{Mod}[\mathsf{T}] \text{ by } S, F \text{ subject to } E\}$$

A basic specification selects all models of the given signature which satisfy its axioms. This is where we start: give some syntax and say what it does. $\mathsf{T} + \mathsf{T}'$ selects models of the combined sorts and operations which can be used as either T or T' models. This is useful in modular construction. For instance, we could write a specification BOOL of the booleans once and for all, and just add it into other specifications when required. For any specification T and set of sorts S, **reachable** T **on** S selects those models of T such that every element of the carriers for the sorts S is expressible as a term with only variables of sorts from $\mathrm{sorts}(\mathsf{T}) - S$. For instance, in any model of FGM (1.12), we can say that every string over a set (sort) `Sub` is expressible as a $\_ \cdot \_$-term with variables over `Sub`, and so the model is reachable on the set of sorts {`Gpd`}.

The three specification-building operations described in the preceding paragraph (basic specification, the "plus" construction, and **reachable**) all act in some sense as filters, though the first two filter all algebras of appropriate signature, and the third just those of the specification T. In contrast, the other two (**derive** and **extend**) have quite a constructive flavour, in that they are based on maps which

take an individual model, and transform it in some way. Sets of models are then mapped model by model, so that, for instance, a singleton set (a set containing just one model) will be mapped to another singleton set. This guarantees that applying them to a *consistent* specification (one which has a model) can never yield an *inconsistent* specification (one which has no models). This makes them very useful in specifying implementations (single models), since any specification constructed entirely from them must be consistent (provided, of course, that all extensions are well-defined). This need not be true of the first three operations.

derive from T **by** σ, for any specification T and signature morphism $\sigma : \Sigma \to$ Sig[T], is based on the σ-reduct construction, making each model of T into a Σ-model by first applying σ to any Σ-syntax which requires interpretation. This tends to be most useful in renaming, for example to distinguish multiple copies, and in "forgetting" syntax which is no longer required. Informally I may sometimes write $T|_\sigma$ for **derive from** T **by** σ, though this notation is properly reserved for models. Similarly $T|_\Sigma$ stands for **derive from** T **by inclusion of** Σ.

The construction **extend** T **by sorts** S **opns** F **axioms** E depends upon the formation of the free extension, which was described in Section 1.7.3. Roughly, we get this by (recursively) adding new objects to the carriers as required by F, identifying objects only when required by E. For instance, extending a sort `A` where $[\![\texttt{A}]\!] = \{0\}$ by a total function `f : B` $\to$ `B` and axiom `A` $\subseteq$ `B` gets us, up to isomorphism $[\![\texttt{B}]\!] = \{0, \texttt{f}(0), \texttt{f}(\texttt{f}(0)) \ldots\}$. As outlined in Section 1.7.3, **extend** cannot be used with arbitrary axioms. However Tarlecki (1984) shows that the free extension is well-defined if we limit ourselves to universally quantified conditionals of the form $\phi_1 \wedge \ldots \phi_n \to \phi$, where each ϕ is an equation or an inequation. In the presence of partial functions or predicates (as opposed to just total functions), hierarchical consistency requires that the tuples of existing objects assigned to any existing predicate, or those at which any existing partial operation is defined, are not changed by the extension. (If we only have total functions, this is obviously not a problem, and so we would only have to worry about additions to the equality predicate, i.e. equating existing terms.) We will write $\emptyset$ to abbreviate the unique empty specification, **sorts** $\emptyset$ **opns** $\emptyset$ **axioms** $\emptyset$. So, for instance,

extend $\emptyset$ **by**

sort Nat

opn 0 :⟶ Nat

opn S : Nat → Nat

implements successor notation for natural numbers. Because of this constructive flavour, **derive** and **extend** can be used to write specifications which can be viewed as *parameterised programs*, or *program modules*. A parameterised program is (essentially) a map from models to models. Such a map can be extended in an obvious way to map sets of models to sets of models (as a parameterised specification), but the converse clearly does not hold: a map from sets of models to sets of models need not map a singleton to a singleton. But those, such as PFGM (1.13), which do, may be viewed as parameterised programs. Parameterised specifications may be written using syntax borrowed from the lambda calculus. The specification $\mathsf{F} = \lambda\mathcal{X}{:}\mathsf{S_i}\,.\,\mathsf{S_o}$ is interpreted by the function whose value on any specification S refining $\mathsf{S_i}$ (written F(S)) is the interpretation of the specification obtained by substituting all free occurrences of $\mathcal{X}$ in $\mathsf{S_o}$ by S.

In future, I will write $S' \sqsubseteq S$ if $S'|_{\mathrm{Sig}[S]}$ refines S. So S' may contain some extra syntax, but in other respects every model of S' acts like a model of S. When the inclusion is strict ($\mathrm{Mod}[S'|_{\mathrm{Sig}[S]}] \subset \mathrm{Mod}[S]$), I may write $S' \sqsubset S$. So for example MON $\sqsubset$ SGP, since $\sqsubset$ now takes care of "forgetting" e. Then for a parameterised construction $\mathsf{F} = \lambda\mathcal{X}{:}\mathsf{A}\,.\,\mathsf{B}$, if $\mathsf{A}' \sqsubseteq \mathsf{A}$, let F(A′) be the same as

$$\mathsf{A}' + \mathsf{F}(\textbf{derive from } \mathsf{A}' \textbf{ by inclusion of } \mathrm{Sig}[\mathsf{A}])$$

This construction uses **derive** to "forget" any extraneous syntax, so fitting the argument supplied to F to its requirement specification A, and then adds it back in explicitly using the "+" construction. The **enrich** construction may be defined metasyntactically by

$$\textbf{enrich } \mathsf{T} \textbf{ by sorts } S \textbf{ opns } F \textbf{ axioms } E = \mathsf{T} + \textbf{sorts } (\mathrm{sorts}(\mathsf{T}) \cup S) \textbf{ opns } (\mathrm{opns}(\mathsf{T}) \cup F) \textbf{ axioms } E$$

where $\cup$ represents set union.

1.9 Montague, ADJ, and Initial Algebra Semantics

Richard Montague is perhaps best known for work such as Montague (1973), in which he supports his claim that there is no fundamental difference between natural language and formal language by giving a formal syntax for a subset of the words, phrases and sentences of English, and providing for this syntax a formal, model-theoretic interpretation, intended to capture some of the everyday meaning of those words, phrases and sentences. Slightly less well known is work such as Montague (1970). Here, Montague defines a system of *Universal Grammar* (UG), by which he means, a framework within which grammars in general may be defined.

Algebraic specification in computer science starts with the work of the ADJ group (e.g. Goguen, Thatcher and Wagner 1978). The ADJ group had the insight that by associating programs and algebras, one abstracts away from the details of calculation. An algebra associates operation names only with functions, not with algorithms. The functionality of a program is what is important in using it. If this can be specified once and for all, we can change the underlying algorithm at will without harming any user of the program. So one needs to be able to specify a single algebra (or isomorphism class thereof) with the required functionality. The ADJ approach to this was to give term equations which describe the required properties. But there will be many algebras which satisfy these equations, not all isomorphic. The algebra (actually, isomorphism class) chosen was the *initial* algebra, that which assumes the least entities subject to equating the fewest terms. A program is then said to implement the specification if it supplies algorithms with this functionality.

Janssen (1983) suggests (and I would concur here) that the approach of Montague (1970) to specification does not differ in essence from that used by the ADJ group, except that the former was developed primarily with natural language in mind, and the latter with computer programs.

As noted, Montague is also interested in producing plausible grammars within his framework, such as Montague (1973). Like the ADJ specifications, these are really just single algebras. Even if one accepts that this is ultimately the right level of description, insisting that all specifications correspond to a single algebra limits the sort of descriptions by which this goal may be approached. For instance, grounding specifications as *classes* of algebras it becomes much easier to break up the notion "grammar" into treatments of the various linguistic phenomena, for instance into some "core" grammar, treatments of relatives, co-ordination, and so on. This also opens the possibility that we may write specifications designed to say in the most general terms how a treatment of (say) relatives may be got from a simpler grammar with certain minimal features. Such a specification may encompass the approaches of a great many different styles of grammar to such a problem, and is thus a further abstraction away from actual details of "program", or construction. The point of this abstraction, as before, is an attempt to describe what it means to be a solution to the problem, other than by saying, "here is a solution".

Montague provides a space of grammars (UG) and some examples of grammars in that space, but provides no means of carving out subspaces, or classes, of grammars. Chomsky (1986) uses the term *Universal Grammar* to refer to a system which describes all and *only* the (idealised, potential) human languages. Montague's UG framework is clearly capable of describing systems bearing little resemblance to human language, and indeed was never claimed to describe *only* potential human languages. Thus it is perfectly reasonable to use this framework to describe computer programs, which do not appear likely candidates for admission as potential human languages. Nevertheless, we can imagine some *restriction* of Montague's framework corresponding to Chomsky's conception of Universal Grammar. But there is no way of describing this restriction *in the language of the framework*, nor any means of explicitly referring to cross-linguistic phenomena, because the framework only describes single models. What is missing, from both Montague's UG, and the work of the ADJ group, is any means of referring to (non-isomorphic) classes of models. Even though it may be reasonable to as-

sociate the division of a program into modules with the construction of a larger algebra from smaller ones, conceptually it may be easier to break the description of an algebra up by describing a series of classes to which it must belong. This is *stepwise refinement*, by which we may form successively more specific specifications as we flesh out the important behaviour of a system (Sannella 1986). This leads us to ground a specification semantically as a class of (not necessarily isomorphic) algebras. This style of semantics is sometimes called *loose* specification. This class-of-algebras, stepwise refinement approach is more closely aligned with the idea that the task of linguistics is to *constrain* the class of possible natural language grammars. This framework also simplifies somewhat the discipline of writing down as much as possible about one's notion of the problem at every stage of investigation, which is a good way of sharpening one's perception of what has been achieved, and what has not, and for communicating precise, formally worked results at a highly abstract level.

Montague (1970) sets up the mechanics of a particular logical framework, which Montague hoped would suffice for all linguistic/semantic and philosophical/mathematical endeavour. Janssen extends the claim to computer science. However Janssen also generalises Montague's logic somewhat, suggesting already the question of whether there can be a single right logic for all things and all time. This setting up of a logical framework corresponds to defining an *institution*. Goguen and Burstall (1985) describes "the notion of an institution as a precise generalisation of the informal notion of logical system". It is Goguen's thesis that the main difference between very many specification languages is just the logical system in which they are operating. These includes systems of equational logic (as used in Universal Algebra), Horn clause logic (as used in Pure Prolog), first order logic with or without equality, and various restrictions thereof, as well as systems which admit partial operations or order-sorted signatures. This phenomenon is surveyed in Goguen (1987). His conclusion is that there can be no single *right* choice of logical framework. It is possible to parameterise a specification language (such as ASL) by an institution, so that one in fact has a whole family of specification languages, which can be used together using operations like

change institution (Sannella and Tarlecki 1988). So there is no need to limit oneself to a single logical language, even within one specification.

Thus I see two main deficiencies in Montague's UG as a framework for linguistic endeavour. The first (as with the ADJ framework) is that our ability to abstract across grammars, and to manipulate these abstractions, is limited by the insistence on initiality of models. The second is that there is no reason why we should limit ourselves forever to any single logical language.

1.10 Other Styles of Specification in Computer Science

The preceding section was devoted to some reasons why I feel the grounding of specifications as *classes* of models is a useful tool for the *specifier* of a large information system (computer system, or natural language system). It also suggests that grounding a specification as a *single* model is entirely reasonable from the point of view of the programmer, or more generally, the implementor of (any part of) such a system. Furthermore, the specifier can look at the particular specification language, and the programmer at the particular programming language, as being embodied by the particular logic (institution) in which this grounding is pursued. In practice, these two jobs are quite likely to be performed by the same individual(s), and this is no bad thing, since the properties of an implementation should suggest improvements in the specification no less than the other way around.

However in Pereira and Shieber (1984) we see the semantics of a language like PATR-II described, not as an institution, but in terms of what has become known as *domain theory* (Stoy 1977). But this style of specification is motivated not by the needs of the specifier of a system, or an implementor of a program involved in that system, but by the specifier of a programming language, or the implementor of a compiler or interpreter for such a language. Taking a step back, it is obvious that a compiler or interpreter *is* a program — so why shouldn't we handle their

specification and implementation in just the same way we have been advocating for other programs?

And there is no reason why we should not. Domain theory is essentially just a specialised kind of model theory, and the definition of domains appropriate to a language is just the definition of the appropriate classes of models for an institution of that language. The constructions chosen in such work have many special properties which make them suited to procedural interpretation, but this doesn't stop us taking a pre-procedural perspective on them. We have seen some hints already (Section 1.5) as to how the model theoretic perspective can be tied to the procedural (proof theoretic), by looking for some routine system of calculation (proof theory) capable of answering questions such as whether any/all of the models of a given specification satisfy some given sentence of the language. In particular, any given definition of model morphism preserves satisfaction of a particular class of sentences, and thus if there is an initial model relative to such a definition, it satisfies such a sentence exactly when every model does so. This does *not* mean that deciding whether the initial model satisfies a given query sentence is necessarily routine, though this may be true for particular kinds of query, such as conjunctions of atomic formulae, existentially quantified (as in Prolog).

Nor is it the case that the over-arching theory of how the different forms of semantics slot together is well-established and completely understood. As with any other scientific endeavour, the elucidation of a particular perspective or modelling construction feeds back into our understanding of the overall task. In this case I use the term "modelling construction" very loosely, to say that both the model theoretic and the proof theoretic approaches to specification and semantics can be said to be about "modelling" a formal language. Domain theory is one technique which offers some promise of elucidating the connection between the two perspectives. In particular it is part of the trend to see them as different manifestations of a more abstract semantics. Closely linked to domain theory in this trend is the more sophisticated use of categories (in the sense of category theory) as domains of interpretation. Also related are the use of type hierarchies, such as that of Martin-Löf (1982).

1.11 Real Programmers

One day, perhaps not too far off, we may understand enough of this connection to make the use of such techniques a practical background for the development of large computer systems. In the short term, however, it seems to me that that an ASL-style class-of-models approach offers the best hope for improving the way we specify and implement large systems. At the moment, more theoretical effort goes into the proof theoretic perspective, and this is probably just as well, as more "real" (commercial) system specification and implementation is done in algebraic-style model theoretic vein. This should not be taken to mean that its use is widespread, but more that the use of algebraic style specification in commercial environments has begun, and little else really has even done that. For an informal overview of the philosophy and research programme behind ASL, see Sannella and Tarlecki (1992).

One should not, however, imagine from this that specification language inventors also invented the problem they aim to solve! The problem of how to go about the specification and implementation of large computer systems is enormous. It generates and consumes huge amounts of money and effort. There exist a vast number of proprietary "systems" which companies can purchase in an effort to ease this task. While these will generally offer some forms to follow in the process, they tend to be as much about management (costing, delegation and co-ordination of personnel) as anything else. Computer system development seems to lead to difficulties in management in a way which previous engineering tasks have not.

Part of the reason is certainly the difficulty of semantics. It is clear that there is more to the semantics of programming than knowing the result of machine instructions. The specifier/programmer has some idea of what needs to be achieved and what this implies for the program. She has besides this an array of techniques and tricks for coming up with an actual program which does these things. The investigation of such a program may give her a more accurate idea of what the task entails, and so on.

However as soon as the task becomes so large that it cannot be carried out by one individual, we see the deficiencies of carrying out this process in an informal way. Misunderstandings develop. Subsystems fail to fit together as expected. For instance, one person may make a broad specification which breaks the task down in a conceptually clear fashion into two subtasks, only to have the (plural) implementors discover that this conceptually clear division is not the appropriate one for implementation. There may be no way to avoid this, but what we *can* do is try to ensure that we discover it at the earliest possible point, by producing simple implementations at an early stage. Even if these suffer known deficiencies, they may be enough to show us that the current division of the task is not appropriate.

One of the difficulties in such a process is the language the various individuals should use to communicate their work. Usually, it is natural language, sometimes helped and sometimes hindered by the sort of proprietary systems alluded to. How widely a formal language will ever become used is debatable, but at least it has proved itself useful in some cases. For instance, it is only quite recently that the correspondence was noticed between, on the one hand, the sort of conflict just outlined, where the appropriate divisions of a task for specification and implementation differ, and on the other, the distinction between specifications of parameterised programs, and program specifications which are themselves parameterised (Sannella and Tarlecki 1988). For a description of the practical application of loose specification and stepwise refinement, see Sannella (1989).

1.12 Real Linguists

It will be noted that the overwhelming majority of my examples of "real" linguistic frameworks are in the feature-value tradition. Of course it is true that these frameworks have developed in intimate association with the burgeoning field of Computational Linguistics, and so one would expect them to be more liable to fit into the context of a theory borrowed from Computer Science. I have little doubt that fitting other frameworks into this setting could be a much more difficult

task, but the need to do so is all the more urgent. The fact that the ideas arise from Computer Science does not mean that concerns of executability need be paramount, since in any case much of the point of algebraic semantics is that it abstracts away from procedurality.

Just as formal specifications may never be more than one tool of many employed in trying to keep a software project on course, they are never likely to be the whole story of linguistic endeavour. For instance, social factors are bound to influence either. But this has not stopped specifications being of use in conveying certain kinds of information in developing computer systems, and they can be used in much the same way in linguistics. A large computer system, incorporating input from many directions (perhaps communication over "noisy" lines, perhaps real-time monitoring of physical systems) is itself subject to vagaries beyond what goes on in the silicon, and cannot be thought of as a mere program. But it has parts which are programs, and these parts must cope in some way with the unpredictability of the other parts, which might remain only loosely specified. There is some parallel between the ideas of graceful degradation, and the performance/competence distinction.

Anyone who has written a fairly large program (or even more so, been one of many participants in such a task), will know how easy it is for unconsidered and apparently trivial details to turn up in the most unlikely places, revealing an aesthetically splendid program to be, in fact, useless. Again, whether a model is running on a computer is not of primary concern (though having a computational implementation often makes deficiencies easier to spot). The same phenomenon will beset anyone dealing with formally precise model construction, including, for instance, workers in the tradition of Montague grammar, or in the semantics of feature-value formalisms. If someone suggests a possible problem in such a construction, it is generally quite obvious whether it is a problem in fact. It may take some time to identify the real source of the problem, and solving the problem may mean anything from a simple correction to a complete redesign, but the status of the claim as a genuine "bug" is rarely in doubt.

In contrast, linguistic arguments (notably in the *Government and Binding* the-

ory (GB) of Chomsky (1981)) very often seem to involve considerable equivocation over whether a putative difficulty has the status of a genuine deficiency of the analysis. For instance, such an equivocation might take the form "perhaps if we assumed a treatment for case theory which XYZ and a theory of theta-roles such that ABC...". Papers are liable to begin "assume some form of X-bar theory", yet there appears to be no standard definition of what it means to be a (generic, rather than specific) X-bar theory: only some example theories, which in turn may be built on similarly imprecisely grounded notions. Even if we take it that the *only* X-bar theories (or whatever) are say a half dozen well-known versions, if we multiply this out for all the other variables in the system, the result is simply more than we can adequately cope with in an informal way. This is evidenced by precisely the sort of equivocations just alluded to. To many people who have worked with large formal systems, this ability to pull out of the air a "tweak" to cover practically any potential problem which is pointed out is deeply suspicious. How many variables have simply been neglected? It is entirely possible, for instance, that this tweak is inconsistent with every possible setting of such an unconsidered variable, in very different and unobvious ways. Can we be sure that this tweak is compatible with the one suggested last week or last year?

It is essential that an analysis be *falsifiable*: that we should be able to tell unequivocally when it truly describes an observation of the phenomenon under study, and when it does not. In a complicated, informally presented system, such an argument is never likely to be unequivocal, even to experts in the system. Contrast this to a (more-or-less) formally presented system: for instance Kaplan and Zaenen (1987) is a description of long distance dependencies in LFG (one of the feature-value systems I will consider). This analysis suffers the following deficiency: their rule (71) which describes the modification of a noun phrase by a relative clause is, as given there, insufficient to force any feature other than RELMOD to appear on the mother NP. The obvious fix (add $\uparrow=\downarrow$ to the decoration on the daughter NP) would cause cyclity in the model, which is explicitly ruled out elsewhere in the paper. Even with little or no background in LFG, it is possible to spot such a deficiency, and formally trace a proof of inconsistency, in such a way

that it cannot be denied. It is unlikely that an inconsistency could be so readily demonstrated in an informally presented system.

I do not wish to appear dismissive of the programme of GB. In many ways it embodies just the sort of characteristics one would look for in an attempt at abstract specification. For instance, in the *principles and parameters* approach to the study of language (e.g. Reape and Engdahl 1990), which has become central in GB (and elsewhere), the idea of constraining the class of possible languages by the imposition of various *principles* has a clear counterpart in the idea of stepwise refinement. For another example, the attempt to draw a strong line between a fixed Universal Grammar and various language-dependent variables is clearly reminiscent of the distinction in logic between logical and non-logical syntax, and suggests it may perhaps ultimately be viewed as an institution. But it seems that only canonical models (trees of some sort) are ever defined, and then a few variations on them are considered. If one does not define a class of models in which to move, this can never be more than grasping in the dark. Any claims that it is too early to consider such a move are long since void. The system is far too complex to be dealt with in this way. If it turns out that a choice of model class is insufficient to the task, so be it. That too is subject to review. But I am convinced that the example of computer systems amply illustrates the folly of specifying ever more properties a system should have, without ever stopping to ground it in a precisely defined implementation.

1.13 Coda

With these background preliminaries complete, I shall move on to consider how ideas of modularisation, parameterisation, loose specification, and stepwise refinement may be useful in describing models for linguistic theory. In Chapter 2, I will consider why the desire to model constituency leads to the introduction of an auxiliary, or intensional, sort. We shall see some simple examples of grammatical models. In Chapter 3, I consider the need to classify this intensional domain along

several different dimensions, and how this can be done using features. I discuss PATR-II, LFG, GPSG and HPSG. In Chapter 4 I introduce "the notion of an institution as a precise generalisation of the informal notion of logical system" (Goguen and Burstall 1985). The **change institution** operation can be used to link together specifications written using different institutions (logics). In Chapter 5 I develop an abstract specification of what it means to treat the phenomenon of topicalisation, and develop a parameterised implementation which, given a core treatment with certain minimal properties, will yield a simple extension which treats topicalised sentences. I then present a similar two-pronged development, of an abstract and a concrete specification, this time for "wh"-relative clauses. In Chapter 6 I consider a simple example of the use of loose specification and parameterisation in developing a treatment which admits different refinements for modelling different languages (in this case, dependent clause order in English, German and Dutch). In Chapter 7 I present some conclusions and directions for further study.

Chapter 2

Grammars

Algebraic-style specifications may be a useful tool in the writing of natural language grammars. Grammars attempt to model human languages, or parts of them. As with programs, there may be many equivalent ways to implement essentially the same account. For instance, much as one may choose from a range of programming languages, there are many grammatical frameworks in which a grammar may be constructed. We might employ *context-free rewrite rules* (Hopkin and Moss 1976), such as S $\longrightarrow$ NP VP (a sentence is (or may be) a noun phrase followed by a verb phrase), or we might employ the *categorial slash* of Lambek (1958) and replace VP by the compound term S\NP (something which looks for a noun phrase to the left in order to form a sentence). Within a framework there are still many ways of achieving the same goal: for instance we could equally have replaced NP by the compound term S/VP.

The principles and parameters approach to the study of linguistic structure has become ubiquitous over the last decade. The *principles* part is easily set into the discipline of stepwise refinement. It is a philosophy of applying successively more constraints to a system, slowly narrowing down to the required behaviour, eliminating possible behaviours as we go, in just the way that stepwise refinement filters away more models at each step. The ideal is that through this process, the range of possible grammars should become so constrained that variation across languages should be explicable in terms of the values taken by a few *parameters*, or variables, of the system. For example, in one of the seminal works for this

approach, Rizzi (1982) suggests that what appear superficially to be quite different restrictions on the syntax of so-called *wh-constructions* (such as *I wonder who the jury will blame*, or *the actor who no-one remembers*) in English and in Italian, can in fact be viewed as the same restriction, *parameterised* by a variable set of *bounding nodes*, which take on slightly different values for the two languages. We can imagine that this might ultimately be characterised by a specification like

(Universal Grammar)

$$\mathsf{UG} \quad = \quad \lambda\mathcal{B}{:}\mathsf{BOUNDSETS}\cdots.\,\mathsf{WHX}(\mathcal{B}) + \textit{other subspecifications}$$

where WHX is a specification of the wh-construction, parameterised by a set $\mathcal{B}$ of possible bounding nodes, and UG is a parameterised specification built up from WHX and specifications of the various other linguistic phenomena, such that, when supplied values for $\mathcal{B}$ and whatever parameters the other subspecifications may require, UG produces a specification of a single model (grammar) of a particular language.

Many theories propose independent "levels" of representation (c- and f-structure in LFG, S-structure, LF and so on in GB and its ancestors: see for instance Sells (1985)), which can only "communicate" in given ways (via rules or relations). This is very reminiscent of the central idea of modular program construction, where a task is divided into various subtasks, performed by separate modules which communicate only by mutual invocation of those items explicitly made visible to the outside.

In attempting to extend coverage, it is quite common to assume some "core" theory, which one then builds upon. For instance we could view the metarules of GPSG (Gazdar et al. 1985) as extending a core of explicitly given phrase structure rules. This extending construction is more or less independent of the particular phrase structure rules used. Such constructions are just another form of parameterised description. (The GB notion of "core" grammar, which is only intended to exclude *marked*, more-or-less one-off constructions, is a considerably more restrictive use of the term than I employ here).

Certainly there seem to be enough parallels between the linguistic task and the software engineering task to make further investigation worthwhile.

2.1 Strings, Substitution and Constituency

I shall consider grammars of sentence structure, which operate at the level of how sentences may be built up from words. Models must contain (at the very least) objects to correspond to words, an operation to correspond to forming sequences of words, and some means of identifying which of those sequences are well-formed strings of the language. I want first to construct a specification we could use to type a parameter which can be thought of as standing for some "core" grammar (for instance covering ordinary subject-verb-object type sentences), which we can then extend (for instance to cover relative clause constructions). In order that this parameter be able to range as widely as possible, we should require of it the minimal structure which still allows us to perform the necessary constructions upon it. Thus I shall begin by aiming just at well-formedness. This is really no restriction, since models involving more sophisticated interpretative domains and constructions are certainly still among the class which deal with well-formedness. As with the formation of HIDEID (1.6) from MON (1.5) above, we should just have to "fit" them to the parameter, by using the **derive** construction to "forget" those parts we don't require. Separating a semantic domain (like the value of the SEM attribute in Pollard and Sag 1987) which deals with (something like) the everyday sense of meaning, and excludes everything else (word order, phonology and so on) is a convenience which marks it off for future use, say in semantic processing. In the context of building a specification at the most general level, there is no need to make such a separation.

(2.1) LEX* = **derive from** FGM
by [Lex $\longmapsto$ Sub, Lex* $\longmapsto$ Mon]

(Well-formed strings)

WFS = **enrich** LEX*
by **sort** `Wfs`
axiom `Wfs` $\subseteq$ `Lex`$*$

We use FGM (1.12) to form strings. Note [`Lex`$\longmapsto$`Sub`,`Lex`$*$ $\longmapsto$`Mon`] represents a signature morphism which maps the sort names `Lex` and `Lex`$*$ to the sort names `Sub` and `Mon` respectively, but changes no other name. At first it seems counter-intuitive that [`Lex`$\longmapsto$`Sub`] is used to change a model of "**sort** `Sub`" to one of "**sort** `Lex`" (which is what it is doing). The morphism seems to be going the wrong way! If you think about it though, in order to form a model of "**sort** `Lex`" we must interpret the sort name `Lex`. Now if all we have is a model M interpreting the sort name `Sub`, one way to go about interpreting `Lex` is first to map it to `Sub`, and then interpret this using M (see again Figure 1–2). So [`Lex`$\longmapsto$`Sub`,`Lex`$*$ $\longmapsto$`Mon`] is the correct morphism for the above renaming. We compose the signature morphism and the mapping of M from syntax to semantics to form the reduct $M|_{[\texttt{Lex}\longmapsto\texttt{Sub},\texttt{Lex}*\longmapsto\texttt{Mon}]}$. So all this was just to form strings of words. Now we observe that only some of these strings are well-formed strings. In summary, we denote the set of well-formed strings (of some language) by the sort name `Wfs` and insist that every such string is a string of words.

Perhaps the fundamental observation in attempts to characterise sentence-level grammar is that there is some interchangeability relation, whereby related words may be substituted one for the other without changing the well-formedness or otherwise of a sentence. The constituent tradition proceeds to generalise this by noticing that there are strings of words which, in a given context, may likewise be interchanged. For instance, the introductory linguistics textbook Radford (1988) lists as the first diagnostic under "Testing the Structure" (p90),

> Does it have the same distribution as (i.e. can it be replaced by) an appropriate phrase of a given type? If so, it is a phrase of the relevant type.

Of course, this is only meant to be one diagnostic in the context of a particular system, as witnessed by the fact that it cannot be applied without knowing some types and corresponding phrases. The best generalisation I can manage across grammars of how substitution is done is to say that if we conclude that $x \cdot s \cdot y$ is well-formed, partly in virtue of s being a well-formed string with some particular characteristics, and these characteristics are shared by another well-formed string s', then we can likewise conclude that $x \cdot s' \cdot y$ is well-formed. Although notions of what it means to be a constituent vary very widely, the idea that the parsing of a constituent contributes to the parsing of the whole can generally be seen there in some form or other, and may be used more widely than in just substitution. Another way of looking at this is to say s is well-formed in the context of x and y. "Well-formed in the empty context", "well-formed in some context", or simply "well-formed" are then all equivalent. I won't be considering discontinuous constituency here, so such a relation should correspond to some subrelation of the substring relation.

Different models go about describing characteristics of strings in very different ways. It seems appropriate to begin by supposing that we do not need explicitly to represent such details — they would be part of the construction which is "derived away". Can we then simply say that constituency *is* a subrelation of substring? Let's try it and see.

enrich WFS **by**

opn $\_[\_]\_ : \texttt{Lex*} \times \texttt{Wfs} \times \texttt{Lex*} \dot{\rightarrow} \texttt{Wfs}$

axiom $u = x_1[u_1]y_1 \rightarrow u = x_1 \cdot u_1 \cdot y_1$

$\_[\_]\_$ is supposed to stand for some version of parsing constituency, as discussed above. We need some sort of index on subconstituents, since the same string may appear as a subconstituent more than once. This is what x, y are doing in $x[u]y$. So, given any well-formed string u and word strings x and y, if $x[u]y$ is defined, then its value is the well-formed string $x \cdot u \cdot y$. This is supposed to express u being a subconstituent of $x \cdot u \cdot y$. The sense of "constituency" is very weak. It could be any subrelation of the substring relation. $x[u]y$ is not meant to say exactly

how x and y contribute to the parse, only that there is some process by which the string u may be built up into the string $x \cdot u \cdot y$, i.e. that u is a stage in a parse of $x \cdot u \cdot y$. At this stage we have not even claimed that a constituent can necessarily be expressed as a string of (immediate) subconstituents, let alone that such an expression would be unique.

I want now to claim that there is a problem with making these definitions simply in terms of strings. As stated previously, what I was trying to specify was a notion of subconstituency which could be used as an argument type, like saying in English "take some notion of subconstituency and augment it as follows", except that I try to say precisely what I mean by "some notion of subconstituency". I try to keep that notion as weak as possible, so that the parameter can range as widely as possible over specifications which might act in its place in the augmented construction, i.e. so that the augmentation is applicable to as many models (or classes of models) as possible. In this spirit, I say essentially we are talking about a substring, and where it occurs.

Here is where equating well-formed strings with strings over **Lex**, and trying to deal with subconstituency just in terms of these strings, causes trouble (despite the fact that I am free to introduce other domains in construction), for _[_]_ has no way of distinguishing between different readings of the same string.

Consider the ambiguous sentence *I saw the man with the telescope.* In one reading the man has the telescope, in the other, I am looking through the telescope at him. In many grammars, this ambiguity is evidenced by having two constituency structures, one for each reading. In the first *the man with the telescope* is a constituent, in the second, it is not. For instance, in the first reading we may describe *the man with the telescope* as a noun phrase, and use this characterisation in deciding that *I saw the man with the telescope* is a sentence. In the second reading, we might decide that *saw the man*, and hence *saw the man with the telescope*, is a verb phrase. We say nothing about *the man with the telescope.*

Such a grammar cannot be described by _[_]_ as it stands, simply because the term

I·saw[the·man·with·the·telescope]e

cannot be simultaneously defined and undefined. Formal languages, such as first order logic, are usually designed to be unambiguous, even if achieving this means imposing explicit brackets (e.g. $\phi \wedge (\psi \vee \zeta)$) or precedence conventions. The language we read or hear clearly does *not* contain such disambiguation, because the ambiguity persists. The syntax of the language we read or hear is basically a sequence of words (albeit partly disambiguated by factors like punctuation or intonation). The preceding argument shows that with this definition of syntax, subconstituency is not a purely syntactic notion. It becomes necessary to introduce into the *metalanguage*, some syntax of disambiguation. Montague (1970) deals with this by the use of a relation between expressions of the ambiguous language, and those of an unambiguous one. This is equivalent to adding bracketings to, or imposing a tree structure on, the ambiguous language. Essentially these are records of the different parses of a string (Janssen 1983). They are a static representation of a procedural notion. Subconstituency may well be a syntactic notion in terms of such a disambiguated language, but the addition of brackets is a step on the way to interpretation, and is therefore a semantic operation. However it is usual in linguistics to refer to the objects of this new domain as giving the syntax of the well-formed strings.

2.2 Intension and Grammar

When we set up a system to describe some phenomena, we first give ourselves syntax for the parts of the model which are to correspond to what we can observe: in this case, words, their sequence in strings, and the well-formedness of those strings. Models which can be described with this syntax alone are called *extensional.* It turns out (as we have just seen) that extensional models of this syntax are not adequate to capture constituency. In a similar way, in modelling the *meaning* of natural language, we find that modelling nouns as objects and verbs as predicates does not give enough metalanguage to describe completely models which capture

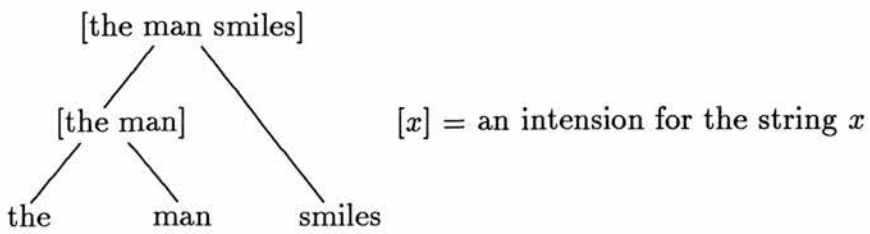

Figure 2–1: Possible parses can be represented as trees

entailments involving concepts like belief. Models which do this will need to have a more complex construction, and a metalanguage for describing them will need to be capable of detailing how observable entities might appear within this more complex construction. The extra level of complexity is sometimes called an *intensional* dimension. Some such intensional dimension is needed in order to capture syntactic constituency and ambiguity, and our specifications will need to establish some vocabulary by which we can refer to this dimension. We will replace `Wfs` by a disambiguating domain `Syn` of syntactic *intensions*. Some examples of the sorts of domains used for this purpose are bracketed strings, category symbols (S, NP...), terms (S\NP), and various graph structures (most notably trees). An intension for a string will correspond roughly to a parse of the string. It must record at least whatever information from the parse is relevant to its incorporation as a sub-parse of a larger string. This may or may not be all it records.

When we say *the man* is a constituent of *the man smiles*, this is shorthand for saying that there is a parse of *the man smiles* which has as a sub-parse a parse of *the man*. There are at least two obvious and interdefinable ways of representing the parsing relation: parse in one step, sometimes called *immediate constituency*, or parse in zero or more steps. For instance, in *the man smiles*, for most grammars of English, *the man* is an immediate constituent, because there are no constituents properly containing *the man* and properly contained in *the man smiles*, but *man* would not be an immediate constituent of the sentence, because it is properly contained in the intermediate constituent *the man*. Possible parses can of course be represented by trees (Figure 2–1), in which intensions are nodes, and one-step parse is given by daughterhood.

I shall use the sort name `Syn` to stand for syntactic intensions, and `Syn*` to

stand for strings thereof. The predicate $r \mapsto x$ I will use to stand for immediate constituency, where r is the mother and x is the sequence of daughters. Somehow the intensional domain must be related to the lexical domain **Lex**. For simplicity, I shall also use $\mapsto$ for this, so I must make **Lex** a subsort of **Syn**.

(2.2) (Lexicon)

LXN = **enrich sort** Syn
by sort Lex
axiom Lex $\subseteq$ Syn

(2.3) (Syntactic Intensions)

SYN* = **derive from** FGM
by [Syn $\longmapsto$ Sub, Syn$*$ $\longmapsto$ Gpd]

(2.4) (Grammar)

GMR = **enrich** SYN*
by pred _ $\mapsto$ _ $\subseteq$ Syn $\times$ Syn$*$
axiom $\neg s \mapsto s$

The transitive (zero or more step) relation is the one we require to talk about well-formed substitution (I shall refer to this relation simply as constituency): for instance, we may replace the noun *man* in the sentence *the man smiles* by such strings as *child*, *old man*, and so on. From the above irreflexive, parse-in-one-step, immediate constituency relation _ $\mapsto$ _ we can define the zero-or-more-step relation _ $\mapsto^*$ _.

(2.5) PARSE = **extend** GMR **by**
pred _ $\mapsto^*$ _ $\subseteq$ Syn$*$ $\times$ Syn$*$
axiom $x \mapsto^* x$
axiom $x \mapsto^* x' \wedge y \mapsto^* y' \rightarrow x \cdot y \mapsto^* x' \cdot y'$
axiom $r \mapsto x \wedge x \mapsto^* x' \rightarrow r \mapsto^* x'$

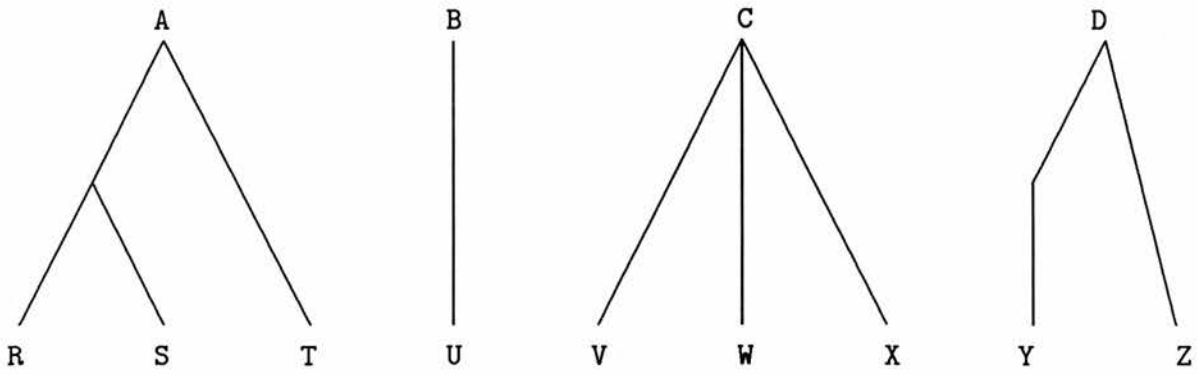

Figure 2–2: A·B·C·D$\mapsto^*$R·S·T·U·V·W·X·Y·Z

Note _ $\mapsto^*$ _ also extends _ $\mapsto$ _ across _ · _, from **Syn** to **Syn***. $r \mapsto x$ represents a local tree with the intension r as the mother and the intensions x as the daughters. $x \mapsto^* x'$ represents the existence of a sequence of parse trees with the intensions x along the roots and x' along the leaves. This is illustrated in Figure 2–2. The idea of constituency is now expressed relative to a parse. We will say s is an immediate constituent of r if r, s in **Syn**, and there exist x, y in **Syn*** such that $r \mapsto x \cdot s \cdot y$. Note that x and y may correspond to a sequence of intensions, or even to **e**. Replacing $\mapsto$ by $\mapsto^*$ gives us the definition of (eventual) constituent.

Because the **extend** operation of PARSE (2.5) does not introduce any new functions, and because the axioms have the form of universally quantified conditionals whose consequents involve no predicate which was part of the specification being extended (not even equality), the extension cannot create new objects of old sorts, nor change the assignment of existing properties (equality included) to existing objects. That is, the extension is sufficiently complete, and hierarchically consistent, and therefore persistent.

Now in any implementation S of LXN + GMR (2.2, 2.4) in which _ $\mapsto$ _ is finite, **Syn** and $\mapsto$ can be viewed as giving, respectively, the atomic categories and context-free rewrite rules of a simple phrase structure grammar (PSG; see for instance Hopkin and Moss 1976). For example the grammar

VP $\longrightarrow$ VT NP
VP $\longrightarrow$ VT NP PP

NP $\longrightarrow$ NP PP

NP $\longrightarrow$ the man

VT $\longrightarrow$ saw

PP $\longrightarrow$ with the telescope

is equivalent to the specification S below

$\mathsf{LXN_S}$ = **extend** $\emptyset$ **by**

sorts Lex,Syn

axiom Lex $\subseteq$ Syn

opns saw, the, man, with, the, telescope $:\rightarrow$ Lex

opns VP,VT,NP,PP $:\rightarrow$ Syn

(2.6) S = **derive from**

extend $\mathsf{LXN_S}$ + SYN* **by**

pred _ $\mapsto$ _ $\subseteq$ Syn $\times$ Syn*

axioms VP$\mapsto$VT·NP, VP$\mapsto$VT·NP·PP, NP$\mapsto$NP·PP,

PP$\mapsto$with·the·telescope,

NP$\mapsto$the·man, VT$\mapsto$saw

by inclusion of Sig[LXN+GMR]

references: LXN (2.2), GMR (2.4)

where the effect of $\mathsf{LXN_S}$ is just to specify sorts containing the required objects, and no more. Every model of S is isomorphic to the model S which has

(2.7) $[S\texttt{Lex}]$ = {saw,the,man,with,telescope}

$[S\texttt{Syn}]$ = $[S\texttt{Lex}] \cup$ {VP,VT,NP,PP}

$[S\_ \cdot \_]$ given by concatenation

$[S\_ \mapsto \_]$ given by

VP $[S \mapsto]$ VT NP

VP $[S \mapsto]$ VT NP PP

NP $[S \mapsto]$ NP PP

NP $[S \mapsto]$ the man

VT $[S \mapsto]$ saw

PP $[S \mapsto]$ with the telescope

If I had used **enrich** rather than **extend** above, I would have been allowing models with extra (unreachable) objects, as well as models which map two or more constants to the same object, but this was not the intention in the given PSG.

I shall digress briefly to consider another class of models, and how to specify them. This time, $\_ \mapsto \_$ will *not* be finite. They are the so-called *categorial grammars* (see for instance Lambek 1958), in which, besides atomic categories like S and NP, we have categorial terms formed by (recursively) combining the atomic categories using / and \. S\NP is the intension given to a string which will form a sentence when preceded by a noun phrase, for instance a verb phrase. (S\NP)/NP is the intension given to a transitive verb, since it forms a verb phrase when followed by an NP. As with PSGs, there is a natural class of models in addition to the term models, constructed from sets of word strings of the various classes.

(Freely Generated Category Algebras)
$\mathsf{CAT_F}$ = $\lambda\mathcal{X}$:**sort** Basic.
extend $\mathcal{X}$
by **sort** Syn
axiom Basic $\subseteq$ Syn
opns $\_\backslash\_, \_/\_$: Syn $\times$ Syn $\rightarrow$ Syn

(2.8) (Simple Applicative Categorial Grammars)
ACG = **extend** SYN* + $\mathsf{CAT_F}$(**sort** Basic)
by **pred** $\_ \mapsto \_ \subseteq$ Syn $\times$ Syn*
axiom $r \mapsto s \cdot (s\backslash r)$
axiom $r \mapsto (r/s) \cdot s$

ACG⊏GMR (2.4), and any implementation of ACG + LXN (2.2) in which the set of lexical categories **Lex** is finite gives an applicative categorial grammar. In ACG, $\mapsto$ can be viewed as metasyntax, in that $r \mapsto x$ is exactly equivalent to $\exists s\,.\,(x = s \cdot (s\backslash r) \vee x = (r/s) \cdot s)$. Thus $r \mapsto x$ implies x is a string of exactly two intensions. In defining a model like S (2.7), we have a choice of either having VT $\mapsto$ **saw**, where VT names a *pre-terminal*, encapsulating transitive verbs by virtue of being able to immediately dominate any one of them, and nothing else, or having VT = **saw**, where VT names an *abstract lexical entry*, and **saw**, **loves** and so on are just different names for the same object (**saw** = **loves** = VT). In a model of ACG, however, we cannot have VT $\mapsto$ **saw**, since **saw** is not a string of exactly two intensions. Thus in refinements of ACG, we must use abstract lexical entries, so that **saw** = **loves** = VT. Then the grammar

Basic Categories:

S, NP, N

Lexical Entries:

man:	N
telescope:	N
the:	NP/N
saw:	(NP\S)/NP
with:	(NP\NP)/NP
with:	((NP\S)\(NP\S))/NP

is equivalent to the specification C below

(Basic categories)

$\mathsf{BSC_C}$ = **extend** $\emptyset$
by **sort** Basic
opns S,NP,N :⟶ Basic

C = **derive from**
extend ACG + $\mathsf{CAT_F(BSC_C)}$

by **sort** Lex

axiom Lex $\subseteq$ Syn

opns saw, the, man, with, telescope $:\rightarrow$ Lex

axioms man = telescope = N

axioms the = NP/N , saw = (NP\S)/NP

axiom with_1 = (NP\NP)/NP

axiom with_2 = ((NP\S)\(NP\S))/NP

by inclusion of Sig[LXN+GMR]

references: LXN (2.2), GMR (2.4)

Recall (Section 1.8) that $\mathsf{CAT}_\mathsf{F}(\mathsf{BSC}_\mathsf{C})$ abbreviates

$$\mathsf{BSC}_\mathsf{C} + \mathsf{CAT}_\mathsf{F}(\textbf{derive from } \mathsf{BSC}_\mathsf{C} \textbf{ by inclusion of sort } \texttt{Basic})$$

derive from BSC_C **by inclusion of sort** Basic employs the signature morphism which "forgets" those parts of the signature of BSC_C not contained in the signature specified by **sort** Basic (i.e. the constants S, NP and N), and leaves everything else unchanged. This is needed here in order to "fit" BSC_C to the signature required by CAT_F. Every model of C is isomorphic to the model C which has

$[C\texttt{Syn}]$	=	all terms built using \ and / from basic terms S,NP,N
$[C\texttt{Lex}]$	=	{N, NP/N, NP\S, (NP\NP)/NP, ((NP\S)\(NP\S))/NP}
$[C_\_ \cdot \_]$		given by concatenation
$[C(r \mapsto x)]$		iff there is an s in $[C\texttt{Syn}]$ s.t. $x = s\ (s\backslash r)$ or $x = (r/s)\ s$
		e.g.: NP$\mapsto$(NP/N) N

My categorial digression complete, I shall return to considering the problem of ambiguity, and how this may be dealt with using $\mapsto$ (Figure 2–3). The two senses of our ambiguous phrase can be represented by the two distinct ways in which it can be licensed. So, because in the model S (2.7) above we have

VT · NP · PP $\mapsto^*$ saw · the · man · with · the · telescope and

VT · NP $\mapsto^*$ saw · the · man · with · the · telescope,

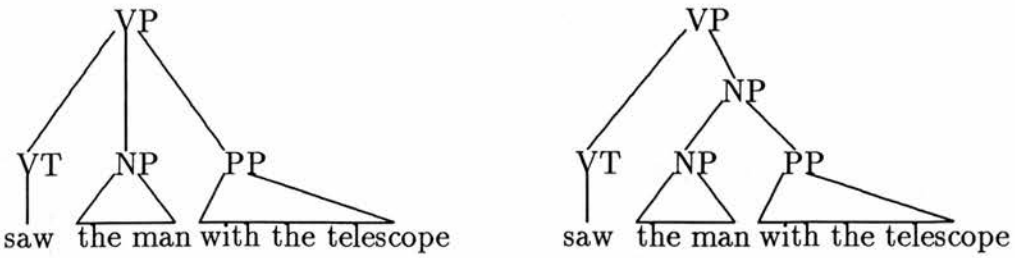

Figure 2–3: Dealing with ambiguity

then we say VT · NP · PP and VT · NP are both intensions for that string. If also

$$\text{VP} \mapsto^* \text{VT} \cdot \text{NP} \cdot \text{PP} \quad \text{and}$$
$$\text{VP} \mapsto^* \text{VT} \cdot \text{NP},$$

then we say also that VP *licenses* the string *via* either of these intension strings. If $r \mapsto^* w$ for some r in **Syn**, we say r licenses w. If $r \mapsto^* x$ *and* $x \mapsto^* w$, we say r licenses w via x. Now if

$$\text{VT} \cdot \text{NP} \mapsto^* \texttt{saw} \cdot \text{NP},$$

then NP is a constituent of VT · NP at the position indexed by **saw** and **e**, or indeed by VT and **e**, but assuming (as we would expect) that

$$\text{VT} \cdot \text{NP} \cdot \text{PP} \not\mapsto^* \texttt{saw} \cdot \text{NP},$$

then NP will not be a constituent of VT · NP · PP at those indices. Even though VT · NP and VT · NP · PP lie in **Syn**$*$ rather than **Syn**, and hence cannot license, their roles in the parse are sufficient to deal with the ambiguity. In this formulation, substitution of one constituent for another corresponds just to different realisations of the same constituent intension: instead of talking about substituting *a squirrel* for *the man* in *saw the man with the telescope*, we simply note

$$\text{VP} \mapsto^* \texttt{saw} \cdot \text{NP} \cdot \texttt{with} \cdot \texttt{the} \cdot \texttt{telescope} \quad \text{and}$$
$$\text{NP} \mapsto^* \texttt{a} \cdot \texttt{squirrel}.$$

These ideas should be equally applicable to any constituency grammar, even though they have been exemplified here in a context-free PSG.

At first it may seem unnecessary to force _·_ to be concatenation on intensions, just because it is concatenation on extensions (i.e. on **Lex***). For instance, we may like to have it be multiset union, to reflect a claim that linear order is independent of dominance structure (c.f. IDLP, Section 3.4). But if, as seems desirable in a broad specification, we are going to be able to index the position of subconstituents by words, intensions, or a mixture of the two, then the axioms will practically force _·_ to be concatenation on intensions as well as on **Lex***. (Proper) strings of intensions are best thought of as just indices for constituency, which is why they do not license.

Since the implementation S (2.6) of LXN + GMR (2.2, 2.4) gives a PSG, as above, then also S + PARSE (2.5) has (up to isomorphism) only one model, and can therefore be viewed as a program implementing an (abstract) parser for S. Let us briefly consider how, for given S, S + PARSE may be viewed as a program. There is a famous "equation" in logic programming,

ALGORITHM = LOGIC + CONTROL

(Kowalski 1979). The motivation behind abstract specification has been to give the logic of programs in a manner which is as far as possible independent of the control. Thus, when we write a "pure" Prolog program, we actually specify the *logic* of the program. Prolog itself has a built-in search strategy, which takes care of the *control.* However, problems of any complexity quickly get beyond what is easily dealt with in this "pure" language, for several reasons. One is that the Horn-clause logic used in Prolog was chosen because it admits straightforward procedural interpretation by any one of several search strategies, but unfortunately this does *not* make it well suited as a language for specifying the logic of a problem, *independently* of concerns about control. Another reason is that a single search strategy can never be appropriate for all applications. For instance, breadth-first search always finds any finite solution eventually, but may be horrendously inefficient, and may loop indefinitely if no solution exists. Any algorithm which

is guaranteed to find any finite solution eventually, even though it may never terminate, is called a *partial* (or *semi-*) *decision procedure* (Rogers 1967). At this level of abstraction we will be happy to show that even a partial decision procedure exists. The conditional form ($P_1 \wedge \ldots \wedge P_n \rightarrow P$) of the axioms of PARSE (2.5) makes it easy to rewrite directly into (roughly!) Prolog syntax.

```
...(insert clauses for S here)...
X↦* X.
X0-X↦* Y0-Y :- X0-X1↦* Y0-Y1, X1-X↦* Y1-Y.
[R|Y]-Y↦* X :- R↦X0, X0↦* X.
```

The use of an associative operation as the basis for encoding strings is not very efficient (we see issues of control re-intruding already!). A common technique in Prolog is to represent strings by a pair of lists `X-Y`, also called a *difference list*, where `X` and `Y` are variables standing for lists. Prolog lists are written `[R|X]`, where `R` is the first element of the list, and `X` is the *tail* of the list, that is, the list obtained when the first element is deleted. (The empty list is written `[]`). A breadth-first interpretation of this will yield a partial decision procedure for Prolog-style queries on PARSE (2.5).

2.3 Coda

In a formal treatment, the use of strings of words to model statements in human language is naturally suggested by our familiarity with the written word. The adequacy and relevance of this approach to modelling the spoken word is open to debate, but does not concern us here. The majority of work in the syntax of natural language is founded on some notion of constituency, which we take as meaning that some substrings of a statement appear to exhibit a privileged status, by virtue of various properties, including substitution properties. A natural assumption might be that subconstituency can be represented as a relation on strings, akin to the substring relation. However this fails because constituency appears to be context

dependent. So a substring may appear to act like a constituent in isolation, but fail to do so when put in a given context. It seems inevitable that some kind of intensional domain will be required. A parse-in-one step relation on the intensions can then be used to represent immediate constituency. We proceed to consider (in the abstract) how such systems can deal with the various cross-cutting dimensions of natural language syntax (case, agreement, and so on).

Chapter 3

Refinement and Implementation

In this chapter I will consider refinements to the basic predicate $\mapsto$ of GMR (2.4), designed to give us some vocabulary for the specification of more complex grammatical phenomena, such as agreement. I will consider the "core" part of several standard grammatical frameworks, attempt to describe some of the ideas they employ in an abstract enough fashion that their approaches can more easily be seen as the conjunction of several such abstract specifications, some in common, and some not. I attempt to sketch how we can make further refinements of these abstract approaches to produce implementations, that is, single models, equivalent to the standard models for grammars in the various frameworks.

The frameworks I shall mainly consider are LFG, GPSG and HPSG. Sells (1985) contains brief introductions to both LFG and GPSG, as well as Government and Binding theory (GB), to which limited reference will also be made. Bresnan (1982) is the standard reference for LFG, as is Gazdar et al. (1985) (henceforth: GKPS) for GPSG. For LFG, some reference is also made to Kaplan and Zaenen (1987). For HPSG, I refer to the presentation in Pollard and Sag (1987) (henceforth: P&S). These three frameworks have in common that they have all been defined in some kind of *attribute-value* language, also called *feature-value*, *feature structure* or *unification-based* formalisms. One of the simplest such languages is PATR-II. Shieber (1986) gives some of the history of these systems, describes the PATR-II language, and discusses what enhancements would be needed in order to define LFG, GPSG or HPSG.

3.1 PATR-II

Perhaps the most obvious move away from context-free PSGs is to put more structure into the intensions, with the consequent ability to replace a great multiplicity of rules by a few more general ones. Categorial grammars offer an extreme example of this. All the work done by the rules in a PSG has been moved into the intensions themselves, by means of the operations $\backslash$ and $/$. Then a rule $r \mapsto s \cdot t$ becomes equivalent to $t = (s\backslash r) \vee s = (r/t)$. Only two rules are needed.

Another direction would be to add structure for dealing with *agreement.* Some grammars may not make any external distinction between NPs, so that a context well-formed for one NP is well-formed for any other. Such a grammar must ignore issues such as agreement, allowing, for instance, *John smile* as a sentence. Many grammars, however, do take sufficient account of agreement phenomena to disallow *John smile.* In such a grammar, there can no longer be a single intension, "NP". For instance, if we are to allow *the women smile*, we must have different intensions for *John* and *the women.* The atomic category labels NP, S and so on give one way of classifying intensions, agreement properties give another. One common way of doing this is to make `cat` a partial function which maps full intensions to atomic category labels, and `agr` a partial function into some domain modelling agreement properties. This might lead to the specification of rules like

(3.1)
$$\begin{aligned}
&\mathtt{cat}(s) = \text{S} \wedge \\
&\mathtt{cat}(n) = \text{NP} \wedge \\
&\mathtt{cat}(v) = \text{VP} \wedge \\
&\mathtt{agr}(n) = \mathtt{agr}(v) \\
&\rightarrow s \mapsto n \cdot v
\end{aligned}$$

$$\begin{aligned}
&\mathtt{cat}(r) = \text{NP} \wedge \\
&\mathtt{num}(\mathtt{agr}(r)) = \text{SINGULAR} \wedge \\
&\mathtt{pers}(\mathtt{agr}(r)) = 3 \\
&\rightarrow r \mapsto \mathtt{john}
\end{aligned}$$

$$\begin{aligned}&\texttt{cat}(r) = \text{VP} \wedge\\ &\texttt{num}(\texttt{agr}(r)) = \text{SINGULAR} \wedge\\ &\texttt{pers}(\texttt{agr}(r)) = 3\\ &\rightarrow r \mapsto \texttt{smiles}\end{aligned}$$

This is (modulo trivial notational variance) exactly the form of rules in the grammar-writing language PATR-II, and (3.1) could be run as a PATR-II program. The sentences are all implicitly universally quantified over their respective variables, so for instance the second says that any r with category NP and the specified agreement properties can act as an intension for `john`. Given a PATR-II interpreter, we should be able to issue the query

(3.2) $\quad ?\ \texttt{cat}(s) = \text{S} \wedge s \mapsto^* n \cdot \texttt{smiles}$

and expect an answer like

$$s = \begin{bmatrix} \texttt{cat: S} \end{bmatrix}$$

$$n = \begin{bmatrix} \texttt{cat: NP} \\ \texttt{agr:} \begin{bmatrix} \texttt{num: SINGULAR} \\ \texttt{pers: 3} \end{bmatrix} \end{bmatrix}$$

(and possibly other answers besides). PATR-II only knows about two kinds of properties: those expressed by unary partial functions (*features* in feature-value languages) like `cat`, `agr` and so on, and those expressed by $\mapsto$. PATR-II keeps track of features, and is able to deduce what features are required of an object s in order that a goal like (3.2) be satisfiable. The matrix notation merely summarises the features we need to look for in the objects to which s and n are to correspond, in order that the sentence in the query is valid. So for instance, any object n

must be mapped by **cat** to the object NP, and by **agr** to an object which is itself mapped by **num** to the object SINGULAR, and by **pers** to the object 3. It could just be considered alternative notation for

$$\begin{aligned}
&\mathtt{cat}(s) = \mathrm{S} \wedge \\
&\mathtt{cat}(n) = \mathrm{NP} \wedge \\
&\mathtt{num}(\mathtt{agr}(n)) = \mathrm{SINGULAR} \wedge \\
&\mathtt{pers}(\mathtt{agr}(n)) = 3
\end{aligned}$$

The answer means that the program (grammar) entails that the formula in the query (3.2) is provably satisfied by any objects s and n which satisfy the conditions given in the answer, which is to say, any model of the PATR-II grammar satisfies the formula with respect to any variable assignment which satisfies the properties listed in the answer. This is analogous to the way Prolog gives as answers properties of variable assignments which make the query valid. Before we come to consider just what *are* the intended model(s) of a PATR-II program, it will be useful to recall the simpler case of Prolog.

3.1.1 Prolog

A Prolog implementation of successor arithmetic like

```
plus(0,X,X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
```

might answer a query

```
?  plus( s(s(0)), X, Y ).
```

by printing something like

```
X = _54, Y = s(s(_54)).
```

which is like `Y = s(s(X))`. This means that the query sentence is satisfied in any model, with respect to any choice of `X` and `Y` which satisfies this property. Unfortunately, Prolog is not capable of considering properties like `cat(X) = np`, because it does not deal with true equality: when we write `X = Y` in a Prolog program, this is interpreted syntactically, by term identity. Thus, PATR-II grammars cannot *directly* be interpreted as Prolog programs.

In algebraic semantics, we identify a program with a model. The only difference is that in a program, we may need to concern ourselves with *how* the result of a function application, say, is calculated, but in a model we are only interested that there be an answer. What model is chosen for the semantics of a Prolog program? A Prolog program can be read as a specification in a very obvious way, for instance,

$$
\begin{aligned}
\mathsf{PLUS} \quad = \quad & \textbf{sort } \mathtt{All} \\
& \textbf{pred } \mathtt{plus} \subseteq \mathtt{All} \times \mathtt{All} \times \mathtt{All} \\
& \textbf{opn } \mathtt{0} :\rightarrow \mathtt{All} \\
& \textbf{opn } \mathtt{s} : \mathtt{All} \rightarrow \mathtt{All} \\
& \textbf{axiom } \mathtt{plus}(\mathtt{0}, x, x) \\
& \textbf{axiom } \mathtt{plus}(x, y, z) \rightarrow \mathtt{plus}(\mathtt{s}(x), y, \mathtt{s}(z))
\end{aligned}
$$

But, as a specification, this will have many models. What model is to give the semantics of the corresponding program (PPP say)? Given the preceding description of the semantics of a Prolog query, it will need to be a model such that any query is satisfied just in case *every* model of the specification satisfies the query. Such a model is called *initial*, and is unique up to isomorphism. (It should be remembered, though, that queries may be drawn only from a limited repertoire — in Prolog, conjunctions of atomic formulae. See Lloyd (1984) for technical details.) This is the model defined by

(Prolog plus program).

$$
\begin{aligned}
\mathsf{PPP} \quad = \quad & \textbf{extend } \emptyset \textbf{ by} \\
& \qquad \textbf{sort } \mathtt{All} \\
& \qquad \textbf{pred } \mathtt{plus} \subseteq \mathtt{All} \times \mathtt{All} \times \mathtt{All}
\end{aligned}
$$

opn 0 :→ All

opn s : All → All

axiom plus(0, x, x)

axiom plus(x, y, z) → plus(s(x), y, s(z))

The use of **extend** means we get only reachable models, and terms are equated only when they *must* be (in this case, not at all). These, the initial models, are mutually isomorphic. For Prolog programs, it easy to see how such specifications may be read as a recipe to construct the initial model. Sorts are interpreted by the sets of ground terms of those sorts, constants are interpreted as themselves (0 as 0), and operations by term formation (so s is interpreted by the function that maps any ground term t to the ground term s(t)). Prolog "axioms" give us the base cases for what to put into the interpretation of the predicates (so for plus we start off with triples $\langle 0, t, t \rangle$ for all ground terms t), and the conditional program clauses may be read as recursive recipes for completing predicate interpretation (thus for plus, recursively add for each triple $\langle x, y, z \rangle$ already in the interpretation, the new triple $\langle s(x), y, s(z) \rangle$). So now, instead of trying to reason about whether a sentence is satisfied in *all* models of PLUS, we can just reason about whether it is satisfied in the initial model given by PPP.

So when we ask Prolog

```
? plus( s(s(0)), 0, Y ).
```

and receive the reply

```
Y = s(s(0)).
```

we can view this as telling us either that the term s(s(0)) is an appropriate object for Y in the initial model, or that in any model, the value of that term will make the query sentence true. Now when we ask

```
? plus( s(s(0)), X, Y ).
```

and get the response

```
X = _54, Y = s(s(_54)).
```

the underscore (_54) tells us that, in the initial model, any ground term t can be used as the object X, provided s(s(t)) is used for Y. Equivalently, for an arbitrary model, we could select any object for X, provided we use for Y the object reached by two applications of the function interpreting the operation name s.

Prolog answers are always either ground, or contain underscore variables. Thus they not only give us relations which must hold between the objects corresponding to the query variables (for instance Y = s(s(X))), but also give us a recipe for finding such objects (by choosing arbitrary objects for underscore variables).

3.1.2 Back to PATR-II

A PATR-II query

$$? \ \mathtt{cat}(s) = \mathrm{S} \wedge s \mapsto^* \mathtt{john} \cdot \mathtt{smiles}$$

might produce an answer like

$$s = \begin{bmatrix} \mathtt{cat}\colon \mathrm{S} \end{bmatrix}$$

or, alternatively,

$$\mathtt{cat}(s) = \mathrm{S}.$$

This is to mean that, in an arbitrary model, s may correspond to any object such that cat(s) = S. This, however, does not tell us how to actually find such an s, and indeed there may not be one. In particular, there will be no such s in the initial model specified by

extend $\emptyset$ **by**

sorts Lex,Syn

pred $\_ \mapsto \_ \subseteq$ Syn $\times$ Syn$*$

opns john, smiles $:\rightarrow$ Lex

opns NP,VP,S,SINGULAR,PLURAL,0,1,2,3 $:\rightarrow$ Syn

opns cat, agr, num, pers : Syn$\dot{\rightarrow}$Syn

axiom Lex $\subseteq$ Syn

axiom $\mathtt{cat}(n) = \text{NP} \wedge$
$\mathtt{cat}(v) = \text{VP} \wedge$
$\mathtt{agr}(n) = \mathtt{agr}(v)$
$\rightarrow s \mapsto n \cdot v$

axiom $\mathtt{cat}(r) = \text{NP} \wedge$
$\mathtt{num}(\mathtt{agr}(r)) = \text{SINGULAR} \wedge$
$\mathtt{pers}(\mathtt{agr}(r)) = 3$
$\rightarrow r \mapsto \mathtt{john}$

axiom $\mathtt{cat}(r) = \text{VP} \wedge$
$\mathtt{num}(\mathtt{agr}(r)) = \text{SINGULAR} \wedge$
$\mathtt{pers}(\mathtt{agr}(r)) = 3$
$\rightarrow r \mapsto \mathtt{smiles}$

(since the operations — **cat**,**agr**,**num**,**pers** — are partial, and only defined when the axioms insist they be defined). So, while the selection of an s with this property would indeed show $\exists s \,.\, \mathtt{cat}(s) = \text{S} \wedge s \mapsto^* \mathtt{john} \cdot \mathtt{smiles}$ to be true, if we are dealing with the initial model, there is no such s, and we should really return some value like **false** or **no** to indicate failure. This would of course ruin the usefulness of PATR-II. By simply adding signature information and restricting to the initial interpretation, a Prolog program is made to specify its intended model, but the correspondence (between program syntax and intended model) is not quite so obvious for PATR-II.

Nevertheless there are parts of the interpretation given to a PATR-II grammar which do have an initial flavour: the fact that the constants (*atoms* in the terminology of feature-value work: `john`, NP, SINGULAR and so on) are kept distinct, and that $r \mapsto x$ and $x \mapsto^* y$ should only hold when forced to by the conditions (rules) of the grammar. The rules can be bolted on at the end, using the free extension operation.

(3.3) (A Lexicon)

PWDS = **extend** $\emptyset$
by **sorts** `Lex`
opns `john, smiles` $:\rightarrow$ `Lex`

(3.4) (Some atoms)

PAT = **extend sort** `Lex`
by sort `Atom`
opns NP,VP,S,SINGULAR,PLURAL,0,1,2,3 $:\rightarrow$ `Atom`

(3.5) (Atoms are intensions)

ATOM = **sorts** `Atom,Syn`
axiom `Atom` $\subseteq$ `Syn`

(Some features)

FSYN = **sorts** `Syn`
opns `cat, agr, num, pers` : `Syn` $\dot{\rightarrow}$ `Syn`

(Lexical entries program)

PLEX = **extend**
Sig[PWDS + PAT + FSYN] + ATOM
by **pred** `lex` $\subseteq$ `Syn` $\times$ `Lex`
axiom $\mathtt{cat}(r) = \text{NP} \wedge$
$\mathtt{num}(\mathtt{agr}(r)) = \text{SINGULAR} \wedge$
$\mathtt{pers}(\mathtt{agr}(r)) = 3$
$\rightarrow \mathtt{lex}(r, \mathtt{john})$

axiom $\mathtt{cat}(r) = \text{VP} \wedge$
$\mathtt{num}(\mathtt{agr}(r)) = \text{SINGULAR} \wedge$
$\mathtt{pers}(\mathtt{agr}(r)) = 3$
$\rightarrow \mathtt{lex}(r, \mathtt{smiles})$

(The parsing part of the grammar)

PP = **extend**
Sig[PLEX] + ATOM + LXN + SYN*
by **pred** $\_ \mapsto \_ \subseteq \mathtt{Syn} \times \mathtt{Syn}*$
axiom $\mathtt{cat}(s) = \text{S} \wedge$
$\mathtt{cat}(n) = \text{NP} \wedge$
$\mathtt{cat}(v) = \text{VP} \wedge$
$\mathtt{agr}(n) = \mathtt{agr}(v)$
$\rightarrow s \mapsto n \cdot v$
axiom $\mathtt{lex}(r, w) \rightarrow r \mapsto w$

references: LXN (2.2), SYN* (2.3)

In PWDS and PAT we use **extend** to ensure that the words and atoms are all kept distinct. ATOM will be used to include the sort `Atom`, used in constructing the atoms, in the sort `Syn`. FSYN gives us the vocabulary to refer to the partial functions `cat`, `agr`, `num` and `pers`. PLEX uses **extend** to create lexical entries $\mathtt{lex}(r, w)$ relating words w and intensions r. This predicate contributes to the definition of $\mapsto$ in PP. By using **extend** to define these two predicates we ensure that they are satisfied only when they *must* be, in virtue of the axioms (rules) in the extensions.

PWDS+PAT+PLEX+PP is quite a concrete construction, but it does still admit non-isomorphic models. What it fails to specify is, what the elements of `Syn` are, and how `cat`, `agr` and so on behave on them. But this is as far as free extension can take us on this vocabulary. If FSYN used **extend**, the functions `cat`, `agr` and so on would be everywhere undefined, and the only elements of `Syn` would be those of `Lex` and `Atom`. This is clearly not what we require.

What can we say of the intended model M (say) of FSYN? A PATR-II interpreter works by attempting to gradually build up a verifying model. Whenever it is faced with a set of equational conditions, it attempts to extend the model to satisfy these axioms. In a free extension, properties hold just in case they are satisfied in *every* suitable extension. In M, however, a set of equations should be satisfiable by some choice of objects for variables, if and only if there is *some* suitable extension in which they are satisfied by some choice of objects for variables.

3.1.3 Constructor-extractor implementation

In the implementation of our PATR-II grammar, an alternative way of adding, say, number information to a category would have been to use terms like NP(SINGULAR) (a strategy employed in so-called *definite clause grammars*). There is no particular reason to give NP different status to SINGULAR: instead we could employ a single constructor f, say, and terms like f(NP, SINGULAR). But as we add more and more features, and allow the values they take on to themselves have complex structure, we might get terms like

$$\mathtt{f}(\mathtt{f}(\mathrm{S}, \mathrm{NIL}, \mathrm{BACK}, \mathtt{f}(\mathrm{NP}, x, \mathrm{NIL}, \mathrm{NIL})), x, \mathrm{FORW}, \mathtt{f}(\mathrm{NP}, y, \mathrm{NIL}, \mathrm{NIL}))$$

and much worse, and it becomes extremely difficult to keep track of which argument is supposed to correspond to which feature, at which level of embedding of a term. So instead we employ *extractors* (Section 1.7.5) to do the work of remembering argument position.

(3.6) (The intended model M of FSYN)

PSYN = **extend sorts** Lex,Atom **by**

sorts Avs,Syn

opn f : $\mathtt{Syn}^4 \rightarrow \mathtt{Avs}$

opns cat, agr, num, pers : Avs $\rightarrow$ Syn

axiom Lex, Atom, Avs $\subseteq$ Syn

axiom $\mathtt{cat}(\mathtt{f}(x_1, x_2, x_3, x_4)) = x_1$

axiom $\mathtt{agr}(\mathtt{f}(x_1, x_2, x_3, x_4)) = x_2$

$$\textbf{axiom } \mathtt{num}(\mathtt{f}(x_1, x_2, x_3, x_4)) = x_3$$
$$\textbf{axiom } \mathtt{pers}(\mathtt{f}(x_1, x_2, x_3, x_4)) = x_4$$

Recall $\mathtt{Syn}^4$ abbreviates $\mathtt{Syn} \times \mathtt{Syn} \times \mathtt{Syn} \times \mathtt{Syn}$. So **f** recursively constructs complex attribute-value structures (**Avs**) out of four-tuples of intensions, one corresponding to each feature **cat**, **agr**, **num**, **pers**. Thus if one feature is defined, all are. This defines a so-called *fixed arity* feature-value system. In an *open arity* system, their definedness is independent. We shall see an example of such a system in Section 3.3. In PATR-II this would be redundant, as the language does not talk about undefined features. Up to isomorphism, PWDS+PAT+PSYN+PLEX+PP has only one model. In a model of PWDS+PAT+PSYN+PLEX+PP+PARSE (2.5) we should be able to ask $\exists sn.s \mapsto^* n \cdot \mathtt{smiles}$ and be given an answer like

$$s = \mathtt{f}(\mathrm{S}, \_, \_, \_),$$
$$n = \mathtt{f}(\mathrm{NP}, \mathtt{f}(\_, \_, \mathtt{singular}, 3), \_, \_)$$

(where the underscores indicate that any value will do) or, with a little syntactic sugar,

$$s = \begin{bmatrix} \\ \mathtt{cat}\colon \mathrm{S} \\ \\ \end{bmatrix}$$

$$n = \begin{bmatrix} \\ \mathtt{cat}\colon \mathrm{NP} \\ \\ \mathtt{agr}\colon \begin{bmatrix} \mathtt{num}\colon \mathrm{SINGULAR} \\ \\ \mathtt{pers}\colon 3 \\ \\ \end{bmatrix} \\ \\ \end{bmatrix}$$

PATR-II is just one, very simple, executable example of a feature-value language. These can all be viewed as being based on specifications which use just the extractors, where there is a canonical choice of (often anonymous) constructor(s). Although I have employed several sorts in modelling PATR-II, it is not truly a multi-sorted language, as the user is not able to make further sortal distinctions.

In a single-sorted language like PATR-II, there is just one constructor. In a multi-sorted setting, we typically have one constructor per sort. Pollard and Sag (1987), for instance, use multiple sorts, and associate each sort with a characteristic set of features, each taking values of a named sort. Again, the features can usefully be viewed as extractors, where each sort is associated with a single constructor (and/or some constants), of arity determined by the sorts associated with the characteristic features. In the single-sorted approach, the constructor typically becomes cluttered with features which are irrelevant as often as not. In a multi-sorted approach, the constructors are designed in such a way that only relevant features participate in a construction. For example, to recast the preceding grammar to make better use of sorts, we might have

(3.7) (Sorted Syn program)

SSP = **extend sort** Lex

by **sorts** Syn,Cat,Agr,Num,Pers

opns NP,VP,S :⟶ Cat

opns 1,2,3 :⟶ Pers

opns SINGULAR,PLURAL :⟶ Num

axiom Lex ⊆ Syn

opn $\langle \_, \_ \rangle$: Cat × Agr → Syn

opn cat : Syn → Cat

opn agr : Syn → Agr

axiom $\mathtt{cat}(\langle x, y \rangle) = x$

axiom $\mathtt{agr}(\langle x, y \rangle) = y$

opn $\langle \_, \_ \rangle$: Num × Pers → Agr

opn num : Agr → Num

opn pers : Agr → Pers

axiom $\mathtt{num}(\langle x, y \rangle) = x$

axiom $\mathtt{pers}(\langle x, y \rangle) = y$

(Lexical entries program)

SPLEX = **extend** Sig[PWDS] + Sig[SSP]
by **pred** $\mathtt{lex} \subseteq \mathtt{Syn} \times \mathtt{Lex}$
axiom $\mathtt{cat}(r) = \text{NP} \wedge$
$\mathtt{num}(\mathtt{agr}(r)) = \text{SINGULAR} \wedge$
$\mathtt{pers}(\mathtt{agr}(r)) = 3$
$\rightarrow \mathtt{lex}(r, \mathtt{john})$
axiom $\mathtt{cat}(r) = \text{VP} \wedge$
$\mathtt{num}(\mathtt{agr}(r)) = \text{SINGULAR} \wedge$
$\mathtt{pers}(\mathtt{agr}(r)) = 3$
$\rightarrow \mathtt{lex}(r, \mathtt{smiles})$

(3.8) (The parsing part of the grammar)

SPP = **extend** Sig[SPLEX] + SYN*
by **pred** $\_ \mapsto \_ \subseteq \mathtt{Syn} \times \mathtt{Syn}*$
axiom $\mathtt{cat}(s) = \text{S} \wedge$
$\mathtt{cat}(n) = \text{NP} \wedge$
$\mathtt{cat}(v) = \text{VP} \wedge$
$\mathtt{agr}(n) = \mathtt{agr}(v)$
$\rightarrow s \mapsto n \cdot v$
axiom $\mathtt{lex}(r, w) \rightarrow r \mapsto w$

references: SYN* (2.3)

Models of PWDS + SSP + SPLEX + SPP are mutually isomorphic. In a model of PWDS + SSP + SPLEX + SPP + PARSE(2.5) the query (3.2) might elicit the response

$s = \langle \text{S}, \_ \rangle,$
$n = \langle \text{NP}, \langle \mathtt{singular}, 3 \rangle \rangle$

or equivalently

$$s = \begin{bmatrix} \texttt{cat: S} \end{bmatrix}$$

$$n = \begin{bmatrix} \texttt{cat: NP} \\ \texttt{agr:} \begin{bmatrix} \texttt{num: SINGULAR} \\ \texttt{pers: 3} \end{bmatrix} \end{bmatrix}$$

The use of **extend**, rather than **reachable enrich**, in the specifications PSYN (3.6) and SSP (3.7) merely serves to keep the constants distinct. Thus we could have written them instead using **reachable**, by including axioms to keep the constants distinct, so $\text{NP} \neq \text{VP}, \text{NP} \neq \texttt{john}, \ldots$. Clearly this will be rather awkward as the number of constants grows large. However, as mentioned before, this route does have the advantage that we could include axioms of the form, say

$$\textit{conjunction of equations} \ldots \rightarrow (\texttt{cat}(x) = \text{VP} \vee \texttt{cat}(x) = \text{S})$$

Of course this means we would have non-isomorphic satisfying models, but it may be that we can nevertheless answer many queries about the class of models (determine whether the corresponding sentences are true in all models of the class), perhaps by reference to a model of a more complex structure designed with those queries in mind. The use of free extensions is one very general way of finding an implementation for a specification, and there are others which will work for different classes of underlying specifications, but all are liable, sooner or later, to fall short. For that reason, it seems of primary importance to draw a firm distinction between specifying a class of models with which to reason, and the isolation of a single model or routine system of calculation in which features of the models may be tested against experience. Considered merely as refinements of the class of validating models, imposition of the axioms poses no problem. Of course implementation is important, but perhaps we had better be certain of

our ground before spending too much time on abstruse model constructions of uncertain generality.

Thus it would seem wise to progress by delineating, to the best of our knowledge at a given time, the set of validating models we want to consider. Given such a specification, it is very likely that implementations capable of deciding desired forms of query will suggest themselves. Such implementations may, when compared against actual linguistic behaviour, in turn suggest deficiencies in the current delineation of suitable models; and so the process continues. This is how science proceeds: we observe, we model, we compare.

Work such as Pollard and Sag (1987) is probably best seen primarily as presenting such a specification of validating models. It also spends much time in suggesting how to build canonical models for (that is, to implement) the fragment it presents (and many papers have been written to do more of this), but in the absence of formal modularisation techniques, it is difficult to be certain how those models will or should interact with what is *not* given. (For instance, one topic not treated in detail in P&S, but much investigated since, is the issue of how properly to deal with set-valued features). This is not meant to suggest that there is anything wrong with what is presented there; but rather to suggest that by recognising that we will *always* be dealing with fragments, and by adopting from the beginning modular techniques for formally combining such fragments, we may avoid much duplication of effort (for instance by producing a single implementation of, say, relative clauses, which can be used, without modification, to extend a whole range of different kinds of "core" grammars). We work in tandem on *both* a general specification of a class of models, *and* model constructions (possibly abstruse, but of certain generality).

3.2 Abstract Features

We have already seen that there are at least two ways of representing the cross-cutting dimensions along which a feature like **cat** or **agr** divides the intensions: we can use a single-sorted approach, or a multi-sorted one. Another variation would be to use more subsorts in place of atomic constants. But the purpose of all of these is to classify the objects of **Syn** across several dimensions, in order to provide a richer structure to represent the niceties of grammatical judgement. Such a classification simply defines an *equivalence relation* on **Syn**, since the effect of such a relation is to partition the carrier into distinct *equivalence classes.* Since other ways of specifying such effects imply an equivalence relation, but not necessarily vice-versa, at the more abstract level, the use of equivalence relations is appropriate. The use features (extractors) is simply one technique for implementing equivalences.

(3.9) (Partial Equivalence Relations)

$$\begin{aligned} \mathsf{PEQV} \quad = \quad & \textbf{sort } \mathtt{Syn} \\ & \textbf{pred } \_ \equiv \_ \subseteq \mathtt{Syn} \times \mathtt{Syn} \\ & \textbf{axiom } r \equiv s \rightarrow s \equiv r \\ & \textbf{axiom } r \equiv s \wedge s \equiv t \rightarrow r \equiv t \end{aligned}$$

(3.10) (Total Equivalence Relations)

$$\begin{aligned} \mathsf{EQV} \quad = \quad & \textbf{enrich } \mathsf{PEQV} \\ & \textbf{by} \quad \textbf{axiom } r \equiv r \end{aligned}$$

(3.11) (Atomic Categories: S, NP...)

$$\begin{aligned} \mathsf{CAT} \quad = \quad & \textbf{derive from } \mathsf{PEQV} \\ & \textbf{by} \quad [\_ =_{\mathtt{cat}} \_ \longmapsto \_ \equiv \_] \end{aligned}$$

(3.12) (Agreement)

$$\begin{aligned} \mathsf{AGR} \quad = \quad & \textbf{derive from } \mathsf{PEQV} \\ & \textbf{by} \quad [\_ =_{\mathtt{agr}} \_ \longmapsto \_ \equiv \_] \end{aligned}$$

An equivalence relation is reflexive, symmetric, and transitive. A partial equivalence relation need not be reflexive, but it must be symmetric and transitive, and thus if $r \equiv r'$, $r' \equiv r$ by symmetricity, and, combining these two facts using transitivity, $r \equiv r$ (and $r' \equiv r'$). A partial equivalence relation, like a total one, partitions its domain of definition into equivalence classes. In fact the only difference is that, in the partial case, there may be a class of objects which play no part in the relation at all. That is, there may be zero, one or more r in **Syn** such that for no r' is $r \equiv r'$. Every model of PEQV is such that $\forall r \,.\, (\exists r' \,.\, r \equiv r') \leftrightarrow r \equiv r$, and in a total equivalence relation, $\forall r \,.\, r \equiv r$. Instead of $\mathtt{agr}(n) = \mathtt{agr}(v)$, we have $n =_{\mathtt{agr}} v$, with no need of special sorts or constants.

One way to get the effect of $\mathtt{cat}(n) = \mathtt{NP}$ is to use a representative of the class, so $n =_{\mathtt{cat}} \mathtt{john}$. Or we can also name the equivalence classes, so $\mathtt{np}(n) \leftrightarrow n =_{\mathtt{cat}} \mathtt{john}$, or $n : \mathtt{NP} \leftrightarrow n =_{\mathtt{cat}} \mathtt{john}$. Partial functions (like **cat**) are just one way of implementing partial equivalence relations (like $=_{\mathtt{cat}}$). If this is the technique chosen, then the equivalence relation can be defined (by $r =_{\mathtt{cat}} (s) \leftrightarrow \mathtt{cat}(r) = \mathtt{cat}(s)$), and, this done, the old (functional) syntax can de derived away, thus giving an implementation (of $=_{\mathtt{cat}}$). But there are other implementation techniques which could be chosen, such as axiomatising the relation directly.

So if S is any sentential intension (say $\mathtt{S} \mapsto^* \mathtt{john} \cdot \mathtt{smiles}$), NP a noun phrase intension (so perhaps $\mathtt{NP} \mapsto^* \mathtt{the} \cdot \mathtt{women}$) and VP a verb phrase intension (say for *eats fish*), we might replace S⟶NP VP by

$$\forall snv : \mathtt{Syn}.s =_{\mathtt{cat}} \mathtt{S} \wedge n =_{\mathtt{cat}} \mathtt{NP} \wedge v =_{\mathtt{cat}} \mathtt{VP} \wedge n =_{\mathtt{agr}} v \rightarrow s \mapsto n \cdot v$$

Note that the **cat** and **agr** equivalences need not be total. For instance, in a model of CAT+AGR+ACG (2.8), we might only have $=_{\mathtt{agr}}$ defined for nominal intensions (such as nouns and noun phrases). So if **he** is an intension for *he*, then for any n such that $n =_{\mathtt{cat}} \mathtt{he}$ and $n =_{\mathtt{agr}} \mathtt{he}$, then $n\backslash\mathtt{S}$ (employing the categorial slash) could serve as a possible intension for *smiles*, *sleeps* and so on, and likewise if $n =_{\mathtt{cat}} \mathtt{he}$ but $n \neq_{\mathtt{agr}} \mathtt{he}$, $n\backslash\mathtt{S}$ might be an intension for *smile*, *sleep* and so on.

Another way of putting more structure on the intensions might be to separate the "N" and "P" of "NP". Suppose we have a rule like S⟶PP VP, for sentences

like *From the cave came a terrible groan.* We might like to replace this rule and S⟶NP VP by a more general rule S⟶XP VP, where the "X" of XP may take on different values: N, P and so on. (Presumably we will also need some way to forbid S⟶VP VP, for instance a side condition like X∈{N,P} on the rule). We might say the category XP is *underspecified*, in that we know it is a full phrase, but we don't know what sort. This is the basis of *X-bar theory* (Jackendoff 1977). Although there are all sorts of possible variations on the theory, the basic notion is that every intension has a *major category* (N, V and so on), and a *bar level.* The number of bar levels may vary, but the lexical level (noun, verb...) is bar-0, the next level is bar-1, and so on up to the full phrase level (NP, VP...), which will have the highest bar number. But as with category values, the important thing is the division into the different classes. How we name the classes is not so important.

(3.13) (Major Category)

$$\mathsf{MAJ} \quad = \quad \textbf{derive from PEQV}$$
$$\textbf{by} \quad [\_ =_{\mathtt{maj}} \_ \longmapsto \_ \equiv \_]$$

(3.14) (Bar Level)

$$\mathsf{BAR} \quad = \quad \textbf{derive from PEQV}$$
$$\textbf{by} \quad [\_ =_{\mathtt{bar}} \_ \longmapsto \_ \equiv \_]$$

(3.15)

$$\mathsf{XBAR} \quad = \quad \textbf{enrich CAT} + \mathsf{MAJ} + \mathsf{BAR}$$
$$\textbf{by} \quad \textbf{axiom } r =_{\mathtt{maj}} s \wedge r =_{\mathtt{bar}} s \rightarrow r =_{\mathtt{cat}} s$$

Clearly there is more to agreement than is seen in AGR (3.12). In fact AGR does no more than give a simple vocabulary to talk about agreement. Likewise XBAR (3.15) does not aim to say very much about X-bar theory. Rather it is intended as a highly abstract characterisation suitable for use in tying together modular descriptions involving X-bar theory, as GMR (2.4) was intended to do for modules involving constituency. XBAR, for instance, does nothing to ensure that $=_{\mathtt{maj}}$ and $=_{\mathtt{bar}}$ are orthogonal: they could even coincide. Since other constructions are unlikely to rely crucially on this orthogonality, we have no need to insist on it at

this point. There may even turn out to be languages in which it is natural to allow them to coincide. Perhaps as we investigate specific constructions which use an X-bar theory, and as we come to see what properties of the theory such constructions actually rely on, it will become clearer what the abstract requirements for such a theory really are.

3.3 LFG

Grammars in LFG are given by annotated context-free rewrite rules. For instance, S $\longrightarrow$ NP VP, might be annotated

(3.16)
$$\begin{array}{lcll} \text{S} & \longrightarrow & \text{NP} & \text{VP} \\ & & \uparrow \texttt{subj} = \downarrow & \uparrow = \downarrow \end{array}$$

The $\uparrow=\downarrow$ annotation on the VP indicates that the mother ($\uparrow$) must share all the properties of the daughter ($\downarrow$) so annotated, *except* the category label (S or VP). The NP is annotated $\uparrow\texttt{subj} = \downarrow$. Here `subj` is a partial function which indicates the subject of a phrase. Such an annotation says that the NP is the mother's subject.

In LFG, the *c-structure* is the part of an intension dealing with category labels and context-free rules, that is, the major category plus the bar level. The rest is called *f-structure*, and it is the sharing of f-structure which is indicated by $\uparrow=\downarrow$. The f-structure shared by the S and VP in an instance of the preceding rule might be described in matrix feature-value notation by s, as follows.

$$s = \begin{bmatrix} \texttt{subj}\colon n \\ \texttt{pred}\colon \texttt{smiles}(n) \end{bmatrix}$$

$$\text{where } n = \begin{bmatrix} \texttt{pred}\colon \texttt{john} \end{bmatrix}$$

n is the f-structure corresponding to the NP. Syntactically, the only purpose of the `pred` values is to encode which grammatical functions (`subj`,`obj`...) must be defined, and which may (optionally) be. In what follows I will therefore only deal with that function, and not explicitly mention `pred` values.

In GPSG and HPSG, the part of the intension shared in the way the f-structure is in LFG is called the *head*, and a daughter marked ↑=↓ is called a *head daughter.* As with c-structure, f-structure can be represented by an equivalence relation, $\_ =_{\mathtt{hd}} \_$, on `Syn`. The purpose of $=_{\mathtt{hd}}$, in all three frameworks, is to group together the various properties shared by mother and head daughter. If the value of `subj` is one such property (and it is), then $r =_{\mathtt{hd}} s$ should mean $\mathtt{subj}(r)$ is defined only when $\mathtt{subj}(s)$ is, when they should have the same value. This can be expressed by the axiom $r =_{\mathtt{hd}} s \rightarrow (\mathtt{subj}(r) = t \leftrightarrow \mathtt{subj}(s) = t)$, which may be abbreviated $r =_{\mathtt{hd}} s \rightarrow \mathtt{subj}(r) \doteq \mathtt{subj}(s)$. Note $=_{\mathtt{hd}}$ also groups together properties expressed by equivalence relations, such as $=_{\mathtt{agr}}$. LFG puts everything except category into f-structure.

(3.17) (Head Sharing)

HEAD = **derive from** PEQV **by** $[\_ =_{\mathtt{hd}} \_ \longmapsto \_ \equiv \_]$

(3.18) (In LFG, intensions consist only of c-structure and f-structure)

CF = **enrich** HEAD + CAT
by **axiom** $r =_{\mathtt{cat}} s \wedge r =_{\mathtt{hd}} s \rightarrow r = s$

(3.19) (Head functions)

HFN = **enrich** HEAD
by **opn** `hfn` : `Syn`$\dot{\rightarrow}$`Syn`
axiom $r =_{\mathtt{hd}} s \rightarrow \mathtt{hfn}(r) \doteq \mathtt{hfn}(s)$

(3.20) (Subject is a head function)

SUBJ = **derive from** HFN
by $[\mathtt{subj} \longmapsto \mathtt{hfn}]$

(3.21) (Object is a head function)

$$\begin{aligned} \mathsf{OBJ} \quad &= \quad \textbf{derive from } \mathsf{HFN} \\ &\quad\ \ \textbf{by } [\mathtt{obj} \longmapsto \mathtt{hfn}] \end{aligned}$$

(3.22) (Other head properties)

$$\begin{aligned} \mathsf{HPRP} \quad &= \quad \textbf{enrich } \mathsf{HEAD} + \mathsf{PEQV} \\ &\quad\ \ \textbf{by } \textbf{axiom } r \equiv r \wedge r =_{\mathtt{hd}} s \rightarrow r \equiv s \end{aligned}$$

(3.23) (Agreement is a head property)

$$\begin{aligned} \mathsf{HAGR} \quad &= \quad \textbf{derive from } \mathsf{HPRP} \\ &\quad\ \ \textbf{by } [\_ =_{\mathtt{agr}} \_ \longmapsto \_ \equiv \_] \end{aligned}$$

references: CAT (3.11), PEQV (3.9)

Functions like `subj` and `obj` shared by mothers and head daughters can be defined using HFN, as in SUBJ and OBJ. Properties shared by mother and head daughter which take the form of an equivalence relation, like $=_{\mathtt{agr}}$ (so that if $m =_{\mathtt{hd}} h$, then also $m =_{\mathtt{agr}} h$), may be defined using HPRP. Note HAGR refines AGR (3.12), modulo signature ($\mathsf{HAGR} \sqsubseteq \mathsf{AGR}$).

Rules like (3.16) above fit easily into the PATR-II format (except we use $=_{\mathtt{cat}}$ instead of `cat`, and so on):

$$\begin{aligned} n =_{\mathtt{cat}} \mathrm{NP} \quad &\wedge v =_{\mathtt{cat}} \mathrm{VP} \wedge s =_{\mathtt{cat}} \mathrm{S} \\ &\wedge n = \mathtt{subj}(s) \wedge v =_{\mathtt{hd}} s \rightarrow s \mapsto n \cdot v \end{aligned}$$

Agreement is achieved in LFG by specifying in lexical entries the agreement properties which values of functions like `subj` must have, when they are defined. Lexical entries also specify which such functions *must* be defined, and which *may* (optionally) be defined. Then any function not mentioned in the lexical entry must, by default, *not* be defined. In our language of partial operations, it is more convenient to have the default be the optional case: we say which functions *must* be defined ($\mathbf{D}\mathtt{subj}(v)$), and which must *not* ($\neg\mathbf{D}\mathtt{obj}(v)$). If we say nothing about a function, it may or may not be defined. (Recall that $\mathbf{D}t$, for any term t, is shorthand for $t = t$, which is satisfied in a model if and only if the term t denotes an object in the appropriate carrier).

$$v =_{\mathtt{cat}} \text{VP} \wedge (n = \mathtt{subj}(v) \rightarrow n =_{\mathtt{agr}} \mathtt{he})$$
$$\wedge\, \mathbf{D}\mathtt{subj}(v) \wedge \neg\mathbf{D}\mathtt{obj}(v) \wedge \neg\mathbf{D}\mathtt{obj2}(v) \wedge \ldots$$
$$\rightarrow v \mapsto \mathtt{smiles}$$

So to a (very simplified) example grammar.

(Some constants).

LC = **sort** Syn
opns S,NP,VP,3S :→ Syn

(LFG toy grammar specification).

LTG0 = **extend** LC+CF+SUBJ+OBJ+PWDS +ATOM+AGR+LXN
by **pred** _ ↦ _ ⊆ Syn × Syn*
axiom $n =_{\mathtt{cat}} \text{NP} \wedge v =_{\mathtt{cat}} \text{VP} \wedge s =_{\mathtt{cat}} \text{S} \wedge$
$n = \mathtt{subj}(s) \wedge v =_{\mathtt{hd}} s \rightarrow s \mapsto n \cdot v$
axiom $n =_{\mathtt{cat}} \text{NP} \wedge n =_{\mathtt{agr}} 3\text{S} \wedge \neg\mathbf{D}\mathtt{subj}(n)$
$\wedge\, \neg\mathbf{D}\mathtt{obj}(n) \rightarrow n \mapsto \mathtt{john}$
axiom $v =_{\mathtt{cat}} \text{VP} \wedge (n = \mathtt{subj}(v) \rightarrow n =_{\mathtt{agr}} 3\text{S})$
$\wedge\, \mathbf{D}\mathtt{subj}(v) \wedge \neg\mathbf{D}\mathtt{obj}(v) \rightarrow v \mapsto \mathtt{smiles}$

references: PWDS (3.3), ATOM (3.5), AGR (3.12), LXN (2.2)

This example makes no abstract use of bar-level, so I will not bother to use XBAR (3.15). Once again, this specification says nothing to guarantee the existence of intensions with the required properties. In producing a refinement to rectify this, we could adopt the strategy suggested for PATR-II of adding a single constructor. Unfortunately in the obvious formulation, the only elements of Syn which have obj undefined are the constants. We need the ability to build intensions in which obj does not figure, a system of *open arity*. We can employ a place-holding constant, 0 say. There is no need to include a place in the constructor for hd, because it can be defined in terms of the other features.

LAT = **extend sort** Lex

by **sort** Atom

opns S,NP,VP,3S :⟶ Atom

(Syn program)

LSYN = **derive from**

extend sorts Lex,Atom **by**

sorts Avs, Atom0, Syn, Syn0

opn $0 :\longrightarrow$ Atom0

opn $\langle \_,\_,\_,\_\rangle$: Atom0 × Atom0 × Syn0 × Syn0 → Avs

opns subj, obj : Avs → Syn0

pred $\_ =_{\mathrm{hd}} \_ \subseteq$ Syn × Syn

pred $\_ =_{\mathrm{cat}} \_ \subseteq$ Syn × Syn

pred $\_ =_{\mathrm{agr}} \_ \subseteq$ Syn × Syn

axiom Lex, Atom, Avs ⊆ Syn ⊆ Syn0

axiom Atom ⊆ Atom0 ⊆ Syn0

axiom $\langle x, y_2, y_3, y_4\rangle =_{\mathrm{cat}} \langle x, z_2, z_3, z_4\rangle$

axiom $\forall a{:}\mathrm{Atom}\,.\, a =_{\mathrm{cat}} \langle a, x_2, x_3, x_4\rangle =_{\mathrm{cat}} a$

axiom $\langle y_1, x, y_3, y_4\rangle =_{\mathrm{agr}} \langle z_1, x, z_3, z_4\rangle$

axiom $\forall a{:}\mathrm{Atom}\,.\, a =_{\mathrm{agr}} \langle x_1, a, x_3, x_4\rangle =_{\mathrm{agr}} a$

axiom $\mathrm{subj}\langle x_1, x_2, x_3, x_4\rangle = x_3$

axiom $\mathrm{obj}\langle x_1, x_2, x_3, x_4\rangle = x_4$

axiom $\langle y, x_2, x_3, x_4\rangle =_{\mathrm{hd}} \langle z, x_2, x_3, x_4\rangle$

by $[\not\longmapsto 0, \mathrm{Avs}, \mathrm{Atom0}, \mathrm{Syn0}, \langle \_,\_,\_,\_\rangle]$

Recall $[\not\longmapsto 0, \mathrm{Avs}, \mathrm{Atom0}, \mathrm{Syn0}, \langle \_,\_,\_,\_\rangle]$ specifies the signature morphism which "forgets" 0, Avs, Atom0, Syn0, and $\langle \_,\_,\_,\_\rangle$, and leaves everything else unchanged. Up to isomorphism, LTG0+LAT+LSYN has only one model.

Typically an LFG rule for verb phrases in English might run

VP	⟶	V	(NP)	(NP)...
		$\uparrow{=}\downarrow$	$\uparrow$ obj $=\downarrow$	$\uparrow$ obj2 $=\downarrow$

where the brackets around NP indicate an optional argument. LTG0 ignored the VP rule, and instead gave the lexical entry for **smiles**'s category as VP, when, according to LFG orthodoxy, it should be listed as a V with `obj`, `obj2` and so on undefined. This was purely for simplicity: in a simple "core" account like LTG0, there is no difficulty in treating optionality as an abbreviatory convention for a collection of rules, one for each choice of allowed argument sets. The obvious formulation for the case when the argument is missing would be

$$\forall vv' \,.\, v =_{\mathtt{cat}} \mathrm{V} \wedge v' =_{\mathtt{cat}} \mathrm{VP} \wedge v =_{\mathtt{hd}} v' \rightarrow v' \mapsto v \;.$$

But this would recognise $v' \mapsto^*$ **hates** where $v' =_{\mathtt{cat}}$ VP and $\mathtt{obj}(v')$ has a value which does not appear in the parse. That is, *hates* by itself would be recognised as a verb phrase, even though its object does not appear with it. There is nothing to prevent the V-by-itself option applying even when `obj` is defined. We can easily prevent this from happening by changing our formulation of the no-object rule to

$$\forall vv' \,.\, v =_{\mathtt{cat}} \mathrm{V} \wedge v' =_{\mathtt{cat}} \mathrm{VP} \wedge v =_{\mathtt{hd}} v' \wedge \neg \mathbf{D}\mathtt{obj}(v) \rightarrow v' \mapsto v \;.$$

This works perfectly well for most purposes, but not for some more "difficult" constructions, such as relative clauses. According to the treatment in Kaplan and Zaenen (1987), a relative construction such as *the man Kim hates*, there is a v' such that $v' \mapsto^*$ **hates**, $v' =_{\mathtt{cat}}$ VP, and $\mathtt{obj}(v')$ is associated with an intension for *the man*, even though *the man* does not appear in the parse of v'. One way of viewing this treatment is to say that we do have $v' \mapsto^*$ **hates**, even when $\mathtt{obj}(v')$ is defined, but when we look for intensions s of matrix (non-embedded) sentences, we not only look for $s =_{\mathtt{cat}}$ S, but also for parses where every **subj** (`obj`, `obj2`...) value is associated with a node of the parse. In standard presentations of LFG (Bresnan 1982), this is an artifact of the procedural definition of the correspondence between c-structure and f-structure. To directly encode this declaratively, we would have to explicitly build up whole trees as syntactic objects, and introduce syntax to keep track of which rule they were constructed by and which obligations they discharge where. This may be even further complicated because (contrary to assumptions detailed in the paper) Kaplan and Zaenan's treatment

forces cyclicity in the model structures. Yet the treatment above is good for the "core" part of LFG. The desire to extend this to unbounded dependencies has greatly complicated the essentially simple core treatment, making it difficult to understand and model-theoretically dubious. I think this illustrates the difficulties and dangers of the "all-or-nothing" approach to building complex models, and suggests the potential advantages offered by a truly modular approach. In Section 5.3 I suggest that the role of equality in the intensional domain ought to be to encode distributional equivalence. This is not consistent with having intensions be whole trees. I therefore suggest (also in Section 5.3) a class of models for LFG in which intensions consist of atomic category labels, plus an f-structure which distinguishes between values taken internally or externally (relative to a tree rooted at such an intension). These models will be amenable to the modular treatment of unbounded dependencies presented later in Chapter 5.

3.4 Immediate Dominance and Linear Precedence

GPSG, and later HPSG and even some recent versions of LFG, use some form of *immediate dominance/linear precedence* (IDLP) format. The idea here is that immediate constituency in natural language can always be described by specifying these two concepts (immediate dominance and linear precedence) separately. Linear precedence refers to the possible sequences in which a collection of intensions can occur if they are to be the daughters of some local tree. Natural languages (or at least, most grammars for natural languages) appear to exhibit strong regularities as to what may precede what, at the level of sisters. In English, for instance, noun phrases precede any sister verb phrases. So

sentence:	NP · VP	*Jesus wept*
verb phrase:	**believe** · NP · VP	*believe Jesus wept*
verb phrase:	**persuade** · NP · VP	*persuaded John to come*

and so on. But IDLP format embodies a stronger claim, that we can classify all strings of intensions according to whether they are OK for linear precedence (`lpok`), or whether they are not. If $\neg\texttt{lpok}(x)$, there is no r such that $r \mapsto x$. This is intended to formulate potential ordering of daughters, *independent* of what the mother may be. So if $r \mapsto x$, then of course $\texttt{lpok}(x)$, but if in addition x' is a permutation of x such that $\texttt{lpok}(x')$, then also $r \mapsto x'$. This might be expressed as follows:

(3.24) (Possible daughter sequences)

LPOK = **enrich** SYN* **by**

pred $\texttt{lpok} \subseteq \texttt{Syn}*$

(3.25) (Permutations)

PERM = **extend** SYN* **by**

pred $\_ \bowtie \_ \subseteq \texttt{Syn}* \times \texttt{Syn}*$

axiom $x \cdot y \cdot z \bowtie y \cdot x \cdot z$

axiom $x \bowtie y \wedge y \bowtie z \rightarrow x \bowtie z$

(3.26) LP = **enrich** LPOK + PERM + GMR

by **axiom** $r \mapsto x \rightarrow \texttt{lpok}(x)$

axiom $r \mapsto x \wedge x \bowtie x' \wedge \texttt{lpok}(x') \rightarrow r \mapsto x'$

references: SYN* (2.3), GMR (2.4)

LP$\sqsubseteq$GMR, because every model of LP includes a model of GMR (2.4). In fact, LP $\sqsubset$ GMR, because there are models of GMR for which no definition of `lpok` will make the axiom of LP hold true. So LP embodies a universal claim about natural language, in that it excludes certain forms of grammar, which (the claim runs) natural language does not require. For example, if a, b, c, d are in `Syn`, it disallows

any grammar in which $a \mapsto c \cdot d$, $a \mapsto d \cdot c$, and $b \mapsto c \cdot d$, but not $b \mapsto d \cdot c$.[1] Recall the PSG model S (2.7):

$S(\texttt{Lex})$ = {saw,the,man,with,the,telescope}
$S(\texttt{Syn})$ = $S(\texttt{Lex}) \cup$ {VP,VT,NP,PP}
$S(\_ \cdot \_)$ given by concatenation
$S(\_ \mapsto \_)$ given by

VP $\mapsto$ VT NP
VP $\mapsto$ VT NP PP
NP $\mapsto$ NP PP
NP $\mapsto$ the man
VT $\mapsto$ saw
PP $\mapsto$ with the telescope

S can, by the existence of a suitable interpretation of `lpok`, be shown to be a model of $(\mathsf{LXN}+\mathsf{LP})|_{\mathsf{Sig[S]}}$ (2.2, 2.6). As there do not exist r, r', x, x' in S such that $r \mapsto x$, $r' \mapsto x'$, $x \bowtie x'$ and $x \neq x'$, we can be sure that one such interpretation would be the set formed by all the x such that $r \mapsto x$:

(3.27) {VT NP, VT NP PP, NP PP, the man, saw, with the telescope}

IDLP is generally associated (as for instance in GKPS) with another claim, that `lpok` depends only on what pairs (sequences of length two) are allowed.

(3.28) (Binary LP)

LPBIN = **enrich** LP **by axiom** $\texttt{lpok}(w) \leftrightarrow$
$\forall st{:}\texttt{Syn}, xyz{:}\texttt{Syn}* \,.\, w = x \cdot s \cdot y \cdot t \cdot z \rightarrow \texttt{lpok}(s \cdot t)$

---

[1] In practice, quite a lot of the work done today on developing these frameworks is on modifying this notion of LP, because there are various phenomena and languages which do *not* seem to be amenable to this treatment. *Crossing dependencies*, addressed in Chapter 6, is such a phenomenon, though I take a somewhat different approach to solving the problem.

This trivially gives $\texttt{lpok(e)}$ and $\forall r : \texttt{Syn}.\texttt{lpok}(r)$ in any model of LPBIN. If a, b, c, d, e are of sort Syn, LPBIN would (for example) disallow any grammar in which $a \mapsto c \cdot d$, $a \mapsto d \cdot c$, and $b \mapsto c \cdot d \cdot e$, but not $b \mapsto d \cdot c \cdot e$. S has no rules of this form and can be shown to be an implementation (modulo signature) of LPBIN+LXN (2.2) by the demonstration of an appropriate lpok. As there do not exist r, r', x, x' in S such that $r \mapsto x$, $r' \mapsto x'$, $x \bowtie x'$ and $x \neq x'$, we just need to show that there is no pair of intensions which appear in different orders in different rules. This shows that the closure of our previous choice of lpok under the axiom of LPBIN will suffice:

(3.29) {ϵ, VT, NP, VP, PP, the, man, saw, with, telescope,
VT NP, VT PP, NP PP, the man, with the, with telescope, the t.s.,
VT NP PP, with the telescope}

(where ϵ is the empty string). In GKPS precedence is expressed by a relation $\_ \prec \_$, where $s \prec t$ means s-daughters *must* precede t-daughters, in any local tree in which both occur. This is clearly equivalent to $\neg\texttt{lpok}(t \cdot s)$. GKPS also has $\_ \prec \_$ antisymmetric and transitive, but I prefer not to follow suit here. Having $s \prec t$ and $t \prec s$ would simply mean s and t could never appear as sisters. So if $\prec$ is antisymmetric, and we want to forbid s and t ever appearing as sisters, we would have to express this by $\neg\exists r{:}\texttt{Syn}, xyz{:}\texttt{Syn}*\,.\, r \mapsto x \cdot s \cdot y \cdot t \cdot z \vee r \mapsto x \cdot t \cdot y \cdot s \cdot z$, rather than the simpler $s \prec t \wedge t \prec s$. The restriction to antisymmetry on $\_ \prec \_$ seems doubly peculiar in that we are still allowed to specify that two instances of the *same* intension cannot appear as sisters ($t \prec t$).

Insisting on transitivity seems to have been mainly a device to simplify specifications, and as such is an implementation technique. Unlike antisymmetry, transitivity does restrict the class of grammars. Of course if $r \prec s$ and $s \prec t$, and we have sister instances of r, s and t, then we must in any case have r preceding t. The only time it makes any difference is when we have r and t sisters, but not s. In that case, r must precede t in a model where $\_ \prec \_$ is transitive, but not necessarily otherwise. As far as I am aware, no work has been done on whether restricting to transitive models is the right thing to do for all languages. If we look

at the $\prec$-transitivity condition in terms of `lpok`, we begin to see how peculiar it really is.

$$\begin{aligned}
&\forall rst\,.\, r \prec s \wedge s \prec t \rightarrow r \prec t \\
&\qquad \equiv \quad \forall rst\,.\, \neg\mathtt{lpok}(r \cdot s) \wedge \neg\mathtt{lpok}(s \cdot t) \rightarrow \neg\mathtt{lpok}(r \cdot t) \\
&\qquad \equiv \quad \forall rst\,.\, \neg(\mathtt{lpok}(r \cdot s) \vee \mathtt{lpok}(s \cdot t)) \rightarrow \neg\mathtt{lpok}(r \cdot t) \\
&\qquad \equiv \quad \forall rst\,.\, (\mathtt{lpok}(r \cdot s) \vee \mathtt{lpok}(s \cdot t)) \vee \neg\mathtt{lpok}(r \cdot t) \\
&\qquad \equiv \quad \forall rt\,.\, \mathtt{lpok}(r \cdot t) \rightarrow \forall s.\mathtt{lpok}(r \cdot s) \vee \mathtt{lpok}(s \cdot t)
\end{aligned}$$

Finally, IDLP format is usually presented via *immediate dominance* rules, which are like the immediate constituency rules $\_ \mapsto \_$, except that the right hand side, instead of being an ordered collection (sequence or string) of intensions, is an unordered one (a multiset or bag). This summarises the information which, in $\_ \mapsto \_$, is duplicated across permutations, and is neater to work with than ensuring permutation invariance (modulo `lpok`) all the time.

(3.30) (Free Abelian Monoids — multisets)

FGAM = **extend sort** `Sub` **by**

sort `Gpd`

opn $\_ \cdot \_ : \mathtt{Gpd} \times \mathtt{Gpd} \rightarrow \mathtt{Gpd}$

opn $\mathtt{e} :\rightarrow \mathtt{Gpd}$

axiom $\mathtt{Sub} \subseteq \mathtt{Gpd}$

axiom $x \cdot (y \cdot z) = (x \cdot y) \cdot z$

axiom $x \cdot y = y \cdot x$

axiom $x \cdot \mathtt{e} = \mathtt{e} \cdot x = x$

(3.31) (Multisets over `Syn`)

MSYN = **derive from** FGAM

by $[\mathtt{Syn} \longmapsto \mathtt{Sub}, \mathtt{MSyn} \longmapsto \mathtt{Gpd}, 0 \longmapsto \mathtt{e}, \_ + \_ \longmapsto \_ \cdot \_]$

(3.32) (Immediate Dominance)

IMD = **enrich** MSYN

by **pred** $\_ \lhd \_ \subseteq \mathtt{Syn} \times \mathtt{MSyn}$

axiom $\neg s \lhd s$

(3.33) (Project strings onto multisets)

MIX = **extend** SYN* + MSYN **by**

opn $|\_| : \mathtt{Syn}* \rightarrow \mathtt{MSyn}$

axiom $|\mathsf{e}| = 0$

axiom $\forall r{:}\mathtt{Syn}, x{:}\mathtt{Syn}* \,.\, |r \cdot x| = r + |x|$

(3.34) IDLP = **enrich** GMR + LPOK + IMD + MIX

by **axiom** $r \mapsto x \leftrightarrow r \lhd |x| \wedge \mathtt{lpok}(x)$

$r \lhd u$ represents immediate dominance. It indicates that there are (or may be) local trees with mother r and daughters u (in no particular order). The intensions in a string x together form a unique multiset $|x|$. For every permutation x' of x, $|x'| = |x|$. So $r \mapsto x$ is exactly the same as $r \lhd |x|$, plus $\mathtt{lpok}(x)$. Every model expressible using permutations is expressible using immediate dominance, and vice-versa. An IDLP equivalent of S (2.7) might have, in addition to **lpok** in (3.29), the following:

VP $\lhd$ VT+NP
VP $\lhd$ VT+NP+PP
NP $\lhd$ NP+PP
NP $\lhd$ the+man
TV $\lhd$ saw
PP $\lhd$ with+the+telescope

The version of IDLP in GPSG or HPSG, will be an implementation of IDLP + LPBIN.

3.5 GPSG

The components of GPSG are

(3.35) immediate dominance rules e.g. $S \longrightarrow X^2, H[-SUBJ]$

(3.36) linear precedence rules e.g. $[SUBCAT] \prec \sim[SUBCAT]$

(3.37) feature co-occurrence restrictions e.g. $[AGR] \supset [+V, -N]$

(3.38) feature specification defaults e.g. $[PFORM] \supset [BAR\ 0]$

(3.39) universal feature instantiation principles, and

(3.40) metarules

In GPSG and HPSG, $=_{\mathtt{maj}}$ is a head property.

(3.41) $\mathsf{HMAJ} = \textbf{derive from } \mathsf{HPRP}$
$\textbf{by } [\_ =_{\mathtt{maj}} \_ \longmapsto \_ \equiv \_]$

Note $\mathsf{HMAJ} \sqsubseteq \mathsf{MAJ}$ (3.13). Thus GPSG will refine $\mathsf{HMAJ} + \mathsf{IDLP} + \mathsf{LPBIN} + \mathsf{XBAR}$ (3.34, 3.28, 3.15). Specification and implementation of GPSG can proceed in much the way outlined in the LFG example, except that, rather than specifying phrase structure rules using $\mapsto$, we encode ID (immediate dominance) rules using $\lhd$, and LP (linear precedence) rules using $\prec$. Most difficulties will arise with GPSG's use of defeasible default inheritance. For simplicity, I will again assume a feature MAJ, and just write [MAJ V] (or [V]) rather than $[+V, -N]$, as in GKPS.

Since *feature co-occurrence restrictions* (FCRs) filter possible intensions, it will be useful to consider them first. FCRs are a means by which we exclude certain objects from the grammar proper. For instance, the example FCR in (3.37) means that if the agreement feature is defined on an intension, then it ought to have major category V. I will suppose the existence of a specification GFEAT which sets up

the feature system in terms of partial equivalences, perhaps using a constructor like `f` in the specification PSYN (3.6) of PATR-II intensions. However I will now be proceeding to exclude some of these intensions, and I will want to refer to this reduced collection of intensions as `Syn`, so GFEAT would actually need to use a different sort name, say `Syn'`, such that `Syn` $\subseteq$ `Syn'`. So rather suppose GFEAT is akin to

$$\textbf{derive from PSYN by } [\texttt{Syn}' \longmapsto \texttt{Syn}] \qquad \text{(PSYN: 3.6)}$$

We might then proceed to give predicates for each restriction.

$$\begin{array}{lcl}
\mathsf{FCR1} & = & \textbf{enrich GFEAT} \\
 & & \quad \textbf{by pred } \texttt{fcr1} \subseteq \texttt{Syn}' \\
 & & \quad\quad \textbf{axiom} \ldots \\
 & \vdots & \\
\multicolumn{3}{l}{(\text{The FCR [AGR]} \supset [+\text{V}, -\text{N}] \text{ of (3.37)})} \\
\mathsf{FCR12} & = & \textbf{enrich GFEAT} \\
 & & \quad \textbf{by pred } \texttt{fcr12} \subseteq \texttt{Syn}' \\
 & & \quad\quad \textbf{axiom } \texttt{fcr12}(r) \leftrightarrow (\mathbf{Dagr}(r) \rightarrow r =_{\texttt{maj}} \text{V}) \\
 & \vdots & \\
\mathsf{FCRs} & = & \textbf{enrich FCR1} + \ldots + \mathsf{FCR12} + \ldots \textbf{ by} \\
 & & \quad\quad \textbf{sort Syn} \\
 & & \quad\quad \textbf{axiom Syn} \subseteq \texttt{Syn}' \\
 & & \quad\quad \textbf{axiom } r\texttt{:Syn} \leftrightarrow \texttt{fcr1}(r) \wedge \ldots \wedge \texttt{fcr12}(r) \wedge \ldots
\end{array}$$

Perhaps the most obvious example of defeasible inheritance is in the inclusion of bar level among head properties. This seems very strange, as we expect bar level in general to increase from daughter to mother. Inspection of the rules given in GKPS shows that most in fact specify the (different) bar levels taken by mother and head daughter. The idea is that when this is done (i.e. when the rules explicitly contradict the usual convention), the "default" that mother and daughter share bar level is overridden. But as we have just said, the majority of rules (and

in particular, all the lexical rules) do make use of this explicit specification of bar levels, so really we might just as well cease to look on bar level as a head property, and instead add explicit equality of bar level to those few rules which allow it. For instance, the ID rule in (3.35) would then become $\mathrm{S} \longrightarrow \mathrm{X}^2, \mathrm{H}^2[-\mathrm{SUBJ}]$. The superscripted 2 indicates bar level, X can stand for any intension (in this case, any intension of bar level two), H indicates the head daughter, and S is an abbreviation for $[\mathrm{V}, +\mathrm{SUBJ}, \mathrm{BAR2}]$. Similar comments apply to the placement of the `subj` feature among head properties (in GPSG, [+SUBJ] merely serves to distinguish sentences from VPs). Using partial equivalences and so on, the example of (3.35) becomes part of the specification of immediate dominance:

GID = **extend** FCRs + MSYN **by**

pred $\_ \lhd \_ \subseteq \mathtt{Syn} \times \mathtt{MSyn}$

$(\mathrm{S} \longrightarrow \mathrm{X}^2, \mathrm{H}^2[-\mathrm{SUBJ}])$

axiom $v =_{\mathtt{maj}} \mathrm{VP} =_{\mathtt{maj}} s \wedge$

$v =_{\mathtt{bar}} \mathrm{VP} =_{\mathtt{bar}} s =_{\mathtt{bar}} x \wedge$

$v =_{\mathtt{subj}} \mathrm{VP} \neq_{\mathtt{subj}} s \wedge$

$v =_{\mathtt{hd}} s \rightarrow s \lhd x + v$

$\vdots$

references: MSYN (3.31)

The treatment of linear precedence presents no great difficulties. The example of (3.36) will form part of such a treatment:

GPREC = **extend** FCRs **by**

pred $\_ \prec \_ \subseteq \mathtt{Syn} \times \mathtt{Syn}$

(the LP rule [SUBCAT] $\prec \sim$[SUBCAT] of (3.36):)

axiom $r =_{\mathtt{subcat}} r \wedge s \neq_{\mathtt{subcat}} s \rightarrow r \prec s$

$\vdots$

LPREL = **enrich** LPBIN **by**

pred $\_ \prec \_ \subseteq \mathtt{Syn} \times \mathtt{Syn}$

$$\textbf{axiom } \texttt{lpok}(r \cdot s) \leftrightarrow \neg s \prec r$$

references: LPBIN (3.28)

The place where defeasible defaults are most heavily used is in the *feature specification defaults* (FSDs). These make little sense at all in a loose specification. Coming to implementation, it is notoriously difficult to model these defaults computationally in a uniform way. It is not even clear that the system as outlined does what was intended. However the intention is that they describe some core, or default, behaviour, which can be overridden in the case of related, but slightly different, exceptional behaviour. For instance, the default in (3.38) is intended to prevent the feature PFORM from being defined, except when a value is imposed, for instance by an ID rule like the following one for *give*:

VP $\longrightarrow$ H[3], NP, PP[PFORM *to*]

One way to handle this is, instead of having an extractor for PFORM, have an extra constructor

$$\textbf{opn } \texttt{pform} : \texttt{Syn} \rightarrow \texttt{Syn},$$

and when specifying immediate dominance, instead of requiring a PP[PFORM *to*], insist on an intension of the form $\texttt{pform}(p)$, where p is a PP. We would then need to specify also the behaviour of the intensions of the form $\texttt{pform}(r)$ with respect to dominance:

$$r \lhd t + x \rightarrow \texttt{pform}(r) \lhd \texttt{pform}(t) + x.$$

(This is an oversimplification, as this formulation would not cope with multiply-headed local trees). Notice that the sort of constructions being undertaken here depend only on a few details of the remainder of the system, suggesting that we may be able to make a parametrised construction, which could be applied to any "core" system which could supply notions like PP and immediate dominance. All

the matters dealt with by FSDs in GKPS ought to be susceptible to such modular treatments. In Chapter 5 we will see in detail how this strategy may be employed to deal with some unbounded dependency phenomena.

The function **agr** acts in much the same way as **subj** in LFG to give agreement, in that lexical entries for verbs will specify what sort of **agr** values they may have. However, instead of having phrase structure rules specify where this value must be found, this is dealt with by the *control agreement principle.* The control agreement principle is supposed to mediate with a semantic treatment to identify when semantic functors require syntactic agreement (as between a subject and a verb phrase). This is quite a modular approach, as it is parameterised by a simple (semantic) type system in which words are assigned semantic types from some domain, **Sem** say, containing a constructor $\langle \_, \_ \rangle$: **Sem** $\times$ **Sem**$\dot{\rightarrow}$**Sem**. The central idea is that if $\mathtt{typ}(s) = \langle \mathtt{typ}(t), \mathtt{typ}(r) \rangle$, we never allow $r \lhd s + t + x$ unless $\mathtt{agr}(s) = t$. Again this is complicated by all sorts of matters which would probably be better dealt with independently. It is also extended to deal with "control" verbs. The system is only sketched and is largely of interest as a putative cross-linguistic principle, and in interfacing with semantics. It is not heavily used in the English fragment of GKPS, where it is just supposed to deal with subject agreement, and agreement in control verbs. For this reason I will not go into details of its treatment. Similarly, the complications on the *head feature convention* (detailing just which head features should be inherited when) are not relevant in a core treatment, and neither is the *foot feature principle.* These three are called *universal feature instantiation principles.* They act as a filter on the immediate dominance rules, disallowing particular instances. One way to achieve this filtering would be to specify four separate predicates

$$\textbf{preds } \mathtt{idr}, \mathtt{cap}, \mathtt{ffp}, \mathtt{hfc} \subseteq \mathtt{Syn} \times \mathtt{MSyn}$$

for the immediate dominance rules and the three universal feature instantiation principles, and then define

$$r \lhd x \leftrightarrow \mathtt{idr}(r, x) \wedge \mathtt{cap}(r, x) \wedge \mathtt{ffp}(r, x) \wedge \mathtt{hfc}(r, x)$$

but (as intimated) I shall not consider the details of specifying these predicates.

The last component of GPSG is the *metarules*. These deal with matters such as passivisation, unbounded dependencies, conjunction, and so on. Their specification need not offer any special challenge, but as most of these constructions do not depend heavily on the fine details of the core treatment, they are exactly the sorts of things we should consider specifying and implementing in a more modular fashion.

(GPSG Core)
GCOR = GID + GPREC + LPREL + IDLP

3.6 HPSG

The components of HPSG are

(3.42) declarations, of features appropriate on a sort, with their own associated sorts, e.g.

$$\left[\begin{array}{l}\texttt{syn: SyntacticValue}\\ \texttt{dtrs: ConstituentStructure}\end{array}\right] : \texttt{PhrasalSign}$$

(3.43) grammatical principles, e.g.

$$\left[\texttt{dtrs: HeadedStructure}\right] \Rightarrow \left[\begin{array}{l}\texttt{syn:loc:head: } h\\ \texttt{dtrs:head\_dtr:syn:loc:head: } h\end{array}\right]$$

(3.44) grammatical rules, e.g.

$$\left[\begin{array}{l} \texttt{syn:loc:subcat: } \langle\,\rangle \\ \texttt{dtrs: } \left[\begin{array}{l} \texttt{head\_dtr:syn:loc:lex: } - \\ \texttt{comp\_dtrs: } \langle [\,] \rangle \end{array}\right] \end{array}\right]$$

(3.45) the lexical hierarchy, which I won't consider here.

Specification and implementation of HPSG can proceed in very much the way of the multi-sorted feature grammar SPP (3.8). As stated there, when we list the features appropriate for a given type, we are really giving the arity of a constructor of the type, and extractors on it. For instance from (3.42) we read the need for a constructor from `SyntacticValue` $\times$ `ConstituentStructure` to `PhrasalSign`, with corresponding extractors `syn` and `dtrs`:

sorts	`PhrasalSign,LexicalSign,Syn`
axiom	`PhrasalSign, LexicalSign` $\subseteq$ `Syn`

sorts	`SyntacticValue,ConstituentStructure`
opn	`phrasal_sign` : `SyntacticValue` $\times$ `ConstituentStructure` $\rightarrow$ `PhrasalSign`
opn	`syn` : `PhrasalSign` $\rightarrow$ `SyntacticValue`
opn	`dtrs` : `PhrasalSign` $\rightarrow$ `ConstituentStructure`
axiom	`syn(phrasal_sign(`s, d`))` $= s$
axiom	`dtrs(phrasal_sign(`s, d`))` $= d$

(I have left out the PHON and SEM attributes for simplicity). Now any object of sort `ConstituentStructure` is either a headed structure or a coordinate structure.

sorts	`HeadedStructure,CoordinateStructure`
axiom	`HeadedStructure, CoordinateStructure` $\subseteq$ `ConstituentStructure`

`HeadedStructure` in turn has attributes `head_dtr`, `comp_dtrs` and `adj_dtrs`

opn	`headed_structure` : `Syn` × `Syn*` × `MSyn` → `HeadedStructure`
opn	`head_dtr` : `HeadedStructure` → `Syn`
opn	`comp_dtrs` : `HeadedStructure` → `Syn*`
opn	`adj_dtrs` : `HeadedStructure` → `MSyn`
axiom	$\texttt{head\_dtr}(\texttt{headed\_structure}(h, c, a)) = h$
axiom	$\texttt{comp\_dtrs}(\texttt{headed\_structure}(h, c, a)) = c$
axiom	$\texttt{adj\_dtrs}(\texttt{headed\_structure}(h, c, a)) = a$

and so on. Since HPSG explicitly builds parse trees into the intensions, by recording the unique daughters of any phrasal sign, GPSG's distinction between universal instantiation principles and feature co-occurrence restrictions is not needed. Like the latter, HPSG's *grammar principles* (which may be language specific or universal) restrict which signs can be considered intensions. We can use the same technique to enforce them. We change every constructor of `Syn` into a constructor of the supersort `Syn'`, and every extractor on `Syn` to one on `Syn'`. Crucially, we do not change any arguments to the constructors. To every principle P_i there corresponds an implicational formula $\phi_i(r)$, where r stands for a point at which P_i is to hold. Then $r{:}\texttt{Syn} \leftrightarrow \phi_1(r) \wedge \ldots \wedge \phi_i(r) \wedge \ldots$. For instance, the principle (3.43) says that if a mother's `dtrs` value is headed, then its syntactic value's local head features are shared with the local head features of the syntactic value of the head daughter. (So this principle just enforces the sharing of head features between mother and head daughter.) So if HFEAT is the result of specifying `Syn'`, we might have

```
PRINC1 = enrich HFEAT by
             pred princ1 ⊆ Syn'
             axiom princ1(m) ↔
                       (dtrs(m) : HeadedStructure
                           → head(loc(syn(head_dtr(dtrs(m)))))
                               = head(loc(syn(m))))
         ⋮
```

$$\begin{aligned}\mathsf{HPRINC} \;=\;& \textbf{enrich } \mathsf{PRINC1} + \mathsf{PRINC2} + \ldots \textbf{ by}\\ & \textbf{sort } \mathtt{Syn}\\ & \textbf{axiom } \mathtt{Syn} \subseteq \mathtt{Syn}'\\ & \textbf{axiom } r{:}\mathtt{Syn} \leftrightarrow \mathtt{princ1}(r) \wedge \mathtt{princ2}(r) \wedge \ldots\end{aligned}$$

Similarly to every rule R_i there corresponds a formula $\psi_i(r)$. For instance, the rule (3.44) describes intensions which have an empty **subcat** list, a single complement daughter, and a head daughter marked $\mathtt{lex}{:}-$. We might then get

$$\begin{aligned}\mathsf{HID} \;=\;& \textbf{extend } \mathsf{HPRINC} + \mathsf{MIX} \textbf{ by}\\ & \textbf{pred } \_ \lhd \_ \subseteq \mathtt{Syn} \times \mathtt{MSyn}\\ & \textbf{axiom } \mathtt{subcat}(\mathtt{loc}(\mathtt{syn}(r))) = \mathtt{e}\\ & \qquad \wedge\, \mathtt{comp\_dtrs}(\mathtt{dtrs}(r)){:}\mathtt{Syn}\\ & \qquad \wedge\, \mathtt{lex}(\mathtt{loc}(\mathtt{syn}(\mathtt{head\_dtr}(\mathtt{dtrs}(r))))) = -\\ & \qquad \rightarrow r \lhd \mathtt{head\_dtr}(\mathtt{dtrs}(r))\\ & \qquad\qquad + |\mathtt{comp\_dtrs}(\mathtt{dtrs}(r))|\\ & \qquad \vdots\end{aligned}$$

references: MIX (3.33)

Among the grammatical principles is supposed to be a *constituent order principle*, which ought to take the form of a map from `ConstituentStructure` to sequences of words. However P&S takes only a tentative stance on how this principle is to work, and so instead of attempting to specify this map, it gives a few linear precedence constraints, rather like those of Section 3.4, employing some *ad hoc* notation. The first of these is

(3.46) HEAD[LEX −] < [ ]

This is supposed to do the same job as the linear precedence statement (3.36) for GPSG:

[SUBCAT] $\prec \sim$[SUBCAT]

That is, it insists that lexical heads precede their sisters. The fact that (3.36) encodes this is due to the fact that all and only lexical heads have SUBCAT defined. The reason that this was *not* written

$$\mathrm{H}[-\mathrm{LEX}] \prec [\,]$$

is because the "H[]" notation is purely the formal syntax used to specify which daughter in an ID-rule is head, and it has no status outside ID-rules. In particular, there is nothing in the internal structure of the object modelling that intension which directly encodes the fact that it is head (although, fortuitously, definedness of SUBCAT happens to correspond to being a *lexical* head). Of course we could add a (non-head) feature to directly encode being a head (say [+HD]), and so insist that *different* intensions must be used in head positions than in non-head positions, but this is not necessary in GPSG (because we can use definedness of SUBCAT instead). However if we are to make sense of (3.46), we must either abandon the requirement of Section 3.4 that specification of LP be an ordering of intensions independent of context, or add to intensions some means (like the feature HD) of creating distinct intensions according to whether they are to act as heads, complements, or whatever. Taking the latter course, we can require (for instance) that Syn be divided into subsorts according to what kind of daughter it can act as, to be defined by the value of a feature dtyp. So HFEAT might run in part

HFEAT = **extend** $\emptyset$ **by**

sorts PhrasalSign, LexicalSign, Syn′

axiom PhrasalSign, LexicalSign $\subseteq$ Syn′

sorts SyntacticValue, ConstituentStructure

sorts HeadedStructure, CoordinateStructure

axiom HeadedStructure, CoordinateStructure $\subseteq$ ConstituentStructure

sort Dtyp

opns $\mathtt{hd}, \mathtt{cmp}, \mathtt{adj} :\rightarrow \mathtt{Dtyp}$

opn $\mathtt{phrasal\_sign} : \mathtt{SyntacticValue} \times \mathtt{ConstituentStructure} \times \mathtt{Dtyp} \rightarrow \mathtt{PhrasalSign}$

opn $\mathtt{syn} : \mathtt{PhrasalSign} \rightarrow \mathtt{SyntacticValue}$

opn $\mathtt{dtrs} : \mathtt{PhrasalSign} \rightarrow \mathtt{ConstituentStructure}$

opn $\mathtt{dtyp} : \mathtt{PhrasalSign} \rightarrow \mathtt{Dtyp}$

axiom $\mathtt{syn}(\mathtt{phrasal\_sign}(s, d, t)) = s$

axiom $\mathtt{dtrs}(\mathtt{phrasal\_sign}(s, d, t)) = d$

axiom $\mathtt{dtyp}(\mathtt{phrasal\_sign}(s, d, t)) = t$

sorts $\mathtt{Hd}, \mathtt{Cmp}, \mathtt{Adj}$

axiom $\mathtt{phrasal\_sign}(s, d, \mathtt{hd}) : \mathtt{Hd}$

axiom $\mathtt{phrasal\_sign}(s, d, \mathtt{cmp}) : \mathtt{Cmp}$

axiom $\mathtt{phrasal\_sign}(s, d, \mathtt{adj}) : \mathtt{Adj}$

opn $\mathtt{headed\_structure} : \mathtt{Hd} \times \mathtt{Cmp}* \times \mathtt{MAdj} \rightarrow \mathtt{HeadedStructure}$

opn $\mathtt{head\_dtr} : \mathtt{HeadedStructure} \rightarrow \mathtt{Hd}$

opn $\mathtt{comp\_dtrs} : \mathtt{HeadedStructure} \rightarrow \mathtt{Cmp}*$

opn $\mathtt{adj\_dtrs} : \mathtt{HeadedStructure} \rightarrow \mathtt{MAdj}$

axiom $\mathtt{head\_dtr}(\mathtt{headed\_structure}(h, c, a)) = h$

axiom $\mathtt{comp\_dtrs}(\mathtt{headed\_structure}(h, c, a)) = c$

axiom $\mathtt{adj\_dtrs}(\mathtt{headed\_structure}(h, c, a)) = a$

$\vdots$

Then we could express linear precedence, and an HPSG core, as follows:

HPREC = **extend** HFEAT **by**

pred $\_ \prec \_ \subseteq \mathtt{Syn} \times \mathtt{Syn}$

axiom $\forall h{:}\mathtt{Hd}, r{:}\mathtt{Syn}' \,.\, h \prec r$

$$\vdots$$

$$\mathsf{HCOR} \quad = \quad \mathsf{HID{+}HPREC{+}LPREL{+}IDLP}$$

3.7 Coda

The primary aim of this chapter has been to illustrate the applicability of formalised notions of refinement and implementation in specifying the formal properties of different linguistic programs, and illuminating some of their shared properties. In particular, I hope to have shown how feature-value grammars are naturally viewed as specifications in ordinary first-order logic, together with a handful of more-or-less standard implementation techniques. A standard technique for producing implementations of particular forms of specification amounts to the definition of a programming language. Such a technique is unlikely to itself be susceptible to specification in the way we have already seen, especially if it is to operate over a variety of different signatures. But clearly we would like some way of avoiding the duplication of effort involved with writing out essentially the same refinement every time want to write a new or extended grammar fragment. We may even want to be able to tie together partial implementations which use different techniques. This is the same as the problem of how to view a "real" program, in Prolog or ML say, as an implementation of an ASL specification, or how to tie together programs written in different languages. It was this sort of problem which led to the idea that a specification language could be parameterised by different *institutions* (logics). This is addressed in the next chapter.

Chapter 4

Institutions

It will be recalled that ASL was designed to be a kernel language, in terms of which more quotidian languages could be defined. Yet even here I have used a version which differs from "standard" ASL in offering partial operations. The changes to the underlying logical framework are in this case strictly generalisations, and add little syntactic burden to specifications which do not use partiality, but in general there is no single *right* choice of logical framework. This phenomenon is described in Goguen (1987), "One, None, A Hundred Thousand Specification Languages". Besides surveying the problem, the main purpose of that paper is to provide a very informal introduction to the method of institutions, developed by Goguen and various others (Goguen and Burstall 1985, Goguen and Burstall 1986, Sannella and Tarlecki 1987). It is Goguen's thesis that the main difference between very many of these specification languages is just the logical system in which they are operating. Goguen and Burstall (1985) describes "the notion of an institution as a precise generalisation of the informal notion of logical system". This includes systems of equational logic (as used in Universal Algebra), Horn clause logic (as used in Pure Prolog), first order logic with or without equality, and various restrictions thereof, as well as systems which admit partial operations or order-sorted signatures.

4.1 Introduction

It is my hope that any reader will be able to pick up from what follows an idea of the "shape" of an institution, and that a reader familiar with just the basic idea of Universal Algebra (see Section 1.3) should see how institutions have been arrived at by generalising from that point. From Goguen and Burstall (1986):

> Intuitively, an *institution* is a formalisation of the notion of "logical system" having the following:
>
> **signatures** which generally provide vocabularies for sentences
>
> **Σ-sentences** for each signature Σ
>
> **Σ-models** for each signature Σ,
>
> a **Σ-satisfaction** relation, of Σ-sentences by Σ-models, and
>
> **signature morphisms** which describe changes of notation, with corresponding transformations for sentences and models.

(The map from signature morphisms to transformations on models is called the *reduct functor.*) Although the above lists all the constituent parts of an institution, in order to be called an institution, these parts must obey the *satisfaction condition*, which requires that satisfaction is preserved under change of notation (signature morphisms). Without worrying about the details of the morphisms, it should be clear, from Section 1.4, how to construe the two "settings" of Section 1.3 (the equational one of (1.1) and the quantificational one of (1.2)) as institutions. In each case, signatures consist of function symbols, indexed by a (numeric) arity, and models must interpret these by actual functions of appropriate arity. Terms are built up from function symbols and variables as in Section 1.4, and equality is the only predicate. The quantificational institution is just the first order logic built up from these atomic formulae, i.e. from equations. In the purely equational institution, the equations themselves are (the only) sentences, and a model M satisfies an equation $t = t'$ just in case $[t] = [t']$ no matter what values $[x] \in [M]$

are assigned to the variables $x \in \mathcal{X}$. When a model M satisfies a sentence s, we write $M \models s$.

Historically, the generalisation from Universal Algebra to institutions was preceded by that to *multi-sorted* algebras. These may involve, not just one, but any number of set names (more properly *sort* names). Operators must then be indexed by the names which are to correspond to the sets for each argument, and to that for the result. This is, of course, merely a more complex institution. The preceding work was based on an institution which also allowed partial operations and predicates.

4.1.1 Using different institutions

The paper Goguen and Burstall (1985) presents some mathematics which might conceivably be used to allow one to work in more than one institution (multiplex institutions). However the mathematics is quite complicated, and is limited (as ASL is not) to putting together classes of models which correspond to *theories* (i.e. the maximal classes of models satisfying some set of sentences of the institution). Sannella and Tarlecki (1985) describes how the language ASL may be further generalised by leaving the underlying institution as a parameter, and defining specification-building operations as far as possible in terms of an arbitrary institution. However in order to write an actual specification, one must use a specific institution. Sannella and Tarlecki (1985) might be thought of as a recipe to simplify the production of a specification language for any institution one cares to define. This theory is recapitulated in Sannella and Tarlecki (1988) in order to introduce a specification-building operation which allows specifications written in different institutions to be tacked together, by the use of *semi-institution morphisms*. (These are simpler than the *institution morphisms* of Goguen and Burstall (1985).) A semi-institution morphism is a mapping between institutions that maps signatures to signatures, signature morphisms to signature morphisms, and, in a fashion consistent with the signature map, models to models (in category theory terms, a *natural transformation* on models). A semi-institution morphism

$\gamma : I \to I'$ can be used to define a new specification-building operation, **change institution of** SP **via** γ (where SP is a Σ-specification in I), which specifies those models in the institution I' which may be formed by mapping a model of the I-specification SP into I' using γ. The details of these definitions will be given later, in Section 4.2.2.

4.1.2 Defining institutions

Churning through all the definitions and conditions needed to demonstrate institutionality can be a tiresome and difficult task. Goguen and Burstall (1986) give some theory designed to make this process easier, but it is conceptually quite difficult, and probably more general than is usually required. Tarlecki (1984) presents a class of *abstract algebraic institutions*, which includes most familiar algebraic variants, but this is aimed at generalising some results about term algebras, and certainly does not make it any easier to produce a definition. A common trick is to start with a very general institution, and define others as restrictions of it. For instance, Goguen and Burstall (1985) demonstrates institutionality for (multi-sorted) (total) first order predicate logic (**TP**, say), and then defines the institution of (multi-sorted) first order predicate logic with equality (**TPQ**) by restricting to signatures containing equality predicates for each sort, signature morphisms which preserve these, and models which correctly interpret them. Since **TP** reducts preserve this property of models, the reduct functor can remain unchanged and the satisfaction condition follows from that of **TP**. If reducts did not preserve the interpretation of equality we would not have a true restriction, and the satisfaction condition would not be met. A natural question to ask is why this cannot be achieved by writing a specification in **TP**. The answer is that the restriction cannot be expressed by sentences of the institution, that is to say, equality is not a first order property. Another example from the same paper shows that the Horn clause fragment of **TPQ** is an institution, because this class of sentences (Horn clauses) is closed under translation by signature morphism.

In the main body of the text I will show, as an example, how to construe

pure, single-sorted predicate logic as an institution, and give some examples of institutions-by-restriction. The idea is that the proofs for most algebraic-style institutions will be much the same, and other proofs can be described in terms of this one. The mass of detail is unfortunately irksome, but should not need to be suffered too often. In Section 1.8 I described the operations of ASL in terms of the institution **Q**, of (multi-sorted) equational logic, assuming a certain encoding for predicates, in order to simplify exposition, by keeping down the size of signatures. Otherwise, though, it would be better to suppose that the "standard" institution, in which all specifications outwith this chapter are written, is **PQ**, the institution of multi-sorted partial predicate logic with equality.

4.2 Institutions and Implementation

In algebraic specification, programs are identified with models. For any programming language, then, if we have a (complete) specification of a program, we must have some way of associating that specification with a single model. One way to go about this is to apply stepwise refinement of a loose specification until it only has one model. But suppose we have made refinements which identify some subtask whose implementation in a particular programming language is apparent, even though the corresponding subspecification could still have many models. The appropriate refinement at this point is to replace that subspecification by the implementation. For instance, if the programming language in question, was, say, Prolog, then the implementation looks just like a specification in (single-sorted) predicate logic anyway, but that specification will probably have many (non-isomorphic) models. In the case of Prolog, we can transform any program into a specification of the intended model in single-sorted predicate logic just by prefixing it with "**extend** $\emptyset$ **by**", but even then, we may need to show how we can use this single-sort specification in refining a multi-sorted specification. One solution is to treat multi-sorted and single-sort logics as different institutions, and use a semi-institution morphism from the latter to the former with the **change**

institution operation. To begin with then, here is the institution of pure, single-sorted predicate logic.

4.2.1 The institution of pure predicate logic

The description in Section 1.4 of single-sort pure predicate logic does most of the work associated with defining it as an institution (say **SP**), describing signatures, models, sentences, and satisfaction. I will go over this again below (the presentation is slightly different, but equivalent). In addition we must characterise change of notation, by giving a definition of signature morphism, and show that satisfaction is preserved under change of notation.

We fix a collection of variables $\mathcal{X}$. A signature Σ is a set of predicate names, indexed by arity; so for instance Σ_3 is the set of three place predicates. Models M of Σ must supply

1. a carrier set (written $[M]$),

2. for every predicate name $p \in \Sigma_i$, an interpretation $[Mp]$ consisting of a set of i-tuples over the carrier $[M]$.

Signature morphisms $\sigma : \Sigma \to \Sigma'$ map predicate names $p_i \in \Sigma_i$ to predicate names $\sigma p_i \in \Sigma'_i$. Given a signature morphism $\sigma : \Sigma \to \Sigma'$, any Σ'-model M' can be interpreted as a Σ-model $\sigma M'$ (or $M'|_\sigma$), the *σ-reduct* of M', by defining

1. $[\sigma M'] = [M']$, and

2. $[(\sigma M')p] = [M'(\sigma p)]$ for every predicate $p \in \Sigma$.

For any signature Σ, the *Σ-formulae of rank 0* have the form $p(x_1, \ldots x_i)$, where $p \in \Sigma_i$ and $x_1, \ldots x_i \in \mathcal{X}$. A signature morphism $\sigma : \Sigma \to \Sigma'$ can be extended to formulae of rank 0 by setting $\sigma(p(x_1, \ldots x_i)) = (\sigma p)(x_1, \ldots x_i)$. For any $n > 0$, the Σ-formulae of rank n are all formulae having the forms

1. $\neg\phi$, where ϕ is a Σ-formula of rank $n - 1$

2. $\phi \wedge \psi$, where ϕ and ψ are Σ-formulae of rank less than n, and one of them is of rank $n-1$ exactly

3. $\forall x \,.\, \phi$, where ϕ is a Σ-formula of rank $n-1$ and $x \in \mathcal{X}$

A signature morphism $\sigma : \Sigma \to \Sigma'$ can be extended from rank $n-1$ to rank n by setting

1. $\sigma(\neg\phi) = \neg(\sigma\phi)$

2. $\sigma(\phi \wedge \psi) = (\sigma\phi) \wedge (\sigma\psi)$

3. $\sigma(\forall x.\phi) = \forall x.(\sigma\phi)$

Σ-sentences are formulae which are *closed* with respect to every variable. A formula is closed with respect to a variable x if it has the form

1. $p(x_1, \ldots x_i)$, where $p \in \Sigma_i$, and $x \neq x_j \in \mathcal{X}$, for $1 \leq j \leq i$.

2. $\phi \wedge \psi$, where both ϕ and ψ are closed with respect to x,

3. $\neg\phi$, where ϕ is closed with respect to x, or

4. $\forall y \,.\, \phi$, where $y \in \mathcal{X}$ and either $y = x$ or ϕ is closed with respect to x,

A *variable assignment* ν for a Σ-model M is a map $\nu : \mathcal{X} \to [M]$. We define *satisfaction* of a Σ-formula by a Σ-model M *with respect to* a variable assignment $\nu : \mathcal{X} \to [M]$ (write $M \models^\nu \phi$) as follows.

1. $M \models^\nu p(x_1, \ldots x_i)$, where $p \in \Sigma_i$, just in case $\langle \nu(x_1), \ldots \nu(x_i) \rangle \in [Mp]$.

2. $M \models^\nu \neg\phi$, where ϕ is a Σ-formula of rank $n-1$, just in case $M \not\models^\nu \phi$.

3. $M \models^\nu \phi \wedge \psi$, where one of ϕ and ψ is of rank $n-1$ and the other is of rank at most $n-1$, just in case $M \models^\nu \phi$ and $M \models^\nu \phi$.

4. $M \models^{\nu} \forall x \,.\, \phi$, where ϕ is a Σ-formula of rank $n-1$, just in case $M \models^{\nu'} \phi$ for every assignment $\nu' : \mathcal{X} \to [M]$ such that $x \neq y \in \mathcal{X}$ implies $\nu'(y) = \nu(y)$.

Then say a Σ-model M *satisfies* a Σ-sentence ϕ (write $M \models \phi$) just in case $M \models^{\nu} \phi$ for *every* variable assignment $\nu : \mathcal{X} \to [M]$.

That completes definition of the constituent parts of this logic. Now in order to show that they constitute an institution, we must prove the satisfaction condition: that if ϕ is a Σ-sentence, $\sigma : \Sigma \to \Sigma'$ is a signature morphism, and M' is a Σ'-model, $\sigma M'$ satisfies ϕ iff (if and only if) M' satisfies $\sigma\phi$. This shows that satisfaction is invariant under change of notation. We prove the stronger condition, that for any signature morphism $\sigma : \Sigma \to \Sigma'$, Σ-formula ϕ, Σ'-model M', and variable assignment $\nu : \mathcal{X} \to [M']$, $\sigma M' \models^{\nu} \phi$ if and only if $M' \models^{\nu} \sigma\phi$. The proof, by induction on rank, follows.

If ϕ is a Σ-formula of rank 0, $\sigma : \Sigma \to \Sigma'$ is a signature morphism, M' is a Σ'-model, and $\nu : \mathcal{X} \to [M']$ is a variable assignment, then clearly $\sigma M' \models^{\nu} \phi$ iff $M' \models^{\nu} \sigma\phi$. Now make the inductive assumption that for any Σ-formula ϕ of rank less than n (some $n > 0$), signature morphism $\sigma : \Sigma \to \Sigma'$, Σ'-model M' and variable assignment $\nu : \mathcal{X} \to [M']$, $M' \models^{\nu} \sigma\phi$ iff $\sigma M' \models^{\nu} \phi$. Then

1. $M' \models^{\nu} \sigma(\neg\phi)$
   iff $M' \models^{\nu} \neg(\sigma\phi)$
   iff $M' \not\models^{\nu} \sigma\phi$
   iff $\sigma M' \not\models^{\nu} \phi$
   iff $\sigma M' \models^{\nu} \neg\phi$

2. $M' \models^{\nu} \sigma(\phi \wedge \psi)$
   iff $M' \models^{\nu} (\sigma\phi) \wedge (\sigma\psi)$
   iff $M' \models^{\nu} \sigma\phi$ and $M' \models^{\nu} \sigma\psi$
   iff $\sigma M' \models^{\nu} \phi$ and $\sigma M' \models^{\nu} \psi$
   iff $\sigma M' \models^{\nu} \phi \wedge \psi$

3. $M' \models^{\nu} \sigma(\forall x \,.\, \phi)$

$$
\begin{aligned}
&\text{iff } M' \models^{\nu} \forall x \,.\, (\sigma\phi) \\
&\text{iff } M' \models^{\nu'} \sigma\phi \text{ for every assignment } \nu' : \mathcal{X} \to [M'] \\
&\qquad \text{such that } x \neq y \in \mathcal{X} \text{ implies } \nu'(y) = \nu(y) \\
&\text{iff } M' \models^{\nu'} \sigma\phi \text{ for every assignment } \nu' : \mathcal{X} \to [\sigma M'] \\
&\qquad \text{such that } x \neq y \in \mathcal{X} \text{ implies } \nu'(y) = \nu(y) \\
&\text{iff } \sigma M' \models^{\nu'} \phi \text{ for every assignment } \nu' : \mathcal{X} \to [\sigma M'] \\
&\qquad \text{such that } x \neq y \in \mathcal{X} \text{ implies } \nu'(y) = \nu(y) \\
&\text{iff } \sigma M' \models^{\nu} \forall x \,.\, \phi
\end{aligned}
$$

Thus by induction, $M' \models^{\nu} \sigma\phi$ iff $\sigma M' \models^{\nu'} \phi$, for ϕ of any rank $n \geq 0$. The satisfaction condition follows immediately. Thus we have an institution, **SP**.[1] We can obtain metasyntactic definitions of $\vee, \to, \leftrightarrow$ and $\exists$.

4.2.2 Semi-institution morphisms and change institution

While we can simulate functions in the preceding institution by using predicates and adding suitable axioms, some of these axioms will be existentially quantified, and therefore not suitable for use in specifying an initial algebra. Even if we restrict to conditional form, the only initial algebra is the trivial one, and no objects are ever reachable (since they can only be named by a variable). Thus in order to produce an institution in which to model Prolog programs, we ought to modify the institution of Subsection 4.2.1 by adding operation names directly to signatures, and adding their interpretation as total functions to the model (call this institution **STP**). This is tedious but not difficult. We then get the familiar ideas of reachability and initiality from Section 1.5. So now the Prolog program

```
plus(0,X,X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
```

---

[1] In order to deal properly with the **extend** operation, the definition of an institution must be augmented with a definition of *model morphisms*: see Section 4.4.

can be written in **STP** as

$$\begin{array}{lll} \mathsf{PLUS} & = & \textbf{extend } \emptyset \\ & & \textbf{by } \forall x \,.\, \mathtt{plus}(0, x, x) \\ & & \quad \forall xyz \,.\, \mathtt{plus}(x, y, z) \rightarrow \mathtt{plus}(\mathtt{s}(x), y, \mathtt{s}(z)) \end{array}$$

In order to incorporate this into a larger specification in a multi-sorted institution, we can use a semi-institution morphism, and the **change institution** operation. These should first be spelt out in more detail.

A semi-institution morphism is a mapping between institutions, $\gamma : I \rightarrow I'$, which maps signatures Σ of I to signatures $\gamma(\Sigma)$ of I', signature morphisms $\sigma : \Sigma \rightarrow \Sigma'$ of I to signature morphisms $\gamma(\sigma) : \gamma(\Sigma) \rightarrow \gamma(\Sigma')$ of I', and Σ-models M of I to $\gamma(\Sigma)$-models $\gamma(M)$ of I'. Furthermore, this map must respect the reduct functor, in that if $\sigma : \Sigma \rightarrow \Sigma'$ is a signature morphism in I and M' is a Σ'-model, then $\gamma(\sigma M') = (\gamma\sigma)(\gamma M')$. Loosely, a semi-institution morphism maps models of one institution into models of another, in a way which respects changes of notation (signature morphisms). A semi-institution morphism $\gamma : I \rightarrow I'$ can be used to define a new operation, **change institution of** SP **via** γ as follows:

$$\begin{array}{l} \mathrm{Sig}[\textbf{change institution of } SP \textbf{ via } \gamma] = \gamma(\mathrm{Sig}[SP]), \\ \mathrm{Mod}[\textbf{change institution of } SP \textbf{ via } \gamma] = \{\gamma(M) | M \in \mathrm{Mod}[SP]\} \end{array}$$

Thus **change institution of** SP **via** γ describes models in the institution I' which may be formed by mapping a model of the I-specification SP into I' using γ.

4.2.3 Prolog

We require a semi-institution morphism, **single** say, from **STP** into our familiar multi-sorted institution **PQ**. Somehow, sorts will need to be imposed on **STP** models. The simplest solution is to introduce a single sort, `All` say, which names the (previously anonymous) single sort modelled by $[M]$ in any **STP** model M. Then for any single-sorted signature Σ, define

1. $\text{sorts}(\textbf{single}\,\Sigma) = \{\texttt{All}\}$,

2. for any $i \geq 0$, $(\textbf{single}\,\Sigma)_{\texttt{All}^i \to \texttt{All}} = \text{opns}(\Sigma_i)$,

3. for any $i \geq 0$, $(\textbf{single}\,\Sigma)_{\texttt{All}^i} = \text{preds}(\Sigma_i)$,

4. for $\sigma : \Sigma \to \Sigma'$ a signature morphism in **STP**, $\textbf{single}(\sigma) : \textbf{single}(\Sigma) \to \textbf{single}(\Sigma')$ is a signature morphism in **PQ** such that for any $q \in \Sigma$ (whether q names a function or a predicate), $(\textbf{single}\,\sigma)q = \textbf{single}(\sigma q)$, and

5. for M a Σ-model, $[(\textbf{single}\,M)\texttt{All}] = [M]$, and for any $q \in \Sigma$, $[(\textbf{single}\,M)q] = [Mq]$.

If M' is a Σ'-model and $\sigma : \Sigma \to \Sigma'$ is a signature morphism (in **STP**), then $\textbf{single}(\sigma M') = (\textbf{single}\,\sigma)(\textbf{single}\,M')$. This shows that **single** respects change of notation in the required fashion, and is a semi-institution morphism. Thus we could for instance use

derive from

change institution PLUS via single

by $[\texttt{Int} \longmapsto \texttt{All}]$

in a larger specification, in order to implement the integers with addition.

4.2.4 PATR-II

Now consider the case of PATR-II. As already indicated in Section 3.1.3, one way of adding, say, number information to a category would be to use terms like NP(SINGULAR). There is no particular reason to give NP different status to SINGULAR: instead we could employ a single constructor f, say, and terms like f(NP, SINGULAR). But as we add more and more features, and allow the values they take on to themselves have complex structure, we might get terms like

$$\texttt{f}(\texttt{f}(\text{S}, \text{NIL}, \text{BACK}, \texttt{f}(\text{NP}, x, \text{NIL}, \text{NIL})), x, \text{FORW}, \texttt{f}(\text{NP}, y, \text{NIL}, \text{NIL}))$$

and much worse, and it becomes extremely difficult to keep track of which argument is supposed to correspond to which feature, at which level of embedding of a term. However we can make use of this correspondence between features and argument positions in a term to form the basis of a semi-institution morphism, which allows us to use the more perspicuous notation of extractors (i.e. feature names: `cat`,`agr` and so on) without having to explicitly write out equations like $\mathtt{cat}(\mathtt{f}(x_1, x_2, x_3, x_4)) = x_1$, as we did in PSYN (3.6). We begin by defining the domain institution, **Patr**.

Again we can start from single-sorted, total, predicate logic (**STP**). Now we will restrict to signatures containing only atoms Σ_0, plus one other operation name, consisting of a list of length some $i_\Sigma > 0$. The arity of this operation must also be i_Σ. Call the elements of this list $\Sigma(1), \ldots, \Sigma(i_\Sigma)$. For clarity, it will be simpler to assume that the arguments to this operation are infixed, so that the operation name could be written $\langle \Sigma(1) : \_, \ldots, \Sigma(i_\Sigma) : \_ \rangle$. For the purposes of defining the internal workings of this institution, the peculiar syntactic form of this operation name is quite irrelevant, but it will be useful when it comes to defining a semi-institution morphism **patr** : **Patr** $\rightarrow$ **PQ**. Signature morphisms clearly only exist between signatures Σ and Σ' when $i_\Sigma = i_{\Sigma'}$. For any such signature Σ, the set of Σ-sentences is empty (as there are no predicate names), so the satisfaction condition is trivial, and we have an institution. **Patr** inherits from **STP** its notions of reachability and initiality. For example, in the institution **Patr**, models of the specification

PSP = **extend** $\emptyset$ **by**
atoms NP,VP,S,SINGULAR,PLURAL,
1, 2, 3, `john`, `smiles`
opn $\langle$`cat`, `agr`, `num`, `pers`$\rangle$

have objects in one-to-one correspondence to the terms built up in the usual way from the constants NP, VP, S, SINGULAR, PLURAL, 1, 2, 3, `john` and `smiles` using the operation $\langle$`cat`:_, `agr`:_, `num`:_, `pers`:_$\rangle$, e.g.:

$\langle$`cat`:NP, `agr`:$\langle$`cat`:`smiles`, `agr`:2, `num`:SINGULAR, `pers`:3$\rangle$`num`:VP, `pers`:1$\rangle$.

We require a semi-institution morphism **patr** from **Patr** into our familiar multi-sorted setting **PQ**. For any **Patr**-signature Σ, set

1. $\text{sorts}(\mathbf{patr}\,\Sigma) = \{\texttt{Syn}\}$,
2. $(\mathbf{patr}\,\Sigma)_{\rightarrow \texttt{Syn}} = \Sigma_0$,
3. $(\mathbf{patr}\,\Sigma)_{\texttt{Syn}\rightarrow\texttt{Syn}} = \{\Sigma(1), \ldots, \Sigma(i_\Sigma)\}$,

and $\mathbf{patr}(\Sigma)$ otherwise empty. If $\sigma : \Sigma \rightarrow \Sigma'$ is a signature morphism in **Patr**, define $\mathbf{patr}(\sigma)$ to be the **PQ**-signature morphism such that

1. $(\mathbf{patr}\,\sigma)(\texttt{Syn}) = \texttt{Syn}$,
2. $(\mathbf{patr}\,\sigma)(a) = \mathbf{patr}(\sigma a)$ for $a \in \Sigma_0$, and
3. $(\mathbf{patr}\,\sigma)(\Sigma(j)) = \mathbf{patr}(\Sigma'(j))$ for $1 \leq j \leq i_\Sigma$.

If M is a Σ-model in **Patr**, then set

1. $[(\mathbf{patr}M)\texttt{Syn}] = [M]$,
2. $[(\mathbf{patr}M)a] = [Ma]$ for each $a \in \Sigma_0$, and
3. $[(\mathbf{patr}M)(\Sigma(j))](x)$, where $1 \leq j \leq i_\Sigma$ and $x \in [M]$, to be undefined, unless $\chi_j^M(x) = \{x_j | x = [M\langle\Sigma(1) : x_1, \ldots, \Sigma(i_\Sigma) : x_{i_\Sigma}\rangle]$, some $x_1, \ldots x_{i_\Sigma} \in [M]\}$ contains exactly one such point x_j (i.e. $\chi_j^M(x) = \{x_j\}$), when $[(\mathbf{patr}M)(\Sigma(j))](x) = x_j$.

This last clause could do with further explanation. In order to determine the value of $[(\mathbf{patr}M)(\Sigma(j))]$ at some point $x \in [M]$, one must first determine which i_Σ-tuples $\langle x_1, \ldots x_{i_\Sigma}\rangle$ make $[M\langle\Sigma(1) : x_1, \ldots, \Sigma(i_\Sigma) : x_{i_\Sigma}\rangle]$ equal to x. Form the set $\chi_j^M(x)$ by selecting just the value x_j from each such tuple. If this set is a singleton $\{x_j\}$, then $[(\mathbf{patr}M)(\Sigma(j))](x) = x_j$, otherwise it is undefined. Thus if M is initial, then

$$[(\mathbf{patr}M)(\Sigma(j))]([M\langle\Sigma(1) : x_1, \ldots \Sigma(i_\Sigma) : x_{i_\Sigma}\rangle] = x_j,$$

but $[(\mathbf{patr}M)(\Sigma(j))]$ is undefined at $[Ma]$ (any $a \in \Sigma_0$).

If $\sigma : \Sigma \to \Sigma'$ is a signature morphism in **Patr** and M' is a Σ'-model, then $\mathbf{patr}(\sigma M') = (\mathbf{patr}\,\sigma)(\mathbf{patr}\,M')$. This shows that **patr** respects signature morphisms in the required fashion, and so is a semi-institution morphism.

We can use **patr** to translate a specification like PSP into **PQ** and produce a domain of intensions for use in PATR-II models. For instance, in implementing LXN (2.2), the specification

PWDS + **change institution** PSP **via patr**

has the same (mutually isomorphic) models as the specification

derive from PWDS + PAT + PSYN **by** $[\not\mapsto \texttt{Atom}, \texttt{Avs}, \texttt{f}]$

references: PWDS (3.3), PAT (3.4), PSYN (3.6)

All we are doing here is using the term structure to ensure that we get all the required attributes, without having to write out new equations ($\texttt{cat}(\texttt{f}(x_1, x_2, x_3, x_4)) = x_1$ and so on) for every new PATR-II specification. It is much more a programming language than a "real" logic. The **patr**-images of non-initial models are rather uninuitive. Even the initial models differ from more familiar ideas about feature-value models (as in, say, GKPS) in that the order in which the features are listed is important: models of

extend $\emptyset$ **by**
atoms NP,VP,S,SINGULAR,PLURAL,
1, 2, 3, `john`, `smiles`
opn $\langle$`agr`, `num`, `cat`, `pers`$\rangle$

are not the same as those of PSP. No other work can be done in **Patr**, since it has no sentences. For instance, it would have been nice if we could have identified, while still in **Patr**, which objects should go into `Lex`. Having decided that this is the correct style of model, we are unable to apply the discipline of stepwise refinement

within **Patr**. Thus **Patr** cannot be used as the basis of a wide-spectrum language. In the following section, we see a feature-value institution in which the non-initial models do correspond to more familiar ideas of feature-value models (as in, say, GKPS), and which could potentially be used as a wide-spectrum language. It is also an open-arity institution, in that it is possible for one feature to fail to take a value, without all of them doing so.

4.3 An Open-Arity Feature-Value Institution

The strategy used in models M for this institution is to provide some base set $[M]$ which interprets *path expressions* over a signature. If f_1 and f_2 are features, and a an atom, $f_1 : f_2 : a$ is a path expression describing a point at which the value of f_1 is such that the value of f_2 is a. Atoms a are interpreted by points $[Ma]$, and features f by total functions $[Mf]$ on $[M]$, where $[Mf]([Ma])$ will be a point where the feature f takes the value a (the value of the path expression $f:a$). The idea then is to think of objects I as *sets*, drawn from this base — that is, sets of paths — but we need to restrict to a collection $[M\top]$ of *coherent* path sets, which excludes I such that $i, [Ma] \in I$ and $i \neq [Ma]$, or $[Mf](i), [Mf](j) \in I$ such that $i, j \in [M]$ are incoherent (for atoms a and features f). Then for $I \in [M\top]$, i is an element of the set corresponding to the value of the feature f at I if and only if $[Mf](i) \in I$. Call the institution **Open**.

We fix a collection of variables $\mathcal{X}$. A signature Σ consists of *atoms* Σ_0, *features* Σ_1, and *sorts* Σ_2. (In this formulation, the elements of Σ_2 might better be termed predicates, but sorts is the more usual term). Models M of Σ must supply:

(4.1) a set $[M]$, and an irreflexive, symmetric *incoherence* relation $\_{}_M\_{} \subseteq [M] \times [M]$ (write $[M\top]$ for the collection of *coherent* subsets of $[M]$, $[M\top] = \{I \subseteq [M] | \text{no } i, j \in I \text{ satisfies } i \,{}_M\, j\}$),

(4.2) for each $a \in \Sigma_0$, an object $[Ma] \in [M]$ such that $[M] \ni i \neq [Ma]$ implies $i \,{}_M\, [Ma]$,

(4.3) for every $f \in \Sigma_1$, a map $[Mf] : [M] \to [M]$ such that $i_M j$ implies $[Mf](i)_M[Mf](j)$, and

(4.4) for every $s \in \Sigma_2$, a collection $[Ms] \subseteq [M\top]$

Signature morphisms $\sigma : \Sigma \to \Sigma'$ map atoms to atoms, features to features, and sorts to sorts. Any Σ'-model M' can be interpreted as a Σ-model $\sigma M'$ (or $M'|_\sigma$), the *σ-reduct* of M', by defining

1. $[\sigma M'] = [M']$, and $\_{}_{\sigma M'}\_{} = \_{}_{M'}\_{}$ (so also $[(\sigma M')\top] = [M'\top]$),
2. $[(\sigma M')a] = [M'(\sigma a)]$ for every $a \in \Sigma_0$ (this satisfies the condition in 4.2 above),
3. $[(\sigma M')f] = [M'(\sigma f)]$ for every $f \in \Sigma_1$ (this satisfies the condition in 4.3 above),
4. $[(\sigma M')s] = [M'(\sigma s)]$ for every $s \in \Sigma_2$.

For any signature Σ, the *Σ-sentences of rank 0* are

1. $\top$ ("top"),
2. x, for $x \in \mathcal{X}$,
3. a, for $a \in \Sigma_0$, and
4. s, for $s \in \Sigma_2$.

For any $r \geq 0$, the *Σ-sentences of rank $r+1$* are either

1. feature terms $f{:}\phi$, where $f \in \Sigma_1$ and ϕ is a Σ-sentence of rank r,
2. conjunctions $\phi \wedge \psi$, where ϕ and ψ are Σ-sentences, one of rank r, the other of rank at most r, or
3. negations $\neg\phi$, where ϕ is a Σ-sentence of rank r.

Assume metasyntactic definitions for $\vee$, $\rightarrow$ and $\leftrightarrow$. A signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ can be extended to map Σ-sentences to Σ'-sentences by setting

1. $\sigma(\top) = \top$,
2. $\sigma(x) = x$, each $x \in \mathcal{X}$,
3. $\sigma(f{:}\phi) = (\sigma f){:}(\sigma\phi)$, each Σ-sentence ϕ,
4. $\sigma(\phi \wedge \psi) = (\sigma\phi) \wedge (\sigma\psi)$, all Σ-sentences ϕ and ψ, and
5. $\sigma(\neg\phi) = \neg(\sigma\phi)$, all Σ-sentences ϕ.

For any Σ-model M set

1. $[[M\top]] = \{\langle I, \nu\rangle | I \in [M\top], \nu : \mathcal{X} \rightarrow [M\top]\}$,
2. for $x \in \mathcal{X}$, $[[Mx]] = \{\langle \nu(x), \nu\rangle | \nu : \mathcal{X} \rightarrow [M\top]\}$,
3. for $a \in \Sigma_0$, $[[Ma]] = \{\langle \{[Ma]\}, \nu\rangle | \nu : \mathcal{X} \rightarrow [M\top]\}$,
4. for $s \in \Sigma_2$, $[[Ms]] = \{\langle I, \nu\rangle | I \in [Ms],\ \nu : \mathcal{X} \rightarrow [M\top]\}$,
5. for Σ-sentence ϕ and $f \in \Sigma_1$,
   $[[M(f{:}\phi)]] = \{\langle I, \nu\rangle | I \in [M\top], \langle \{i | [Mf](i) \in I\}, \nu\rangle \in [[M\phi]]\}$,
6. for any Σ-sentences ϕ and ψ, $[[M(\phi \wedge \psi)]] = [[M\phi]] \cap [[M\psi]]$, and
7. for ϕ a Σ-sentence,
   $[[M(\neg\phi)]] = \{\langle I, \nu\rangle | I \in [M\top],\ \nu : \mathcal{X} \rightarrow [M\top],\ \langle i, \nu\rangle \notin [[M\phi]]\}$.

Then say $M \models \phi$ (for any Σ-sentence ϕ) if and only if for every $I \in [M\top]$ there exists a $\nu_I : \mathcal{X} \rightarrow [M\top]$ such that $\langle I, \nu_I\rangle \in [[M\phi]]$.

In order to show that these definitions constitute an institution, we need to prove the satisfaction condition: that for any signatures Σ, Σ', signature morphism $\sigma : \Sigma \rightarrow \Sigma'$, Σ'-model M' and Σ-sentence ϕ, $M' \models \sigma\phi$ iff $\sigma M' \models \phi$. We prove the stronger condition, $[[M'(\sigma\phi)]] = [[(\sigma M')\phi]]$.

1. $\langle I, \nu \rangle \in [[M'(\sigma\top)]]$
   iff $\langle I, \nu \rangle \in [[M'\top]]$
   iff $I \in [M'\top]$ and $\nu : \mathcal{X} \to [M'\top]$
   iff $I \in [(\sigma M')\top]$ and $\nu : \mathcal{X} \to [(\sigma M')\top]$
   iff $\langle I, \nu \rangle \in [[(\sigma M')\top]]$

2. For any $x \in \mathcal{X}$, $\langle I, \nu \rangle \in [[M'(\sigma x)]]$
   iff $\langle I, \nu \rangle \in [[M'x]]$
   iff $I \in [M'\top]$, $\nu : \mathcal{X} \to [M'\top]$ and $\nu(x) = I$
   iff $I \in [(\sigma M')\top]$, $\nu : \mathcal{X} \to [(\sigma M')\top]$ and $\nu(x) = I$
   iff $\langle I, \nu \rangle \in [[(\sigma M')x]]$

3. For $a \in \Sigma_0$, $\langle I, \nu \rangle \in [[M'(\sigma a)]]$
   iff $I = \{[M'(\sigma a)]\}$ and $\nu : \mathcal{X} \to [M'\top]$
   iff $I = \{[(\sigma M')a]\}$ and $\nu : \mathcal{X} \to [(\sigma M')\top]$
   iff $\langle I, \nu \rangle \in [[(\sigma M')a]]$

4. For $s \in \Sigma_2$, $\langle I, \nu \rangle \in [[M'(\sigma s)]]$
   iff $I \in [M'(\sigma s)]$ and $\nu : \mathcal{X} \to [M'\top]$
   iff $I \in [(\sigma M')s]$ and $\nu : \mathcal{X} \to [(\sigma M')\top]$
   iff $\langle I, \nu \rangle \in [[(\sigma M')s]]$

Thus the condition is satisfied for Σ-sentences of rank 0. Now make the inductive assumption that the condition holds for all Σ-sentences of rank less than or equal to r, some $r \geq 0$. Then

1. For any Σ-sentence ϕ of rank r and $f \in \Sigma_1$, $\langle I, \nu \rangle \in [[M'(\sigma(f{:}\phi))]]$
   iff $\langle I, \nu \rangle \in [[M'((\sigma f){:}(\sigma\phi))]]$
   iff $I \in [M'\top], \langle \{i | [M'(\sigma f)](i) \in I\}, \nu \rangle \in [[M'(\sigma\phi)]]$
   iff $I \in [(\sigma M')\top], \langle \{i | [(\sigma M')f](i) \in I\}, \nu \rangle \in [[(\sigma M')\phi]]$
   iff $\langle I, \nu \rangle \in [[(\sigma M')(f{:}\phi)]]$

2. For any Σ-sentences ϕ and ψ, such that one is of rank r
   and the other is of rank at most r, $\langle I, \nu \rangle \in [[M'(\sigma(\phi \wedge \psi))]]$

iff $\langle I, \nu \rangle \in [[M'((\sigma\phi) \wedge (\sigma\psi))]]$
iff $\langle I, \nu \rangle \in [[M'(\sigma\phi)]]$ and $\langle I, \nu \rangle \in [[M'(\sigma\psi)]]$
iff $\langle I, \nu \rangle \in [[(\sigma M')\phi]]$ and $\langle I, \nu \rangle \in [[(\sigma M')\psi]]$
iff $\langle I, \nu \rangle \in [[(\sigma M')(\phi \wedge \psi)]]$

3. For any Σ-sentence ϕ of rank r, $\langle I, \nu \rangle \in [[M'(\sigma(\neg\phi))]]$
iff $\langle I, \nu \rangle \in [[M'(\neg(\sigma\phi))]]$
iff $I \in [M'\top]$, $\nu : \mathcal{X} \to [M'\top]$ and $\langle I, \nu \rangle \notin [[M'(\sigma\phi)]]$
iff $I \in [(\sigma M')\top]$, $\nu : \mathcal{X} \to [(\sigma M')\top]$ and $\langle I, \nu \rangle \notin [[(\sigma M')\phi]]$
iff $\langle I, \nu \rangle \in [[(\sigma M')(\neg\phi)]]$

Thus, by induction, the condition is proven.

4.4 Model Morphisms and Initiality

The main difficulty in formulating institutions such as **Open** (besides guaranteeing the satisfaction condition), is in getting the right notion of initiality. The definition of initiality independent of institution rests on the precise definition of *model morphism* employed. So far we have neglected to consider model morphisms. To deal with them properly, we should modify the definition of an institution to identify, in categorical terms, certain minimal requirements they should meet. I do not intend to consider them in such detail, but in any case most algebraic morphisms have a similar structure. In essence, they are structure-preserving maps, between models of the same signature (Σ say). For instance, if $h : M_1 \to M_2$ is a monoid morphism, $h([M_1\_ \cdot \_](x, y)) = [M_2\_ \cdot \_](h(x), h(y))$ for every x, y in the carrier of the monoid. In sketch, if we consider all the denotations supplied by a model M as sets (so, for example, if f is partial function name, the denotation $[Mf]$ is a set of pairs $\langle x, y \rangle$ such that $\langle x, y \rangle, \langle x, z \rangle \in [Mf]$ implies $y = z$), then a model morphism $h : M_1 \to M_2$ is a map from $[M_1]$ to $[M_2]$ such that for every $q \in \Sigma$, $h[M_1 q] \subseteq [M_2 q]$. This works even when q is a sort name. This implies that the *h-image* of M_1, formed by $[hM_1] = h[M_1]$ and $[(hM_1)q] = h[M_1 q]$ for

all $q \in \Sigma$, is a Σ-submodel of M_2. Algebraic morphisms are designed to preserve the satisfaction of certain Σ-sentences or formulae. The sort of definition above would mean that if M_1 and M_2 are augmented by variable valuations such that for any $x \in \mathcal{X}$, $[M_2 x] = h([M_1 x])$, then always $M_1 \models \phi$ and $M_2 \models \phi$ implies that the Σ-submodel $h(M_1)$ of M_2 also satisfies ϕ, for any Σ-formula ϕ of rank 0. This preservation of satisfaction then extends to a larger class of all *positive* Σ-formulae. By varying the precise definition of morphism (across different institutions), we can vary the class of Σ-formulae whose satisfaction is preserved in the submodel.

Such a definition should make the collection of Σ-models which satisfy any collection E of positive Σ-sentences into a *category* (see Pierce (1990) for an introduction to basic category theory). A model I is called *initial* in such a category if, for each model M in the category, there exists a unique model morphism from I to M. Thus any positive Σ-formula satisfied in initial I must be satisfied in *every* M. If there is an initial model, it must be unique, up to isomorphism. The existence of an initial model can be guaranteed, in algebraic institutions, by limiting E to a class of *conditional* sentences (Tarlecki 1984).

The choice of $\_{M}\_$ (rather than $[M\top]$) as basic in **Open** was made precisely so that the effect of the usual definition of model morphism would be to put into $\_{M}\_$, (for M initial) only those objects which *must* be there, so that $[M\top]$ will then contain everything that *could* be there (so initial models are *maximally coherent*). One class of sentences for which initial models will be defined consists, for any signature Σ, of all Σ-sentences of the form $\phi_1 \wedge \ldots \wedge \phi_n \rightarrow s$, where $s \in \Sigma_2$, $n \geq 0$, and each ϕ_i either involves no sorts, or, if it does, is of the form $f_1 : f_2 : \ldots : f_j : s'$, for some $s' \in \Sigma_2, f_1, f_2, \ldots f_j \in \Sigma_1, j \geq 0$.[2] Say that an object (coherent path set) $I \in [M\top]$ is reachable in M just in case it is uniquely *described* by some *ground*

---

[2]Consider the initial model M_Σ of a signature alone (i.e. with no axioms added). Since no axiom of the above conditional form forces elements of the base $[M_\Sigma]$ to become equated, the base is the same in any initial model over such axioms. Since the truth of any sentence is defined relative to $[\top]$, no such axiom can force any new pairs into the incoherence relation, so incoherence too is invariant in initial models over such axioms.

sentence ϕ_I : i.e. for some ϕ_I containing no variables and some $\nu : \mathcal{X} \to [M]$, $[[M\phi_I]] = \{\langle I, \nu\rangle\}$. If every $I \in [M\top]$ is reachable, we say that M is reachable. Then a model is initial just when it is reachable and only interprets different terms (terms and sentences are conflated here) by the same object when every model does likewise.

We can define a semi-institution morphism **open** into our more familiar first order institution **PQ** as follows. Set

1. $\text{sorts}(\mathbf{open}\,\Sigma) = \{\texttt{All}\} \cup \Sigma_2$,

2. $(\mathbf{open}\,\Sigma)_{\to\texttt{All}} = \Sigma_0$,

3. $(\mathbf{open}\,\Sigma)_{\texttt{All}\dot{\to}\texttt{All}} = \Sigma_1$

and $\mathbf{open}(\Sigma)$ otherwise empty. If $\sigma : \Sigma \to \Sigma'$ is a signature morphism in the domain institution of **open**, then for any $q \in \Sigma_i, 0 \leq i \leq 3$, set $(\mathbf{open}(\sigma))(q) = \mathbf{open}(\sigma q)$. Then $\mathbf{open}(\sigma)$ is a signature morphism from $\mathbf{open}(\Sigma)$ to $\mathbf{open}(\Sigma')$. If M' is a Σ'-model in the domain institution, then set

1. $[(\mathbf{open}M')\texttt{All}] = [M'\top]$,

2. $[(\mathbf{open}M')a] = \{[M'a]\}$ for each $a \in \Sigma'_0$,

3. $[(\mathbf{open}M')s] = \{I \in [M'\top] | \langle I, \nu\rangle \in [[M's]],\ \text{some } \nu\}$

4. $[(\mathbf{open}M')f](I) = \{i | [M'f](i) \in I\}$.

If also $\sigma : \Sigma \to \Sigma'$ is a signature morphism in the domain institution, $\mathbf{open}(\sigma M') = (\mathbf{open}\,\sigma)(\mathbf{open}M')$. This shows that **open** respects signature morphisms in the required fashion, and so is an institution morphism. We could, for instance, in this institution easily write a specification of the feature system for GPSG (Section 3.5)

Thus in any such model, the truth of any sentence not involving sorts is determined by the signature alone.

which incorporates FCRs from the beginning, without requiring the explicit supersort Syn′:

OFCR = **extend** $\emptyset$ **by**
atoms N,V,0,1...
feats maj,bar,...
sorts Maj,Bar,Syn,Fcr1,Fcr2...
axioms N $\rightarrow$ Maj, V $\rightarrow$ Maj...
axioms 0 $\rightarrow$ Bar, 1 $\rightarrow$ Bar...
$\vdots$
axiom $\neg$agr:$\top$ $\rightarrow$ Fcr12
axiom maj:V $\rightarrow$ Fcr12
$\vdots$
axiom maj:Maj $\wedge$ bar:Bar $\wedge$ ...
$\wedge$ Fcr1 $\wedge$ Fcr2 $\wedge$ ... $\rightarrow$ Syn

Then **open** can be employed to "translate" OFCR into our familiar institution, where we could proceed to bolt on ID and LP rules:

extend
change institution OFCR **via open**
by pred _ $\lhd$ _ $\subseteq$ Syn $\times$ MSyn
$\vdots$

(Alternatively, we could change the feature-value institution to have predicates of arity greater than one, and specify $\lhd$ in that institution as well.) In **Open**, unlike **Patr**, we do get classes of models we can work with, with objects consisting of coherent sets of paths. This allows us also to write more abstract specifications in **Open**. For instance we can express the above FCR more abstractly by the specification

OF12 = **sort** Syn
atom V

feats maj,agr

axiom Syn ∧ agr:⊤ → maj:V

and then note that OFCR is an implementation: OFCR ⊏ OF12.

4.5 Coda

In this chapter we have seen how different forms of logic may be construed as different institutions. Different institutions carry different notions of signature morphism, reachability and free extension. A specification which is awkward or impossible in one institution may be completely natural in another. Notably, the free extension construct can be viewed as an (abstract) programming language, so the notion of institution can be used to capture the use of different programming languages. The use of semi-institution morphisms allows us to use the mechanics of a model in one institution to define a similar model in quite a different institution. This can be used in fitting a module written in one programming language to the parameter of a parameterised module written in another, or in formally showing a program in a specialised logic to be an implementation of a specification in, say, full first order logic.

Chapter 5

Unbounded Dependencies

There are many styles and notations of grammar. In general, they all do quite well with a core of simple declarative sentences. The constructs of the grammar which deal with such sentences will usually be relatively simple and often constitute an easily identifiable sub-formalism. Typically, while some more complex phenomena may be handled with little extra effort, others require more difficult extensions to the basic grammar. In many cases, one feels that the spirit of these extensions does not vary markedly from grammar to grammar, and that notational differences spring just from the core mechanism being extended. Modularity (and parameterisation) may be helpful here, both at the level of description (a parameterised specification which, given a class of models covering at least the core phenomena, describes a class which also covers the additional phenomena) and at the level of construction (another such specification, with the additional property that, given a single model, it describes a single model, i.e. that it is a parameterised program). As an example, let us consider the problem of so-called *unbounded dependencies.* These include topicalisations and relative clauses.

5.1 Modularity, Parameterisation, Abstraction

Previously (in Section 3.2) I introduced partial equivalence relations as a way of talking about the sorts of effects feature-value systems aim at, abstracting away from the various techniques of implementation they employ. However there may be cases in which even the level of specificity of equivalence relations is unnecessary. For instance, below we will see a description of topicalisation which assumes the concepts of sentential intension, and maximal projection, but does not particularly rely on there being equivalence classes $=_{\texttt{cat}}$ and $=_{\texttt{bar}}$. Now we can write grammars which do *not* employ such equivalences (for instance, we might allow an intension to bear more than one possible category). Then, provided we can nevertheless identify sentential intensions and maximal projections, the description of topicalisation ought still to apply. The most general way to say that it is necessary that sentential intensions and maximal projections be identifiable, is to presuppose sort or predicate names which pick them out. Then in order to demonstrate that a particular account refines this description, we should have to show how it identifies sentential intensions and maximal projections, by defining the inhabitants of the corresponding sorts or predicates in terms of the particular syntax of the account in question, and also to show that the general topicalisation description which is couched in terms of those sorts or predicates then applies to the particular account.

For instance, suppose that our abstract specification of topicalisation employs sort names `S` and `XP` to refer to sentential intensions and maximal projections, but that the particular account (say in GPSG) which we wish to show implements this abstraction instead employs features encoded as partial equivalences $\_ =_{\texttt{cat}} \_$ and $\_ =_{\texttt{bar}} \_$, and identifies sentences by their being `cat`-equal to some known sentential intension, S say, and maximal projections by their being `bar`-equal to some known maximal projection, NP say. Then we would need to show how this account can furnish us with definitions of the sorts `S` and `XP`, to wit: $s : \texttt{S} \leftrightarrow s =_{\texttt{cat}} \text{S}$, and $r : \texttt{XP} \leftrightarrow r =_{\texttt{bar}} \text{NP}$. Then we would need to show that, this interface fitting

complete, the particular account satisfies the requirements of the abstract one (expressed in terms of S and XP).

So the general description of topicalisation is intended to codify what is relatively certain ground. The work of showing that actual accounts, which linguists work with now, refine this description, is a demonstration that it is a good abstraction of what it *means* to be an account of topicalisation. Of course working accounts are not necessarily perfect anyway, and when we discover that an account does not refine our description, we must try to determine whether the deficiency is in the particular account, or in our abstraction. Such an abstraction, once established, should make it easier to describe the particular details of construction used in adding topicalisation, to, say, GPSG, in terms minimally reliant on the *other* details of GPSG. As an extreme example, it may even be possible to produce a constructive account of topicalisation which assumes no more than does the general description thereof (though we should not, in general, expect a task division apposite for the purposes of specification to be equally good for implementation). For such a parameterised construction, one should prove that, given any appropriate model to stand in place of the parameter, that the constructed model refines the general description. Then such an account of topicalisation could be added to any "core" grammar which supplies the required notions, to produce a grammar which is known to implement our abstraction of what topicalisation is. As above, we may have to "fit" the syntax of the core grammar to that required by the argument of the parameterised account, for instance (again) by defining sorts using axioms like $s : \mathtt{S} \leftrightarrow s =_{\mathtt{cat}} \mathtt{S}$ and $r : \mathtt{XP} \leftrightarrow r =_{\mathtt{bar}} \mathtt{NP}$. Of course it is entirely possible that the resulting grammar will nevertheless suffer clear deficiencies. Hopefully, this should give us some clues as to deficiencies in our current abstract topicalisation description.

5.2 Unbounded Dependencies

Adequate treatment of so-called unbounded dependency phenomena has been an important criterion in evaluating linguistic theories for many years. Any recent general introduction to syntax in linguistics (for instance Radford 1988) should give an outline of such phenomena. Here we will consider the particular cases of topicalisation, and wh-relative clauses.

English is sometimes called an *SVO* language, because the so-called canonical sentence order is subject-verb-object (*John loves Mary*). Topicalisation is one way of varying this order. For instance, in the sentence *On the table, I placed a cup*, the *topic* is the phrase *on the table.* We might then refer to the remainder of the sentence as the *comment.* One way of describing this sentence (though not necessarily its production) would be to say that the topic seems to have moved out of its canonical position (in *I placed a cup on the table*), and ended up instead at the beginning of the sentence. This process is sometimes referred to as *extraction.* Clearly there is some sort of relationship between possible topics and possible extraction sites. For instance, they share category (in the sense of $\_ =_{\mathbf{cat}} \_$): one cannot have a PP topic corresponding to an NP extraction site (so *On the table, I placed a cup on* is forbidden). We say that a *dependency* exists between them.

A wh-relative clause is a phrase like *who smiles*, which can be used to modify a noun phrase such as *the man* (*the man who smiles*). The essential observation is that a wh-relative is like a sentence in which some noun phrase is replaced by a relative pronoun (such as *who*), so replacing *the man* in *the man smiles* by *who* produces *who smiles.* Examples can become much more complicated: from the topicalised sentence, *the woman in the red shirt, Harry saw Mary walking with* we produce *the woman in the red shirt who Harry saw Mary walking with.* Again, we might say that the NP has been (or appears as though it has been) replaced by a relative pronoun to form the relative clause.

As discussed in Section 3.1, some grammars may not make any external distinction between NPs, so that a context well-formed for one NP is well-formed

for any other. Such a grammar must ignore issues such as agreement, allowing, for instance, *John smile* as a sentence. Thus also it can place no agreement restrictions on an NP being modified by a relative clause, and the "vacated" NP position of the relative clause, allowing, for instance *the man who smile.* Many grammars, however, do take sufficient account of agreement phenomena to disallow *John smile*, and so we would expect them to similarly account for the sort of agreement restrictions in the use of relatives which would outlaw examples like *the man who smile.* This shows a slightly more sophisticated dependency than that outlined for *on the table.* An obvious candidate method for capturing this dependency is suggested by the above procedural description of extraction. This would be to say that it must at least be *possible* to put the topic into the extraction site to form a sentence, and for topicalisation, this seems sufficient. For relatives, however, it is not. For instance, we can say *he whom I curse dies*, but not *I curse he.* For relatives, the NP being modified must be allowed to have a different grammatical case to that imposed by the extraction site. The dependency, then, does *not* extend to case.

Note also that it is not enough for extraction that a substring be merely a *possible* NP. For instance, *the jar on the table* can act as a noun phrase, as in *the jar on the table is full of salt.* So removing *the jar on the table* from the sentence *he placed the jar on the table* would produce a sentence missing a noun phrase, to wit *he placed.* However, prepending this by the appropriate relative pronoun does *not* produce a candidate for a relative clause: one cannot say *Here is the jar on the table which he placed.* The NP string must *act* like an NP in the sentence in question. One must be able to parse it as an NP, within the sentence. In short, it must be a constituent.

Moreover, topicalisation and relativisation (as I shall call the process of modifying an NP with a relative clause) are examples of *long distance*, or *unbounded*, dependency, because the extracted item need not be an immediate constituent of the sentence, but may be a constituent of a constituent, or a constituent of a constituent of a constituent, and so on to apparently arbitrary depth. We can

have not only *the man who smiles*, but also *the woman in the red shirt who Harry saw Mary walking with.*

Many, if not most, grammars (for instance LFG, in Kaplan and Zaenen (1987)) explicitly restrict the class of NPs which can be extracted. For example, we might say that the NP must not be part of an *adjunct* (optional modifier): so we may have *the man with the glove on his left hand sang* and *the glove fell* but not *the glove which the man with on his left hand sang fell.* A first specification is to say simply that only constituents may be extracted, *without* saying that all constituents may be. Later, as we refine our abstraction, we can say something more about which kind of intension can be extracted.

There are also examples of unbounded dependencies in which a constituent appears to have been "extracted" from more than one site. While so-called *across-the-board* examples (*the cat which John loved and Mary hated*) might well be obtained by further enriching our specification by a treatment of co-ordination, this cannot be said of examples involving *parasitic gaps* (*the papers which I filed without reading*).

In general, in English, it appears one cannot extract from an item from which extraction has already occurred. For instance, we can have *Mary, I believe John loves*, but not (most people would say) *John, Mary, I believe loves.*[1] Note that in a topicalised sentence, it is the comment, not the topic, from which further extraction is disallowed. Thus the preceding description of relatives, which involves extraction from the topic, does not violate the restriction on double extraction.

---

[1]This does not hold in some languages, such as Swedish. There are also a few examples in English (e.g. *the violin on which this sonata is easy to play*) involving another UDC (so-called *tough movement*, or missing object construction) which appear to violate the restriction. See for instance Maling and Zaenen (1982).

5.3 Equality and Level of Abstraction

We might describe a topicalised sentence by $\mathrm{S} \mapsto t \cdot \mathrm{S}/t$, where S stands for sentences, t is a topic, and S/t stands for the corresponding comments, which are like sentences with "t-gaps". (Evidence that comments ought to be considered constituents seems to be limited largely to co-ordination, but as most accounts assume such constituents, I shall follow suit). However this does not allow for the possibility that intensions may codify whole trees, as they are usually described as doing in LFG or GB, in which case we require several "gappy" intensions according to where the topic came from. Nor does it allow the possibility that there may be different intensions to dominate topicalised or untopicalised sentences. To allow this, we would need to be able to describe the parallel distribution of the two types of sentence. We might use some syntax like $s \equiv s'$ to indicate that two intensions can occur in like places. So we will need to insist, for instance, that

$$
\begin{aligned}
& s \equiv s' \\
& \quad \rightarrow (((\exists t.t \mapsto x \cdot s \cdot y) \vee (\exists t' \,.\, t' \mapsto x \cdot s' \cdot y)) \\
& \qquad\qquad \leftrightarrow ((\exists t \,.\, t \mapsto x \cdot s \cdot y) \wedge (\exists t' \,.\, t' \mapsto x \cdot s' \cdot y) \wedge t \equiv t'))
\end{aligned}
$$

This might be paraphrased, if s distributes like s', then s or s' can appear between sisters x and y in a local tree, if and only if *both* s and s' can appear between x and y in a local tree, and the trees so created are themselves distributionally equivalent. This is trying to capture a notion of substitution. While this is certainly a necessary condition on substitution, it is probably not sufficient. If we want to identify the result of substituting s' for s in some context, we will need syntax to identify just which t' corresponds to substituting s' for which occurrence of s. This quickly becomes unwieldy (though the axiom above is bad enough!). It seems to me that this idea of substitutability ought to be captured by the notion of equality employed in the intensional domain. Even in calculi in which sentential intensions are marked for topicalisation, the two kinds are rarely given different distributions. In a system which is not interested in matters such

as number agreement, we can safely form a quotient which equates objects which only differ in this respect. In a system which concerns itself with semantic considerations, á la Montague (1973), we must maintain semantic distinctions when considering questions of substitution. I will proceed on the basis that the same sentential intension which dominates a sentence in canonical order, also dominates the topicalised version. (This need not make it any more difficult to ban double extraction, provided that the comment intension *is* marked in some way). Then a topicalised sentence must have the same distribution as the untopicalised.

This means dispensing with models in which intensions correspond one-to-one to full parse trees. I will consider how we might make such a move for LFG. I think GB should be susceptible to the same strategy. The strategy also owes something to the style of subcategorisation analysis employed in GPSG or HPSG. In the HPSG of Pollard and Sag (1987), it is not clear what part of a sign is relevant to distribution. In more recent work in HPSG (Pollard and Sag 1992), it seems clear that the values of daughters should not be consulted in determining whether their mother can in turn fulfill the obligations of a daughter in a given construction. In particular, the encoding of subcategorisation explicitly excludes reference to the daughter attributes. In LFG, distribution is largely determined by category label, but then we have the complication that certain trees are excluded by a final filtering process. We would like to be able to characterise at any node exactly what information from further down the tree might conceivably cause a tree to be rejected by this filter. One class of models which allows this has intensions consisting of an atomic category label, plus a version of f-structure such that features are marked as being satisfied either *internally* (in which case they can only appear at the root of parse trees such that the feature's value is a subtree), or *externally* (in which case they can only appear at the root of trees such that the feature's value is *not* a subtree). In Figure 5–1 we see a parse tree for a topicalised sentence, in which grammatical functions are marked with a superscript x if their values are external. I have simplified somewhat by writing just "Mary" or "John" for the NP intensions. I have also written in the annotated rules justifying each step of the parse, but if one ignores the right hand side of these, each node is

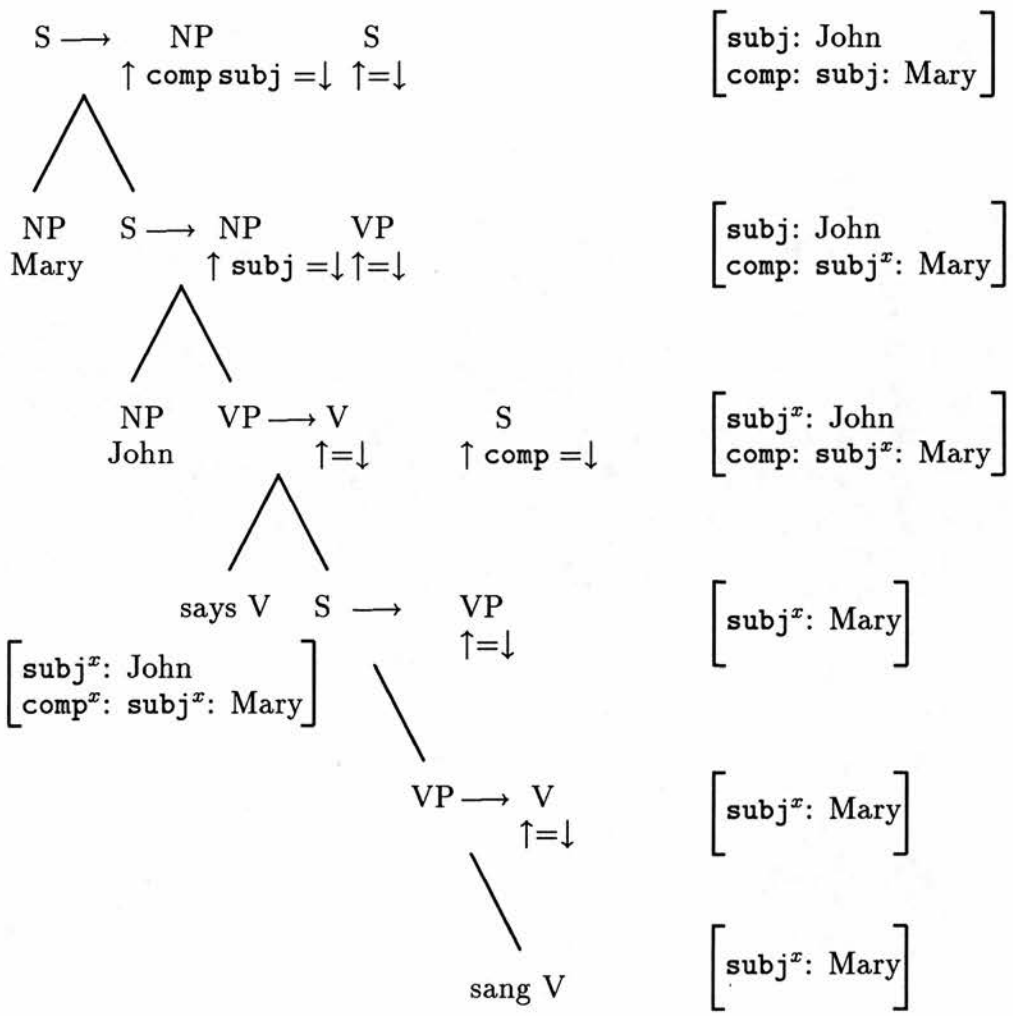

Figure 5–1: Topicalised sentence in LFG: *Mary, John says sang*

decorated by a representation of the corresponding intension: an atomic category label and an x-marked f-structure. So for example corresponding to the word *says* we have the intension

$$\mathrm{V}\left[\begin{array}{l}\mathtt{subj}^{x}\text{: John}\\ \mathtt{comp}^{x}\text{:}\mathtt{subj}^{x}\text{: Mary}\end{array}\right].$$

Here `subj` is external and so its value (the NP *John*) does not appear below it in the parse tree. However for the S intension at the right hand daughter of the root, `subj` is not marked external, and so its value (again the NP *John*) must appear below it in the tree: in this case, at its left hand daughter. With respect to such models, we can take it that part of the meaning of the rule

$$\begin{array}{lcll}\mathrm{VP} & \longrightarrow & \mathrm{V} & \mathrm{NP}\\ & & \uparrow{=}\downarrow & \uparrow\,\mathtt{obj}\,{=}\downarrow\end{array}$$

is that the `obj` value will be external at the V node, and internal at the VP node. So, if an S intension were to have as a realisation *John hates* (where we assume *hates* is strictly transitive) then that S intension must have its `obj` value marked external, and so does not stand for a matrix sentence. In standard presentations of LFG, an S dominating *John hates* will, because of the procedural definition of the correspondence between c-structure and f-structure, have `obj` undefined, contrary to the lexical requirements of *hates*. When LFG is augmented with functional uncertainty (as in Kaplan and Zaenen (1987)), this "procedure" is non-deterministic. By introducing this marking for externality, we can localise the correspondence between c-structure and f-structure, and make it declarative: when a rule says $\uparrow\mathtt{obj}\,{=}\downarrow$, then `obj` is internal in the mother, and external in the head daughter. Writing a specification for this somewhat unconventional (in LFG orthodoxy) class of models ought certainly to be possible, but is also likely to be tedious, complicated, and not, at this point, especially enlightening. All I have tried to do here is to convince the reader that by localising the correspondence between c-structure and f-structure, we can produce an adequate class of models

in which intensions need not encode whole trees. Then we can again allow equality of intensions to encode distributional equivalence, and simply say that the same intension can dominate a topicalised sentence and its canonical equivalent. Thus the intension at the root of *Mary, John says sang* in Figure 5–1 can also stand at the root of its canonical equivalent, *John says Mary sang.*

5.4 Topicalisation

Following on the above arguments, I proceed on the basis that the level of abstraction inherent in the equality predicate should give us a sufficient account of distribution equivalence (replaceability). We need some vocabulary to refer to sentential intensions, and maximal projections.

(5.1) (Maximal Projections)

XP = **sorts** XP,Syn
axiom XP $\subseteq$ Syn

(5.2) (Sentential intensions)

SENT = **sorts** S,Syn
axiom S $\subseteq$ Syn

We might try to characterise topicalised strings by a predicate top such that

$$\texttt{top}(w) \rightarrow \exists s{:}\texttt{S}, t{:}\texttt{XP}, xy{:}\texttt{Syn}* \,.\, s \mapsto^* t \cdot x \cdot y \wedge t \cdot x \cdot y \mapsto^* w \wedge s \mapsto^* x \cdot t \cdot y$$

So if the string w is a topicalised sentence, there must be a sentential intension s and an intensional string $t \cdot x \cdot y$, where t is a maximal projection, such that s licenses w via $t \cdot x \cdot y$, and such that also $s \mapsto^* x \cdot t \cdot y$ (the corresponding canonical order). This fails to group x and y together as a comment constituent. A predicate com which identifies such constituents can replace top.

$$\forall s{:}\texttt{S}, t{:}\texttt{XP}, u{:}\texttt{Syn} \,.\, \texttt{com}(u, s, t) \rightarrow$$
$$\exists xy{:}\texttt{Syn}* \,.\, s \mapsto^* x \cdot t \cdot y \wedge s \mapsto t \cdot u \wedge u \mapsto^* x \cdot y$$

If u is a comment for the sentence s and topic t, then $s \mapsto t \cdot u$, and also there is a parse $x \cdot y$ of u such that $s \mapsto^* x \cdot t \cdot y$. This does not allow more than one extraction site. It would also be preferable if our definition were to admit refinement in different directions for different UDCs. In a comment structure, t is simply left out (replaced by e), but in a wh-relative, it is replaced by a relative pronoun, such as *who* or *which*. The important features of a UDC are

(5.3) the top: the highest local tree at which the dependency occurs. For topicalisations, this is where the comment is introduced; for wh-relatives, the relative clause.

(5.4) the middle: both comments and relative clauses parallel the structure of a sentence.

(5.5) the bottom: within this structure, a particular string (empty, or a relative pronoun) is substituted for one or more instances of a particular intension (the topic, or the NP undergoing modification by the relative).

Although the description of UDCs in these terms is clearly influenced by GPSG, this does not mean that such terms of description cannot be equally applied to other kinds of grammar, as we shall see. When we come to produce a particular (parameterised) account, however, it will be natural to choose one which is also heavily influenced by GPSG.

The use of rules which introduce UDC intensions is sufficient to characterise the top. From now on I will refer to these special UDC intensions (which stand for comments or relative clauses) as simply UDCs. For the middle, we need to be able to express the structure (sentence) to which a UDC corresponds. At the bottom, we need to know the dependent intension (extraction site), and the string to substitute in for it.

(5.6) UDS = **sorts** Udc,Syn
axiom Udc $\subseteq$ Syn

(5.7) UDC = **enrich** UDS+SYN*

by **opn** uhd : Udc → Syn
opn udep : Udc → Syn
opn uft : Udc → Syn*

references: SYN* (2.3)

To every UDC u there corresponds a sentential intension $\mathtt{uhd}(u)$ which stands at the top of the canonical structure which the UDC parallels, as well as a dependent intension (topic or NP being modified) $\mathtt{udep}(u)$. In the UDC parse, one or more occurrences of $\mathtt{udep}(u)$ will be replaced by the foot value $\mathtt{uft}(u)$ (**e** or a relative pronoun). I aim to say that a realisation of a UDC of s with respect to t (i.e. $u : \mathtt{Udc}$ such that $\mathtt{uhd}(u) = s$ and $\mathtt{udep}(u) = t$) consists of a realisation of s with one or more t altered in a regular fashion. To say that a sentence can be topicalised with respect to a constituent t is to say that a sentential intension s may dominate t followed by a comment of s with respect to t. We need to be able to tell when two strings differ by one or more substitutions. This will be the job of the **feet** predicate: $\mathtt{feet}_{t,w}(x, x')$ should mean that x' is like x except that one or more t have been replaced by w.

(5.8) (Substitute w for one or more t)

FEET = **extend** SYN*
by **pred** $\mathtt{feet}\_,\_(\_,\_) \subseteq \mathtt{Syn} \times \mathtt{Syn}* \times \mathtt{Syn}* \times \mathtt{Syn}*$
axiom $\mathtt{feet}_{t,w}(x \cdot t \cdot y, x \cdot w \cdot y)$
axiom $\mathtt{feet}_{t,w}(x, x') \wedge \mathtt{feet}_{t,w}(y, y') \rightarrow \mathtt{feet}_{t,w}(x \cdot y, x' \cdot y')$

(5.9) PSIG = **enrich** SYN*
by **pred** $\_ \mapsto^* \_ \subseteq \mathtt{Syn}* \times \mathtt{Syn}*$

(5.10) (Realisation)

UDR = **enrich** UDC + FEET + PSIG
by **axiom** $\forall u{:}\mathtt{Udc}, y{:}\mathtt{Lex}* \,.\, u \mapsto^* y$
$\rightarrow \exists x \,.\, \mathtt{uhd}(u) \mapsto^* x \wedge \mathtt{feet}_{\mathtt{udep}(u),\mathtt{uft}(u)}(x, y)$

(5.11) UDX1 = **enrich** UDC
by **axiom** $\neg\,\mathtt{uhd}(u){:}\mathtt{Udc}$

(5.12) (Slash constructions are UDCs)
SLASH = **enrich** UDC
by **sort** `Slash`
axiom $\mathtt{Slash} \subseteq \mathtt{Udc}$
axiom $\forall u{:}\mathtt{Slash}\,.\,\mathtt{uft}(u) = \mathtt{e}$

(5.13) (Topicalisation)
TOPIC = **enrich** SENT+XP+UDC+SLASH+GMR
by **axiom** $\forall u{:}\mathtt{Slash}\,.\,\mathtt{uhd}(u){:}\mathtt{S} \wedge \mathtt{udep}(u){:}\mathtt{XP}$
$\rightarrow \mathtt{uhd}(u) \mapsto \mathtt{udep}(u) \cdot u$

references: SYN* (2.3), UDC (5.7), SENT (5.2), XP (5.1), GMR (2.4)

$\mathtt{feet}_{t,w}(x, x')$ is satisfied when x' is like x except that one or more t have been replaced by w. By the first axiom this holds where x and x' differ by one substitution. The second axiom extends this to strings differing by more than one substitution. Thus the axiom of UDR ensures that every lexical realisation of a UDC u must be like some x such that $\mathtt{uhd}(u) \mapsto^* x$, except that one or more instances of the dependency $\mathtt{udep}(u)$ are replaced by $\mathtt{uft}(u)$ (`e` for topicalisations, a relative pronoun in wh-relatives). UDX1 forbids double extraction, by insisting that the intension at the head of any canonical parse paralleling a parse of u in `Udc` is not itself in `Udc`. If a grammatical system has different types of intensions for different types of UDC, we must be able to distinguish those which take part in, say, topicalisation, from those which take part in relativisation: this is the purpose of the sort `Slash`. The axiom of TOPIC is responsible for introducing a topic-comment structure: if u is a `Slash` intension, then $\mathtt{uhd}(u)$ is a sentential intension which can immediately dominate daughters $\mathtt{udep}(u)$ (topic) and u (comment). An account of topicalisation ought to refine UDR + UDX1 + TOPIC.

This is not to say that every type of maximal projection need be subject to extraction, or from every position. These are matters which, for the moment,

are allowed to vary from grammar to grammar. As we become more sure of our ground, we may refine our abstract notion of topicalisation to restrict this freedom. We could fairly easily, for instance, forbid extraction of finite VPs, if this seems to be the right thing to do.

While I assume that the same sentential intension which stands at the head of a canonical parse also dominates a parse of a topicalised version, I do not assume that there is only *one* sentential intension: there may be other distinctions they need to recognise. For instance, there is a class of sentences whose syntactic subjects appear to act just as place-holders, as in *It appears they act as place-holders.* One may wish to have distinct intension(s) for these, because they behave differently in some respects to other sentences. For instance, one cannot form the relative clause *the thing which appears they act as place-holders.* (One may wish to say that the subject of *It appears they act as place holders* is really *they*, but we would still need some intensional distinction to account for the fact that, under such an analysis, the subject appears in a non-standard position). But such sentences can undergo topicalisation: *Place-holders, it appears they have been known to act as — but as relatives, never.*

Consider how our abstract specification relates to existing accounts. In a system (such as GPSG) where intensions can be conceived of as quite local structures (they do not bear the traces of particular daughters), we can implement `Slash` using a GPSG-style slash function, $u = s/t$. It should be fairly clear that, with appropriate definitions of `feet` and so on, the GPSG treatment meets the conditions of FEET+UDC0+SLASH+TOPIC — or at least it is clear that it was intended to meet such conditions. In order to show this formally we would first need to produce a complete specification for a GPSG grammar which treats topicalisation. A more practical way to proceed will be to note that the conditions are certainly part of the intention behind GPSG, or whatever grammar is being considered, and then, when the abstract specification has been seen to be acceptable, go on to produce a formal, parameterised implementation of (say) the basic GPSG technique, which will be applicable in extending a broad range of core grammars to

cover topicalisation, as well as core GPSG. I intend to give a simple parameterised implementation in this style shortly.

But first we should consider whether the abstraction covers other accounts as well. Broadly speaking, constituency treatments seem to fall into two classes: those, like GPSG or categorial-style treatments, which rely on a relatively limited type of inheritance to transmit non-local information up the tree; and those, like LFG, GB or HPSG, in which intensions themselves correspond to whole trees (we can think of them as equivalence classes thereof), and something like a pointer into the tree is passed up in some fashion. Put like this, they don't sound so very different. In the latter kind of treatment, topics are an example of trees in which a particular syntactic obligation is satisfied externally.

In an LFG system of the sort sketched in Section 5.3, for instance, the requirement that grammatical functions take values within the tree is really just another property required of matrix sentences, like having a finite verb form, or not having an overt complementiser, or (in HPSG) having an empty subcat list. In the models of Section 5.3, an intension u can be a comment (i.e. an object of `Udc`) if

1. its category label is S,

2. $\mathtt{udep}(u)$ is the value in the f-structure at some path $\mathtt{comp}:\mathtt{comp}:\cdots\mathtt{comp}:f$, where the grammatical function $f \neq \mathtt{comp}$ (I simplify somewhat),

3. $\mathtt{udep}(u)$ represents the only externally met syntactic obligation of u, and

4. $\mathtt{uhd}(u)$ is just like u but has no syntactic obligations marked external.

Then also $\mathtt{uhd}(u) \mapsto \mathtt{udep}(u) \cdot u$ in these models. For example, where u is the intension at the right daughter of the root in Figure 5–1, u has category label S and the value at $\mathtt{comp}:\mathtt{subj}$ is the only external obligation at u, $\mathtt{udep}(u)$ is the NP "Mary", and $\mathtt{uhd}(u)$ is the root intension.

5.5 A Simple Implementation: Topicalisation

The basic idea of slash-categories is very simple. From a "core" category s and an XP t, we can form a new category s/t. If s can dominate a string w, s/t can dominate a string w' similar to w in much the way described by $\mathbf{feet}_{t,\mathbf{e}}(w, w')$. For the purposes of this example, I will assume that if s may dominate some $x \cdot r \cdot y$, then s/t may dominate $x \cdot (r/t) \cdot y$, and also that t/t may dominate $\mathbf{e}$. So I will allow any XP to be extracted once only, from anywhere within any parse. Of course this implementation undergenerates in not allowing for, say, across-the-board phenomena, and overgenerates in allowing extraction from too many sites, but the mere fact of having an implementation may give us some further insights into the correct abstract notion of topicalisation, and give us some clues on how to go about getting a better implementation. This will be a parameterised implementation, which adds a slash-category account to any core grammar which gives us sentences, maximal projections, and a $\mapsto$ predicate.

(5.14) TCOR = **derive from** GMR + SENT + XP
by [Cor $\longmapsto$ Syn, Cor* $\longmapsto$ Syn*]

TSYN = **extend sort** Cor
by **sort** Syn
axiom Cor $\subseteq$ Syn
opn _/_ : Cor $\times$ Cor $\rightarrow$ Syn

(5.15) TTOP = **extend** TCOR+TSYN+SYN*
by **pred** _ $\mapsto$ _ $\subseteq$ Syn $\times$ Syn*
axiom $\forall s$: S, t : XP. $s \mapsto t \cdot s/t$
axiom $\forall xy$: Cor* . $s \mapsto x \cdot r \cdot y \rightarrow s/t \mapsto x \cdot r/t \cdot y$
axiom $t/t \mapsto$ e

(5.16) TIMP = $\lambda\mathcal{X}$: TCOR . $\mathcal{X}$ + TTOP

references: GMR (2.4), SENT (5.2), XP (5.1), SYN* (2.3)

TCOR describes the type of core grammar the construction requires: we need to know about sentences, maximal projections, and immediate constituency. We rename its intensions `Cor`. In TSYN, `Cor` becomes a subsort of a new, extended `Syn`, which also has new objects of the form r/t, where r and t are old objects (i.e., in `Cor`). At this stage, $\mapsto$ only covers the old intensions: it is the job of TTOP to extend it to the new ones. The first axiom accounts for topicalisation by the introduction of a slash term, the second describes how its slash value is to be passed to one of its daughters, and the third discharges a slash by allowing any t/t to dominate the empty string. Since every new instance of $\mapsto$ involves a /-term (which cannot evaluate to a member of `Cor`), this extension is hierarchically consistent. In the second axiom of TTOP, it is important that x and y be limited to `Cor`$*$, otherwise S $\mapsto$ PP · S/PP would imply that also S/NP $\mapsto$ PP/NP · S/PP, which allows a topic to be extracted from a topic, as in *Cats, of, John has a fear.* TIMP shows how the preceding specifications may be used to extend any model of TCOR (i.e. any model which knows about sentences, maximal projections and immediate constituency) to form a model which also has a (rather simplistic) account of topicalisation.

Obviously TIMP is not in itself an implementation of UDX1+UDR+TOPIC (5.11, 5.10, 5.13), but we can establish a one-to-one correspondence between models TIMP($\mathcal{X}$) and a certain class of models of UDX1 + UDR + TOPIC. To do this, we show how the extra syntax required by UDX1 + UDR + TOPIC can be defined in models TIMP($\mathcal{X}$). Any syntax in those models which is *not* required by UDX1+UDR+TOPIC could then be derived away, to leave implementations proper. We know how to get $\mapsto^*$ from $\mapsto$. `Slash`-objects are the terms r/t. Since this is only an implementation of topicalisation, we may as well suppose `Slash`-objects are the only `Udc`-objects. Put $\mathtt{uhd}(r/t) = r$, $\mathtt{udep}(r/t) = t$ and $\mathtt{uft}(r/t) = \mathtt{e}$. The axioms of UDX1 (5.11), SLASH (5.12) and TOPIC (5.13) are then satisfied.

Of course sometimes this process of demonstrating implementation proves unsuccessful, possibly because of deficiencies in the attempted implementation, or

possibly because of a fault in the abstract specification. There is something to be learned from either. In an earlier attempt, I had not included $\texttt{udep}(u):\texttt{XP}$ in the axiom of TOPIC (5.13). This meant that the axiom was insisting that if $s:\texttt{S}$, then $s \mapsto t \cdot s/t$ for *any* t: but in TIMP, this will only hold when $t:\texttt{XP}$. The abstraction was asking too much. There is another mismatch above between implementation and specification, which I previously glossed over. That is that the implementation says nothing about Lex*, but the axiom of UDR does. Instead of just models $\mathsf{TIMP}(\mathcal{X})$ for $\mathcal{X}:\mathsf{TCOR}$, we must look at models $\mathsf{TIMP}(\mathcal{X})$ where $\mathcal{X}$ ranges over the models specified by

derive from GMR+LEX*+LXN+SENT+XP **by**
$[\texttt{Cor} \longmapsto \texttt{Syn}, \texttt{Cor*} \longmapsto \texttt{Syn*}]$
references: TCOR (5.14), GMR (2.4), LEX* (2.1),
LXN (2.2), SENT (5.2), XP (5.1)

Then the axiom of UDR does follow from the previous definitions.

5.6 Refining the Abstraction

We should now see if there are other features of the implementation which might usefully be included in the abstract specification. One candidate might be the way that every parse of s/t in TIMP (5.16) parallels, in lock-step, some parse of s (the "middle" of 5.4). This is due to the axiom $s \mapsto x \cdot r \cdot y \rightarrow s/t \mapsto x \cdot r/t \cdot y$. This does not appear, on the face of it, to be a property of systems (like LFG) with intensions presented as trees, where properties (such as coherence and completeness) of full trees are applied at the final stage. But if we take the view that a comment (say) is distinguished from other nodes labelled S by its having an externally filled argument, then we can distinguish verb phrases in the same way, according to whether they have an externally filled argument (other than subj). Then this property of having an externally filled argument can again be viewed as a local property passed between mother and daughter. So the parse of the right

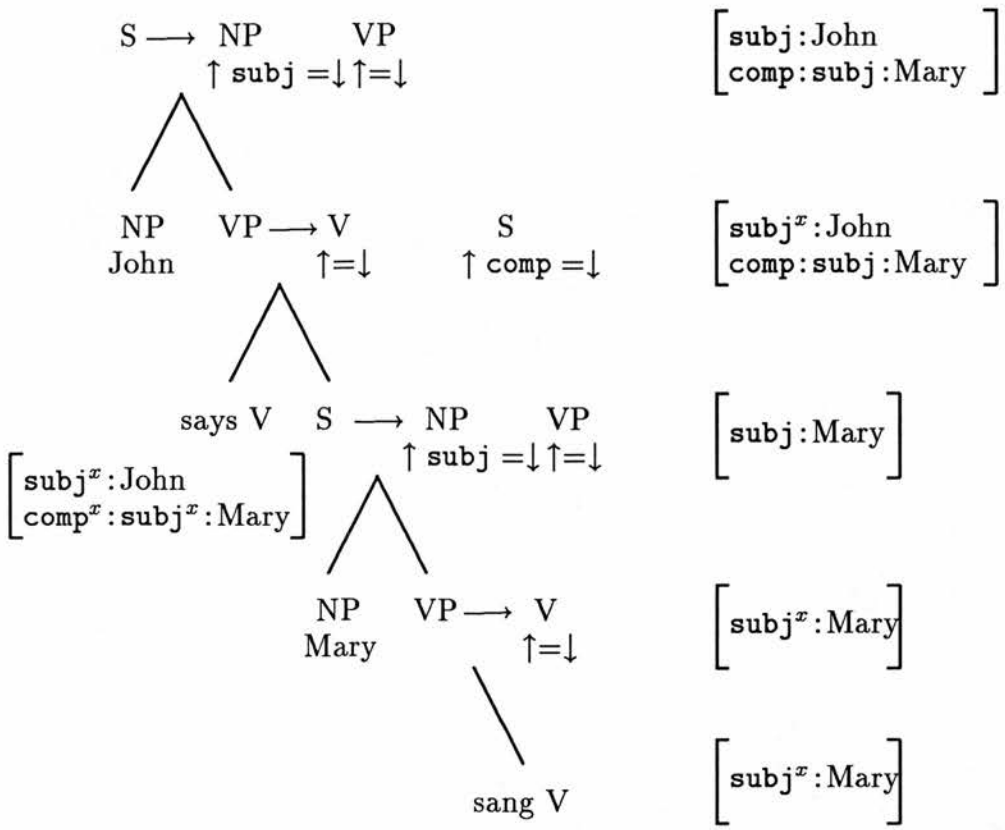

Figure 5–2: Canonical parse corresponding to comment parse

daughter in Figure 5–1 parallels in lock-step the parse shown in Figure 5–2, with at every step a Figure 5–1 node either being equal to the corresponding Figure 5–2 node, or absent entirely, or a UDC of the Figure 5–2 node. (In which case, its mother must also be a UDC of the Figure 5–2 mother, and the manner of the correspondence of the mothers, together with the rule which relates mothers and daughters, determines the manner of correspondence of the daughters. By manner of correspondence I mean the particular additional external argument present in a Figure 5–1 node as compared to the corresponding Figure 5–2 node.) Note I only present this as evidence that we might reasonably consider incorporating this lock-step property into our abstract specification. I certainly do not mean to imply that the LFG-style of account could be considered an instance of a slash-function account in the style of TIMP (5.16). That is clearly not true, as the LFG account is not functional in the same sense. For any given s and t there can be more than one "s with a t-gap", depending on which path into s the "extraction site"

is associated with. In the example of Section 5.3 it was `comp:subj`, but it might have been `obj`, or `comp:comp:obj2`, or any one of a host of others. Nor should it be thought that the lack of this lock-step property in the LFG account would *necessarily* have stopped us from including the property in a refined abstraction, if, for instance, its absence had seemed a clear deficiency of the LFG account. Let us now consider how this lock-step correspondence might be described as it occurs in TIMP (5.16).

$$\begin{aligned}\mathsf{TLOCK} \;=\;& \textbf{extend } \mathrm{Sig}[\mathsf{TSYN}] + \mathsf{SYN*} \\ & \textbf{by } \textbf{pred } \mathtt{slock\_(\_,\_)} \subseteq \mathtt{Syn} \times \mathtt{Syn*} \times \mathtt{Syn*} \\ & \quad \textbf{axiom } \mathtt{slock}_t(x \cdot r \cdot y, x \cdot (r/t) \cdot y)\end{aligned}$$

$$\begin{aligned}\mathsf{TLIMC} \;=\;& \textbf{enrich } \mathsf{TLOCK} + \mathsf{GMR} \\ & \textbf{by } \textbf{axiom } u = s/t \wedge u \mapsto x' \rightarrow \\ & \qquad (s = t \wedge x' = \mathtt{e}) \vee \exists x.s \mapsto x \wedge \mathtt{slock}_t(x, x')\end{aligned}$$

references: GMR (2.4), SYN* (2.3)

The construction of $\mathtt{slock}_t(x, x')$ is quite similar to that of FEET. It says x' differs from x in that an intension r is replaced by r/t. Thus `slock` describes the ways in which daughters of a UDC must correspond to daughters in a non-UDC parse, except for the case $t/t \mapsto \mathtt{e}$. For any model $\mathcal{X}$ of TCOR (5.14), the value at $\mathcal{X}$ of

$$\begin{aligned}&\lambda\mathcal{X}{:}\mathsf{TCOR}.\mathcal{X} + \mathsf{TTOP} + \mathsf{TLOCK} \\ &\qquad = \lambda\mathcal{X}{:}\mathsf{TCOR}.\mathsf{TIMP}(\mathcal{X}) + \mathsf{TLOCK}\end{aligned}$$

references: TCOR (5.14), TTOP (5.15), TIMP (5.16)

is a model of TLIMC. Abstracting from this description to fit in with other strategies for constructing UDC intensions ought not to be too difficult. There are several additional generalisations we might also add at this stage. Perhaps the most obvious is that we ought to allow more than one daughter intension to bear a t-dependency, to allow for phenomena like across-the-board or parasitic gaps. A second is to allow t on the non-UDC side to correspond directly to `e`, instead of

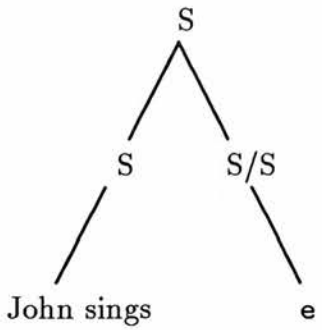

Figure 5–3: Spurious Topicalisation

insisting on always having it correspond to t/t (and then adding t/t dominating e). In fact a grammar which doesn't have intensions dominating e has several advantages. One is that it is susceptible to left-corner parsing. Another advantage is that it prevents the sort of spurious topicalisation seen in Figure 5–3.

(5.17) (UD daughter correspondence)

UDD = **extend** UDC **by**

pred $\mathtt{udd}\_,\_(\_,\_) \subseteq \mathtt{Syn} \times \mathtt{Syn}* \times \mathtt{Syn}* \times \mathtt{Syn}*$

axiom $\forall u{:}\mathtt{Udc}\,.\,\mathtt{udd}_{\mathtt{udep}(u),\mathtt{uft}(u)}(x \cdot \mathtt{udep}(u) \cdot y, x \cdot \mathtt{uft}(u) \cdot y)$

axiom $\forall u{:}\mathtt{Udc}\,.\,\mathtt{udd}_{\mathtt{udep}(u),\mathtt{uft}(u)}(x \cdot \mathtt{uhd}(u) \cdot y, x \cdot u \cdot y)$

axiom $\mathtt{udd}_{t,w}(x, x') \wedge \mathtt{udd}_{t,w}(y, y') \rightarrow \mathtt{udd}_{t,w}(x \cdot y, x' \cdot y')$

UDIMC = **enrich** UDD **by**

axiom $\forall u{:}\mathtt{Udc}\,.\,u \mapsto x' \rightarrow \exists x\,.\,\mathtt{uhd}(u) \mapsto x \wedge \mathtt{udd}_{\mathtt{udep}(u),\mathtt{uft}(u)}(x, x')$

UDL = **enrich** UDX1 + UDIMC **by**

axiom $\forall u{:}\mathtt{Udc}\,.\,u{:}\mathtt{Lex} \rightarrow \mathtt{uhd}(u) = \mathtt{udep}(u) \wedge \mathtt{uft}(u) = u$

references: UDX1 (5.11)

The first axiom of UDD gives us $\mathtt{udd}_{\mathtt{udep}(u),\mathtt{uft}(u)}(x, x')$ for any u when x and x' differ by one substitution of $\mathtt{uft}(u)$ for $\mathtt{udep}(u)$. The second does the same for one substitution of u in place of $\mathtt{uhd}(u)$. The third extends this to cover more than one such difference. Then UDIMC insists that immediate constituents of a UDC u must differ from immediate constituents of $\mathtt{uhd}(u)$ in just this way. UDIMC+PARSE (2.5) gives us the axiom of UDR (5.10), provided that any UDC u corresponding to a non-UDC $\mathtt{uhd}(u)$ is not a word (element of **Lex**), but this may not always be true.

As an example of why we might have a UDC in **Lex**, in a simple account of relatives involving, say, *which*, we might use, in analogy to _/_ in TIMP (5.16), a constructor $\mathtt{wh} : \mathtt{Cor} \times \mathtt{Cor} \to \mathtt{Syn}$. So we might introduce the relative clause with a rule like $\mathrm{S} \mapsto \mathtt{wh}(\mathrm{S}, \mathrm{NP})$. The middle part of the construction would be taken care of by an axiom like $s \mapsto x \cdot r \cdot y \to \mathtt{wh}(s,t) \mapsto x \cdot \mathtt{wh}(r,t) \cdot y$. When it comes to the bottom of the construction, we must somehow get from $\mathtt{wh}(\mathrm{NP},\mathrm{NP})$ to the lexical item **which**. While we could certainly put $\mathtt{wh}(\mathrm{NP}, \mathrm{NP}) \mapsto \mathtt{which}$, we could just as reasonably set $\mathtt{wh}(\mathrm{NP}, \mathrm{NP}) = \mathtt{which}$. In that case we will get the axiom of UDR provided $u = \mathtt{uft}(u)$ and $\mathtt{uhd}(u) = \mathtt{udep}(u)$. Hence every model of PARSE+UDL is a model of UDR.

This refinement of the abstraction in turn suggests a way we might go about improving the implementation, which is to use the correspondence **udd** to *define* the behaviour of $\mapsto$ on slash-categories.

USYN = **derive from**
  reachable enrich UDC **by**
    opn _/_ : $\mathtt{Cor} \times \mathtt{Cor} \dot{\to} \mathtt{Udc}$
    axiom $\mathtt{Cor} \subseteq \mathtt{Syn}$
    axiom $\mathtt{uhd}(s/t) = s$
    axiom $\mathtt{udep}(s/t) = t$
  on {Udc,Syn}
by [Slash $\longmapsto$ Udc]

(5.18) UTOP = **extend** TCOR+USYN+UDD **by**

$$\textbf{pred } \_ \mapsto \_ \subseteq \mathtt{Syn} \times \mathtt{Syn}*$$
$$\textbf{axiom } \forall s{:}\mathtt{S}, t{:}\mathtt{XP} \,.\, s \mapsto t \cdot s/t$$
$$\textbf{axiom } \forall x{:}\mathtt{Cor}* \,.\, s \mapsto x \wedge s' = s/t \wedge \mathtt{udd}_{t,\mathbf{e}}(x, x')$$
$$\rightarrow s' \mapsto x'$$

references: UDC (5.7), TCOR (5.14), UDD (5.17)

If _/_ were total in USYN, then the axioms given there, with the imposition of reachability, would make _/_ just the same as if it had been defined using **extend** instead of **enrich**. Thus the degree of definedness of _/_, and values of `uft`, are the only slack in USYN, and if we have another specification, say UXOK, which pins this down, then UXOK+UTOP+SLASH (5.12) can form the basis of implementations in much the same way as TTOP (5.15) did previously. For instance UANYX, below, allows any extraction, by making _/_ total, and so UIMP is a parameterised implementation which allows any XP to be extracted from anywhere within any sentence, forming a topic, and leaving a comment.

$$\begin{aligned} \mathsf{UANYX} \;=\; & \textbf{sorts } \mathtt{Cor}, \mathtt{Slash} \\ & \textbf{opn } \_/\_ : \mathtt{Cor} \times \mathtt{Cor} \rightarrow \mathtt{Slash} \end{aligned}$$

$$(5.19) \quad \mathsf{UIMP} \;=\; \lambda \mathcal{X}{:}\mathsf{TCOR} \,.\, \mathcal{X} + \mathsf{SLASH} + \mathsf{UANYX} + \mathsf{UTOP}$$

5.7 Relativisation

Here are some different forms of relative clause:

1. the woman Mary loves (is here)
2. the picture that hung there (has gone)
3. the shirt the colour of which I hate (is missing)

We might describe the first two by axioms like

1. $\forall u{:}\texttt{Slash}, s{:}\texttt{S}, n{:}\texttt{NP}\,.\,\texttt{uhd}(u){:}\texttt{VP} \wedge s \mapsto n \cdot \texttt{uhd}(u) \wedge \texttt{udep}(u){:}\texttt{NP}$
$\rightarrow \texttt{udep}(u) \mapsto^* \texttt{udep}(u) \cdot n \cdot u$

2. $\forall u{:}\texttt{Slash}\,.\,\texttt{uhd}(u){:}\texttt{S}' \wedge \texttt{udep}(u){:}\texttt{NP} \rightarrow \texttt{udep}(u) \mapsto \texttt{udep}(u) \cdot u$

The first axiom is intended to describe an NP like *the woman Mary loves* by saying that an intension (NP′ say) which licenses *the woman* may also license *the woman Mary loves*, *via* an intension string $\text{NP}' \cdot \text{NP} \cdot u$, where u is a slash intension with $\texttt{udep}(u) = \text{NP}'$ and $\texttt{uhd}(u)$ a VP such that $\text{NP} \cdot \texttt{uhd}(u)$ can form a sentence (for instance $u = \text{VP/NP}$). The idea in the second axiom is that an intension (NP say) which licenses *the picture* will also also license *the picture that hung there*, if there is a slash intension u where $\texttt{udep}(u) = \text{NP}$ and $\texttt{uhd}(u)$ licenses the complementised sentence *that the picture hung there* (perhaps S′/NP). This is achieved by setting $\texttt{udep}(u) \mapsto \texttt{udep}(u) \cdot u$ (e.g. $\text{NP} \mapsto \text{NP} \cdot (\text{S}'/\text{NP})$).

I don't wish to claim that these constitute very general (or even good) descriptions, but only to argue that accounting for these types of relative clause does not seem to require any really new machinery. The final type however, the *wh-relative*, is a little different. A wh-relative is formed using a relative pronoun, like *who* or *which*, which stands in a place where, in a matrix clause, we might expect to see an ordinary noun phrase. As noted before, there is a grammatical dependency between the position in the corresponding canonical construction, occupied in the relative construction by the relative pronoun, and the noun phrase that the clause modifies. For instance we can have *the woman who loves Mary* but not *the women who loves Mary*. Moreover, once again, it is possible to intercede a potentially unlimited amount of material between the NP being modified and the relative pronoun: *the book which irks me*, *the book the cover of which irks me*, *the book the colour of the cover of which irks me*, and so on. Thus wh-substitution represents a new kind of UDC.

(5.20) RELPRO = **enrich** LXN
by **opn** wh :→ Lex

(5.21) WH = **enrich** RELPRO+UDC
by **sort** Wh
axiom $\mathtt{Wh} \subseteq \mathtt{Udc}$
axiom $\forall u{:}\mathtt{Wh}.\mathtt{uft}(u) = \mathtt{wh}$

(5.22) NP = **sorts** NP,Syn
axiom $\mathtt{NP} \subseteq \mathtt{Syn}$

WHREL0 = **enrich** WH + AGR + SENT + NP + UDC + LEX* + PSIG
by **axiom** $\forall s{:}\mathtt{S}, n{:}\mathtt{NP}, u{:}\mathtt{Wh}\,.\, s \mapsto \mathtt{uhd}(u) \cdot v$
$\wedge\ \mathtt{udep}(u){:}\mathtt{NP} \wedge \mathtt{udep}(u) =_{\mathtt{agr}} n$
$\rightarrow n \mapsto^* n \cdot u \cdot v$

references: LXN (2.2), UDC (5.7), AGR (3.12), SENT (5.2),
LEX* (2.1), PSIG (5.9)

I make the simplifying assumption that there is only one wh-relative pronoun, wh:Lex. A new sort of UDC intensions, Wh, is introduced to encode wh-substitution (so the foot value for such intensions is wh). A wh-relative clause is like a sentence (possibly topicalised) in the first constituent of which one or more occurrences of an NP have been replaced by wh. The relative clause can modify an NP, possibly of different case, but sharing agreement features proper (number, person, and perhaps others). WHREL0 is a first attempt at abstract characterisation of what a treatment of wh-substitution must offer. If n is an NP intension (perhaps for *the shirt*), and u is a wh-intension with $\mathtt{udep}(u)$ an NP in (non-case) agreement with n (perhaps again for *the shirt*), and v is an intension (say of *I hate*) such that $\mathtt{uhd}(u) \cdot v$ (where say $\mathtt{uhd}(u)$ licenses *the colour of the shirt*) can form a sentence (here the topicalised sentence *The colour of the shirt, I hate*), then WHREL0 insists that from n we must be able to get $n \cdot u \cdot v$ (and hence n licenses the relativised noun phrase *the shirt the colour of which I hate*).

Some of the sorts of constituent (parse) structures commonly given to relative clauses (and allowed by WHREL0) are shown in Figure 5–4. WHREL0 ensures that

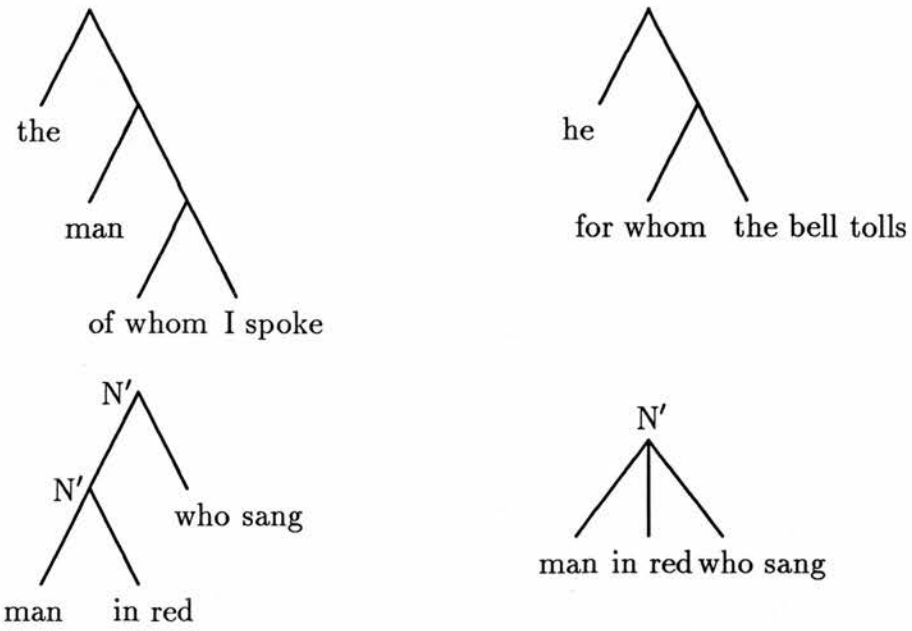

Figure 5–4: Constituency In A Relative Construction

we get the right realisations in Lex*, but does not insist on grouping together *man* and *of whom I spoke* in *the man of whom I spoke.* We can say something about this by having a sort N′. In an X-bar grammar, we would certainly expect this to consist literally of N′ intensions, but in a categorial treatment say, we would probably interpret it as the same as N. This is because N in a categorial treatment does the same job as N′ in (say) LFG. In LFG if r is in Lex, there is no x such that $r \mapsto x$; in categorial grammar, this need not be the case. Thus LFG cannot use N to represent both a lexical item and a phrase, but a categorial grammar can. WHREL0 also fails to insist on grouping *of whom* and *I spoke* together into the relative clause *of whom I spoke.* Thus WHREL0 allows the assignment of structures like Figure 5–5, which most linguists would consider inadequate. The specification WHREL, following, aims to address some of these deficiencies of WHREL0.

(5.23) NBAR = **sorts** N′,Syn
axiom N′ ⊆ Syn

WHREL = **enrich** WH+AGR+SENT+NP+ NBAR+UDC+GMR **by**
axiom $\forall n{:}\mathtt{N}', u{:}\mathtt{Wh}\,.\,\mathtt{uhd}(u){:}\mathtt{S}$
$\wedge\, \mathtt{udep}(u){:}\mathtt{NP} \wedge \mathtt{udep}(u) =_{\mathtt{agr}} n$

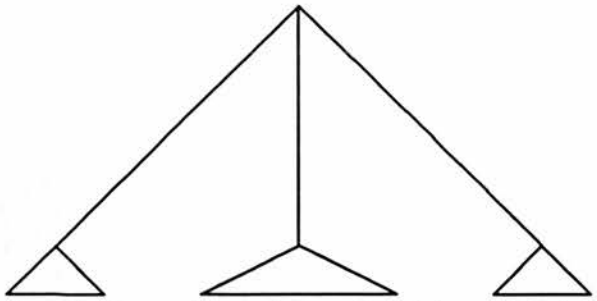

Figure 5–5: Inadequate structure allowed by WHREL0

$$\rightarrow n \mapsto n \cdot u \vee \forall w.(n \mapsto w \rightarrow n \mapsto w \cdot u)$$

$$\textbf{axiom } \forall u{:}\mathtt{Wh}\,.\,\mathtt{uhd}(u){:}\mathtt{S} \wedge \mathtt{udep}(u){:}\mathtt{NP} \wedge u \mapsto x$$

$$\rightarrow \exists v{:}\mathtt{Wh}, y{:}\mathtt{Syn*}\,.\,x = v \cdot y$$

$$\wedge\, \mathtt{uhd}(u) \mapsto \mathtt{uhd}(v) \cdot y$$

references: WH (5.21), AGR (3.12), SENT (5.2), NP (5.22),
UDC (5.7), GMR (2.4)

The disjunction in the first axiom allows for the two possible structures given to *man in red who sang* in Figure 5–4. So if u is a wh-intension whose dependent value $\mathtt{udep}(u)$ agrees with some intension $n{:}\mathtt{N'}$ (say of *man*), and the top of the canonical parse ($\mathtt{uhd}(u)$) is in S, then the first disjunct allows grammars which assign the left hand of the structures for *man in red who sang* in Figure 5–4, and the second disjunct allows grammars which assign the right hand of the two structures. Since the first axiom does have a constituent (u) for the relative clause, the second axiom is needed to ensure that wh-substitution only occurs in the first part of the sentence. This also loses us the possibility of wh-substitution in non-initial constituents of embedded sentences, but since most speakers seem to find examples of this (such as *That is the man, my belief that John killed whom, Mary destroyed*) unacceptable (or at least highly questionable), this is perhaps no real loss.

5.8 Implementation

Perhaps we can use the same strategy to add wh-relatives as was used in UTOP (5.18) to add topicalisation (as suggested on page 163). We could use a constructor wh(_,_) instead of _/_, with a suitable introduction rule. We can imagine applying this construction to models produced by UTOP, to get examples like *man who Mary likes* in which there is wh-substitution into a topic. A straightforward implementation of this strategy will fail to limit wh-substitution to the first constituent of a sentence (so we might get *This is the man Mary loves who*), or to prevent wh-substitution into slash-categories (*the woman John, who loves*).

This might be thought of as evidence that we ought to construct wh- and slash-categories simultaneously. We know that we will need slash categories when constructing wh-categories, in order to get wh-substitution into a topic (*man who Mary likes*). But if the whole construction is simultaneous, it will become difficult to prevent also allowing slash-substitution into a topic, which would allow, say *Cats, of, John has a fear.* In GKPS, this is ruled out by making SLASH a head feature, which in turn is part of the reason for the rather contorted form of the Head Feature Convention. We can avoid this by making the construction of wh-categories dependent on (rather than simultaneous with) the construction of slash-categories. So an implementation might use UIMP (5.19) to first create slash-intensions, and then add more structure to deal with wh-intensions, something like:

$$\mathsf{WIMP} \quad = \quad \lambda\mathcal{X}{:}\mathsf{WCOR}\,.\,\mathsf{UIMP}(\mathcal{X}) + \textit{ other subspecifications...}$$

Here we see the advantage of formal modularisation techniques.

This returns us, however, to the problem of ensuring that wh-substitution only occurs in the first constituent of a relative clause.[2] In constructing wh-categories,

---

[2]It may be that that this is too restrictive an assumption for dealing with arbitrary

we should not use slash-categories (they are only needed in defining $\mapsto$ on the wh-categories). This prevents "double extraction", which is enough to ensure that wh-substitution cannot occur in the comment of a topicalised sentence. In GKPS, the job is completed by also forbidding wh-substitution into a VP. We can easily do likewise, though it is worth noting a couple of possible disadvantages. One is that it is only good for extending grammars in which "NP VP" and "topic comment" are the only sentence forms — but this has been implicit all along. Another is that it prevents wh-substitution into embedded VPs, as in *That is the tyrant, my plan to kill whom backfired* — but once again, such examples seem to be at the limits of acceptability.

WTOP = **enrich**
derive from TOPIC
by [SlSyn $\longmapsto$ Syn, SlSyn* $\longmapsto$ Syn*]
by **sort** Syn
axiom SlSyn $\subseteq$ Syn

WCOR = **derive from** GMR+SENT+NP+NBAR+AGR+RELPRO
by [Cor $\longmapsto$ Syn, Cor* $\longmapsto$ Syn*]

WSYN = **derive from** USYN **by** [Wh $\longmapsto$ Slash, wh(_,_) $\longmapsto$ (_/_)]

WREL = **extend** WTOP+WCOR+WSYN+UDD+WH
by **pred** _ $\mapsto$ _ $\subseteq$ Syn $\times$ Syn*
axiom $\forall s{:}\mathtt{S}, n{:}\mathtt{NP}, n'{:}\mathtt{N}'\,.\, n =_{\mathtt{agr}} n' \rightarrow n' \mapsto n' \cdot \mathtt{wh}(s,n)$
axiom $\forall x{:}\mathtt{SlSyn*}\,.\, s \mapsto x \wedge s' = \mathtt{wh}(s,t) \wedge \mathtt{udd}_{t,\mathtt{wh}}(x,x')$
$\rightarrow s' \mapsto x'$

---

wh-phenomena. For instance, echo questions (e.g. *You saw who?*) may have wh-words inside a non-initial constituent of a sentence. But for wh-relatives, no such problem arises.

references: TOPIC (5.13), GMR (2.4), SENT (5.2), NP (5.22),
NBAR (5.23), AGR (3.12), RELPRO (5.20)

WCOR+WTOP is supposed to give the requirements of a core grammar to which WREL adds an account of wh-relatives. The sort name `SlSyn` is used to refer to the `Cor` plus the `Slash` intensions. This is necessary so that we can allow relative clauses to correspond to topicalised sentences (e.g. *who Mary loves* corresponds to *John, Mary loves*) in the axioms of WREL. WREL uses much the same strategy as UTOP. The first axiom is the introduction rule: if n' is an N′ intension, s is a sentential intension, and n is an NP agreeing with n', we get n' dominating $n' \cdot \mathtt{wh}(s, n)$, giving us the kind of local tree at the root of the leftmost of the two analyses given for *man in red who sang* in Figure 5–4. The second axiom then says that $\mathtt{wh}(s, t)$ may dominate a string x' differing in the way specified in UDD (5.17) from a string x, consisting of core and slash intensions, such that $s \mapsto x$. As with UIMP (5.19), a specification will be needed to pin down exactly when $\mathtt{wh}(\_, \_)$ is to be defined. A simple example might be

$$\begin{aligned} \mathsf{WXOK} \quad = \quad & \textbf{enrich WCOR+WTOP+WSYN} \\ & \textbf{by} \quad \textbf{axiom D}\, \mathtt{wh}(v, t) \leftrightarrow \\ & \qquad \neg \exists s{:}\mathtt{S}, r{:}\mathtt{SlSyn}, xy{:}\mathtt{SlSyn}{*}\,.\, s \mapsto r \cdot x \cdot v \cdot y \end{aligned}$$

This says that we may substitute into any core or slash intension which cannot appear as a non-initial constituent of a sentence. The reason for insisting on this is aimed at ensuring the second condition of WHREL: that we can only substitute into the first constituent of a sentence. Although this may look a formidable condition, a typical grammar for English which deals with topicalisation will only have sentences of the form NP-VP, or topic-comment. We already know comments cannot be wh-substituted into, so all WXOK then says is that VPs cannot be substituted into. We can form a simple parameterised implementation designed to add an account of relatives to an account of topicalisation.

$$\mathsf{WIMP} \quad = \quad \lambda \mathcal{X}{:}\mathsf{WCOR+WTOP}.\ \mathcal{X} + \mathsf{WREL} + \mathsf{WXOK}$$

Of course this can be composed with an implementation of topicalisation, such as UIMP (5.19), to produce an implementation which adds both topicalisation and relativisation.

$$\begin{aligned}\mathsf{WUIMP} \;=\; & \lambda\mathcal{X}{:}\mathsf{WCOR}.\ \mathsf{WIMP}(\\ & \quad \textbf{derive from } \mathsf{UIMP}(\mathcal{X}) \\ & \quad \textbf{by } [\mathtt{SlSyn} \longmapsto \mathtt{Syn}, \mathtt{SlSyn*} \longmapsto \mathtt{Syn*}])\end{aligned}$$

references: UIMP (5.19), TCOR (5.14)

5.9 Coda

The aim of this chapter has been to exemplify the use of stepwise refinement and modular techniques at both abstract and concrete levels, and their interplay. Thus we have seen a succession of attempts at abstract characterisation of the phenomenon of topicalisation, each motivated by deficiencies in its predecessor discovered by considering its implementation. This was followed by production of abstract and concrete specifications aimed at the related phenomenon of wh-relatives. The closeness of these phenomena allowed us to re-use for wh-relatives some of the specifications used for topicalisation. This sort of re-usability has been one of the major motivations behind the development of modular techniques.

Chapter 6

Germanic Word Order and Dependency Grammar

In this chapter I wish to exemplify the use of specification and refinement to talk about linguistic phenomena across a range of languages. I will consider (a subset of) dependent clause order in English, Dutch and German. I shall consider what vocabulary might be needed to describe the three cases, consider some simple examples of refining grammars, and describe a parameterised construction.

6.1 Dependent Clause Order

A dependent clause is an embedded clause like *that Michael saw Harold swim*, in *I believe that Michael saw Harold swim.* In English, the word order of the material following *that* is much the same as it would be in a main clause: *Michael saw Harold swim.* In German and Dutch, however, this is not so. For instance, in Dutch, we would have

dat	Michael	Harold	zag	zwemmen
that	Michael	Harold	saw	swim

for the dependent clause, but

Michael	zag	Harold	zwemmen
Michael	saw	Harold	swim

for the main clause. In these languages, it is often taken that the embedded order is canonical. For instance, in German one may describe main clause order by a combination of (main verb) *inversion*, and topicalisation (as for instance in Reape 1990). In Chapter 5, we have seen something of how one such process (topicalisation) may be dealt with by a modular extension (at least for the case of English). Thus dependent clause order is a reasonable choice for a Germanic "core" grammar, which may be built up by modular extension to broader coverage.

For the sake of simplicity, I will limit my consideration here to verbs which take nominal and verbal arguments only (so, for instance, I will not deal with *place*, as in *place the salt on the table*). In English we may indicate constituent structure by brackets:

(6.1) that [Michael$_1$ saw$_1$ [Harold$_2$ swim$_2$]$_1$].

(Here the numerical subscripts are meant to indicate arguments to the verb). Similarly in German:

(6.2) dass [Michael$_1$ [Harold$_2$ schwimmen$_2$]$_1$ sah$_1$].

Dutch is less obviously susceptible to a constituency treatment, because the dependencies (indicated by the subscripts) *cross* (Bresnan et al. 1982):

(6.3) dat Michael$_1$ Harold$_2$ zag$_1$ zwemmen$_2$.

However the less restrictive notion of *dependency* can still be useful, at the very least to provide a descriptive vocabulary.

6.2 Dependency Grammar

From this simple example it might be thought that we could describe the Dutch case by first saying that *zag* subcategorises directly for an NP and a VP, instead for an S, but if we replace *zwemmen* with the transitive verb *kussen* (to kiss), we see this will not help:

dat Michael_1 Harold_2 Maria_2 zag_1 kussen_2.

We could try to push the strategy even further, and say for instance that *zag* subcategorises for a verb v plus other intensions x such that $S \lhd v + x$. But it will be much simpler if we simply say that *zag* subcategorises for a verb, in this case *kussen*, which in turn subcategorises for a subject *Harold* and an object *Maria*. Then we might describe the Dutch order (in part) by saying that a verb v which is argument to some r must succeed r, but other arguments to r precede it. Having eliminated the S and VP level intensions, we may as well eliminate NP as well, simply saying that *zwem* (for example) subcategorises for a noun, and that a proper noun (such as *Harold*) need take no arguments, but a (singular) common noun (like *man*) subcategorises for a determiner (say *een*) which must come immediately to its left.

Thus we eliminate all phrasal intensions, and are left just with lexical entries. In a dependency grammar (see for instance Matthews 1981), we say that *Michael* and *zwemmen* (or rather the lexical entries — intensions — corresponding to the particular occurrences of those words) are *dependents* of *zag*, and *zag* is *head* to *Michael* and *zwemmen*.[1] Every intension is thus associated with a particular collection (multiset) of dependent intensions. (In dependency grammar it

---

[1]My terminology here is perhaps somewhat non-standard, in that I refer to the fully grounded intensions which stand for words as lexical entries, rather than some underspecified entities which must be filled in before use. In examples like IENT below, such underspecified entities will instead correspond to constructor functions.

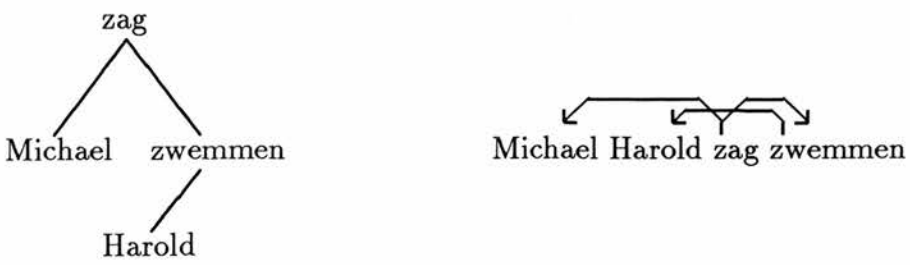

Figure 6–1: A dependency tree

is usual to describe adjuncts — like *quickly* in *Harold swam quickly* — as well as subcategorised-for arguments as dependents.) Since each dependent intension has its own associated multiset of dependents, any intension can be seen as the root of a tree, with its dependents forming the daughters, as in Figure 6–1. On the left, the dependencies are drawn in a familiar tree form, with mothers (heads) at the top of branches, and daughters (dependents) at the bottom. On the right, we see the conventional representation used in dependency grammar, called a *dependency diagram*, with arrows running from head to dependent. If s appears immediately under r in the tree on the left, it is a dependent. If it appears *somewhere* under r, it is called a *subordinate* of r. Subordinacy is the reflexive, transitive closure of dependency. (We could use the term *proper subordinacy* to refer to closure under transitivity alone — so r is not properly subordinate to r.) Write $r \gg s$ to mean s is subordinate to r.

Dependency is the same sort of relation which exists between a head daughter and its sisters in HPSG (the sisters depend on the head), except that instead of having separate intensions for a head daughter and mother, the intension associated with the head must itself, in some fashion, license some realisation(s) as string(s) of words. The existence of dependents already associates with any entry a particular multiset of words, namely the words for which its subordinates are entries; the collection formed from the word associated with the entry, plus the words associated with its dependents, plus the words associated with *their* dependents, and so on. For instance, consider the sentence *Michael saw Harold swim.* In a dependency grammar, this would be licensed by a lexical entry for the word *saw.* This entry would have as dependents entries for *Michael* and *swim.* The entry for

Michael has no dependents; that for *swim* has as a dependent an entry for *Harold* (which also has no dependents). Thus such an entry for *saw* is associated with the multiset of words $\{saw, Michael, swim, Harold\}$. Every realisation for such an entry must be a linearisation of this multiset. A description of realisation can thus be completed by specifying a precedence relation on intensions, much as in Section 3.4, except that the relation will be used in ordering, not just "sister" intensions, but clauses as a whole.

(Multisets over Ent)

MENT = **derive from** MIX

by [Ent $\longmapsto$ Syn, MEnt $\longmapsto$ MSyn, Ent* $\longmapsto$ Syn*]

(6.4) (Dependents)

DEP = **enrich** MENT **by**

sort Lex

opn lex : Ent $\to$ Lex

opn deps : Ent $\to$ MEnt

(6.5) (Subordinacy)

SUBORD = **extend** DEP **by**

pred _ $\gg$ _ $\subseteq$ Ent $\times$ Ent

axiom $r \gg r$

axiom deps$(r) = t + x \to r \gg t$

axiom $r \gg s \wedge s \gg t \to r \gg t$

(6.6) DLEX* = **extend** DEP **by**

opn lex : Ent* $\to$ Lex*

axiom lex(e) = e

axiom lex$(x \cdot y)$ = lex$(x) \cdot$ lex(y)

(6.7) (Realising Multiset)

MREAL = **extend** DEP **by**

opn $\texttt{mdep} : \texttt{MEnt} \to \texttt{MEnt}$

axiom $\forall r{:}\texttt{Ent} \,.\, \texttt{mdep}(r) = r + \texttt{mdep}(\texttt{deps}(r))$

axiom $\texttt{mdep}(0) = 0$

axiom $\texttt{mdep}(x + y) = \texttt{mdep}(x) + \texttt{mdep}(y)$

DLP = **derive from** LPOK **by** $[\texttt{Ent} \longmapsto \texttt{Syn}, \texttt{Ent}* \longmapsto \texttt{Syn}*]$

(6.8) (Dependency Realisations)

DREAL = **extend** DLEX* + MREAL + DLP **by**

pred $\texttt{rlsn} \subseteq \texttt{Ent} \times \texttt{Lex}*$

axiom $\texttt{mdep}(r) = |x| \wedge \texttt{lpok}(x) \to \texttt{rlsn}(r, \texttt{lex}(x))$

references: MIX (3.33), LPOK (3.24)

Ent is an intensional domain of lexical entries. MENT gives us multisets of entries, MEnt, using the construction MIX. 0 is the empty multiset, $\_ + \_$ is multiset union (where objects r of Ent are identified with singleton multisets $\{r\}$), and $|\_|$ projects strings over Ent (called Ent*) onto multisets. DEP associates with every entry r, the particular word $\texttt{lex}(r)$ for which it stands, and its multiset of dependent entries, $\texttt{deps}(r)$. SUBORD uses this to define subordinacy, by reflexive transitive closure. In DLEX*, the correspondence between an entry and a word given by lex is extended to map a string of entries to a string of words. By collecting r, plus its dependents, plus their dependents, and so on, MREAL defines the realising multiset of entries, $\texttt{mdep}(r)$. Every realising string of entries x for r must have $|x| = \texttt{mdep}(r)$. In DREAL we define a realisation of r to be any string of words corresponding to some x which satisfies both $|x| = \texttt{mdep}(r)$ and $\texttt{lpok}(x)$. The parameterised construction

DGMR = $\lambda\mathcal{X}{:}(\text{DEP} + \text{DLP}) \,.\, \mathcal{X} + \text{DREAL}$

will add realisations to a model describing dependency and linear precedence.

6.3 Implementation

In this section we will see how DREAL may be refined in different directions to produce grammar fragments for the three languages under consideration, beginning with English.

EWDS = **extend** $\emptyset$ **by**

opns $\mathtt{michael}, \mathtt{harold}, \mathtt{swim}, \mathtt{saw} :\rightarrow \mathtt{Lex}$

IENT = **extend** $\emptyset$ **by**

sorts Noun,Verb,Ent

axiom $\mathtt{Noun}, \mathtt{Verb} \subseteq \mathtt{Ent}$

opns $\mathtt{michael}_1, \mathtt{harold}_1 :\rightarrow \mathtt{Noun}$

opn $\mathtt{swim}_1 : \mathtt{Noun} \rightarrow \mathtt{Verb}$

opn $\mathtt{saw}_1 : \mathtt{Noun} \times \mathtt{Noun} \rightarrow \mathtt{Verb}$

opn $\mathtt{saw}_2 : \mathtt{Noun} \times \mathtt{Verb} \rightarrow \mathtt{Verb}$

IDEPS = **extend** MENT + IENT **by**

opn $\mathtt{deps} : \mathtt{Ent} \rightarrow \mathtt{MEnt}$

axiom $\mathtt{deps}(\mathtt{michael}_1) = \mathtt{deps}(\mathtt{harold}_1) = 0$

axiom $\mathtt{deps}(\mathtt{swim}_1(n)) = n$

axiom $\mathtt{deps}(\mathtt{saw}_1(n, n')) = n + n'$

axiom $\mathtt{deps}(\mathtt{saw}_2(n, v)) = n + v$

ISUBJ = **extend** IENT **by**

opns $\mathtt{subj} : \mathtt{Verb} \dot{\rightarrow} \mathtt{Noun}$

axiom $\mathtt{subj}(\mathtt{swim}_1(n)) = n$

axiom $\mathtt{subj}(\mathtt{saw}_1(n, n')) = n$

axiom $\mathtt{subj}(\mathtt{saw}_2(n, v)) = n$

ELEX = **extend** EWDS + IENT **by**

opn $\mathtt{lex} : \mathtt{Ent} \to \mathtt{Lex}$

axiom $\mathtt{lex}(\mathtt{michael}_1) = \mathtt{michael}$

axiom $\mathtt{lex}(\mathtt{harold}_1) = \mathtt{harold}$

axiom $\mathtt{lex}(\mathtt{swim}_1(n)) = \mathtt{swim}$

axiom $\mathtt{lex}(\mathtt{saw}_1(n, n')) = \mathtt{saw}$

axiom $\mathtt{lex}(\mathtt{saw}_2(n, v)) = \mathtt{saw}$

EWDS gives the words for this fragment. IENT uses operations to construct the needed intensions. So for instance Noun will contain the intensions $\mathtt{michael}_1$ and $\mathtt{harold}_1$, and there will be different intensions $\mathtt{swim}_1(\mathtt{michael}_1)$ and $\mathtt{swim}_1(\mathtt{harold}_1)$ corresponding (as we see in IDEPS) to the different values for the dependent noun. We will also need to be able to distinguish the subject noun from any others: this is taken care of in ISUBJ. ELEX takes care of associating a particular (English) lexical item with every intension. IDEPS+ELEX specifies, up to isomorphism, the model of DEP which has

(6.9) $[\mathtt{Lex}] = \{michael, harold, swim, saw\}$

$[\mathtt{Ent}] = [\mathtt{Noun}] \cup [\mathtt{Verb}]$

$[\mathtt{Noun}] = \{michael_1, harold_1\}$

$[\mathtt{lex}](michael_1) = michael$, $[\mathtt{deps}](michael_1) = \emptyset$

$[\mathtt{lex}](harold_1) = harold$, $[\mathtt{deps}](harold_1) = \emptyset$

$[\mathtt{Verb}]$ contains $swim_1(n)$ and $saw_1(n, n')$ for each $n, n' \in [\mathtt{Noun}]$,

plus, recursively, $saw_2(n, v)$ for each $n \in [\mathtt{Noun}], v \in [\mathtt{Verb}]$

$[\mathtt{lex}](swim_1(n)) = swim$, $[\mathtt{deps}](swim_1(n)) = \{n\}$, all $n \in [\mathtt{Noun}]$,

$[\mathtt{lex}](saw_1(n, n')) = saw$, $[\mathtt{deps}](saw_1(n, n')) = \{n, n'\}$, all $n, n' \in [\mathtt{Noun}]$,

$[\mathtt{lex}](saw_2(n, v)) = saw$, $[\mathtt{deps}](saw_2(n, v)) = \{n, v\}$,

each $n \in [\mathtt{Noun}], v \in [\mathtt{Verb}]$.

Thus IDEPS+ELEX+MREAL identifies, up to isomorphism, the model which has in addition

$$[\mathtt{mdep}](michael_1) = \{michael_1\},\ [\mathtt{mdep}](harold_1) = \{harold_1\},$$

$$[\texttt{mdep}](swim_1(n)) = \{swim_1(n), n\}, \text{ each } n \in [\texttt{Noun}],$$
$$[\texttt{mdep}](saw_1(n,n')) = \{saw_1(n,n'), n, n'\}, \text{ each } n, n' \in [\texttt{Noun}],$$
$$[\texttt{mdep}](saw_2(n,v)) = \{saw_2(n,v), n\} \cup [\texttt{mdep}](v),$$
$$\text{each } n \in [\texttt{Noun}], v \in [\texttt{Verb}].$$

We now proceed to specify linear precedence.

$$\mathsf{DLPBIN} \;=\; \textbf{derive from LPBIN by } [\texttt{Ent} \longmapsto \texttt{Syn}, \texttt{Ent*} \longmapsto \texttt{Syn*}]$$

$$\begin{aligned}\mathsf{EPREC} \;=\; &\textbf{enrich IDEPS + ISUBJ + SUBORD + DLPBIN by}\\ &\quad\textbf{axiom } \neg\texttt{lpok}(r \cdot s) \leftrightarrow\\ &\qquad(\exists v{:}\texttt{Verb}, n{:}\texttt{Noun}, t{:}\texttt{Ent}, x{:}\texttt{MEnt}\,.\\ &\qquad\quad \texttt{deps}(t) = v + n + x \wedge (v \gg r) \wedge (n \gg s))\\ &\qquad\vee (\exists v{:}\texttt{Verb}, n{:}\texttt{Noun}\,.\, n = \texttt{subj}(v) \wedge (n \gg s)\\ &\qquad\quad \wedge (v \gg r) \wedge \neg(n \gg r))\\ &\qquad\vee (s{:}\texttt{Verb} \wedge (s \gg r) \wedge s \neq r \wedge \neg(\texttt{subj}(s) \gg r))\end{aligned}$$

The axiom used in EPREC to give linear precedence is rather stronger than is really needed in this simple example, the idea being that conditions of this form will also do the job in more complicated examples (though then more disjuncts would be needed as well). The first disjunct says that where both a noun and a verb occur as direct dependents of some intension, the noun and all its dependents come before the verb and all of its dependents. The second says that the subject of a verb, and all its successive dependents, must precede the verb, and all its (other) successive dependents. The last disjunct says that eventual dependents of the verb which are *not* eventual dependents of the subject must follow the verb.[2]

---

[2]In order to allow a verb phrase argument to a verb v (perhaps *promise*), we could give v a verb as its only dependent. The subject of v would be stipulated to be the same as that of the dependent verb, but not a *direct* dependent.

ELEX + EPREC specifies, up to isomorphism, a single model of dependency and precedence for this tiny fragment of English, and DGMR(ELEX + EPREC) is an implementation giving realisations such as

$$\begin{aligned}&\mathtt{rlsn}(\mathtt{saw}_2(\mathtt{harold}_1, \mathtt{saw}_1(\mathtt{michael}_1, \mathtt{harold}_1)),\\ &\qquad \mathtt{harold} \cdot \mathtt{saw} \cdot \mathtt{michael} \cdot \mathtt{saw} \cdot \mathtt{harold}).\end{aligned}$$

Let us now turn to an example in German.

$$\begin{aligned}\mathsf{GLEX} \quad &= \quad \textbf{derive from ELEX}\\ &\qquad \textbf{by } [\mathtt{schwimmen} \longmapsto \mathtt{swim}, \mathtt{sah} \longmapsto \mathtt{saw}]\end{aligned}$$

$$\begin{aligned}\mathsf{GPREC} \quad &= \quad \textbf{enrich IDEPS + SUBORD + DLPBIN by}\\ &\qquad \textbf{axiom } \neg\mathtt{lpok}(r \cdot s) \leftrightarrow\\ &\qquad\quad (\exists v{:}\mathtt{Verb}, n{:}\mathtt{Noun}, t{:}\mathtt{Ent}, x{:}\mathtt{MEnt}\,.\\ &\qquad\qquad \mathtt{deps}(t) = v + n + x \wedge (v \gg r) \wedge (n \gg s))\\ &\qquad\quad \vee\, (r{:}\mathtt{Verb}\,.\, r \gg s \wedge r \neq s)\end{aligned}$$

The fact that we are able to use for German, and later Dutch, the same intensional system IDEPS used for English, is partly a reflection of the closeness of the languages (though of course it also has a lot to do with the triviality of the example). However we need a different precedence relation, and of course the actual words are different. The first disjunct of the axiom of GPREC is the same as for EPREC. The second says that a verb succeeds all its dependents. GLEX + GPREC specifies, up to isomorphism, a single model of dependency and precedence for this fragment of German, and DGMR(GLEX + GPREC) is an implementation giving realisations such as

$$\mathtt{rlsn}(\mathtt{saw}_2(\mathtt{harold}_1, \mathtt{swim}_1(\mathtt{michael}_1)), \mathtt{harold} \cdot \mathtt{michael} \cdot \mathtt{schwimmen} \cdot \mathtt{sah}).$$

Now for the Dutch case.

$$\begin{aligned}\mathsf{NLEX} \quad &= \quad \textbf{derive from ELEX}\\ &\qquad \textbf{by } [\mathtt{zwemmen} \longmapsto \mathtt{swim}, \mathtt{zag} \longmapsto \mathtt{saw}]\end{aligned}$$

$$
\begin{aligned}
\mathsf{NPREC} \quad = \quad & \textbf{enrich } \mathsf{IDEPS} + \mathsf{ISUBJ} + \mathsf{SUBORD} + \mathsf{DLPBIN} \textbf{ by} \\
& \quad \textbf{axiom } \neg \mathtt{lpok}(r \cdot s) \leftrightarrow \\
& \qquad (\exists v{:}\mathtt{Verb}, n{:}\mathtt{Noun}, t{:}\mathtt{Ent}, x{:}\mathtt{MEnt}\,. \\
& \qquad\quad \mathtt{deps}(t) = v + n + x \wedge (v \gg r) \wedge (n \gg s)) \\
& \qquad \vee (\exists v{:}\mathtt{Verb}, n{:}\mathtt{Noun}\,. n = \mathtt{subj}(v) \wedge (n \gg s) \\
& \qquad\quad \wedge (v \gg r) \wedge \neg(n \gg r)) \\
& \qquad \vee (\exists x{:}\mathtt{MEnt}\,. r{:}\mathtt{Verb} \wedge \mathtt{deps}(s) = r + x)
\end{aligned}
$$

The first two disjuncts of the axiom are the same as for EPREC. The last says that dependent verbs follow their head. NLEX + NPREC specifies, up to isomorphism, a single model of dependency and precedence for this fragment of Dutch, and DGMR(NLEX + NPREC) is an implementation giving realisations such as

$$\mathtt{rlsn}(\mathtt{saw}_2(\mathtt{harold}_1, \mathtt{swim}_1(\mathtt{michael}_1)), \mathtt{harold} \cdot \mathtt{michael} \cdot \mathtt{zag} \cdot \mathtt{zwemmen}).$$

We might produce a more heavily parameterised construction which abstracts over these examples:

(A simple Germanic core dependency treatment)
$\mathsf{GMNC} \quad = \quad \lambda\mathcal{X}{:}\mathsf{DLPBIN}\,.\,\mathsf{DGMR}(\mathcal{X} + \mathsf{ELEX})$

The model specified by DGMR(ELEX+EPREC) is also specified by GMNC(EPREC). The model specified by DGMR(GLEX + GPREC) can also be specified by a simple relabelling in GMNC(GPREC):

derive from GMNC(GPREC) **by** $[\mathtt{schwimmen} \longmapsto \mathtt{swim}, \mathtt{sah} \longmapsto \mathtt{saw}]$.

The model specified by DGMR(NLEX+NPREC) can be specified by relabelling in GMNC(NPREC):

derive from GMNC(NPREC) **by** $[\mathtt{zwemmen} \longmapsto \mathtt{swim}, \mathtt{zag} \longmapsto \mathtt{saw}]$

6.4 Dependency Constituents

What allows us to assign constituent structures in (6.1) and (6.2), is the fact that in English and German (at least in these fragments), all words subordinate to any particular word must appear contiguously. We can express this by insisting that treatments of English and German must refine the following specification:

(Adjacency Condition)
ADJCND = **enrich** SUBORD + DLP **by**
axiom $\texttt{lpok}(w \cdot r \cdot x \cdot s \cdot y \cdot t \cdot z) \wedge r \gg t \rightarrow r \gg s$
axiom $\texttt{lpok}(w \cdot t \cdot x \cdot s \cdot y \cdot r \cdot z) \wedge r \gg t \rightarrow r \gg s$

This just says that, in an LP-acceptable sequence, if t is subordinate to r, and s occurs between them, then s must also be subordinate to r. For any grammar refining ADJCND, we can make a definition of dependency constituents as follows:

DCIN = **extend sorts** Lex,Ent **by**
sort Syn
opn phr : Ent $\rightarrow$ Syn
axiom Lex, Ent $\subseteq$ Syn

MPHR = **extend** DCIN + MENT + MSYN **by**
opn phr : MEnt $\rightarrow$ MSyn
axiom $\texttt{phr}(0) = 0$
axiom $\texttt{phr}(x + y) = \texttt{phr}(x) + \texttt{phr}(y)$

DCIMD = **extend** DEP + MPHR **by**
pred _ $\lhd$ _ $\subseteq$ Syn $\times$ MSyn
axiom $\texttt{phr}(r) \lhd r + \texttt{phr}(\texttt{deps}(r))$
axiom $r \lhd \texttt{lex}(r)$

DCIN admits models in which every object of **Syn** is either a word (comes from **Lex**), a lexical entry (comes from **Ent**), or is the phrasal projection of a lexical entry (is of the form $\mathtt{phr}(r)$, for r in **Ent**). In MPHR, **phr** is extended to map multisets of lexical entries to the corresponding multisets of projections. In DCIMD we define a notion of dominance in which a projection dominates the multiset consisting of the lexical head, plus the projections of its dependents. The lexical head, of course, dominates the corresponding lexical item (i.e. word). Producing an extension of linear precedence to projections in the general case is possible but messy. Instead I give a definition which works when precedence on **Ent** only depends on which pairs are allowed.

$$\begin{aligned}\mathsf{DCLP} \quad = \quad & \textbf{enrich } \mathsf{DLPBIN} + \mathsf{LPBIN} + \mathsf{DCIN} + \mathsf{SYN}* \textbf{ by} \\ & \textbf{axiom } \forall st{:}\mathtt{Ent}\,.\,\mathtt{lpok}(s \cdot t) \leftrightarrow \mathtt{lpok}(s \cdot \mathtt{phr}(t)) \\ & \qquad \leftrightarrow \mathtt{lpok}(\mathtt{phr}(s) \cdot t) \leftrightarrow \mathtt{lpok}(\mathtt{phr}(s) \cdot \mathtt{phr}(t))\end{aligned}$$

$$\begin{aligned}\mathsf{WEQV} \quad = \quad & \textbf{enrich } \mathsf{DCIMD}{+}\mathsf{DCLP}{+}\mathsf{DREAL}{+}\mathsf{IDLP} \textbf{ by} \\ & \textbf{axiom } \forall r{:}\mathtt{Ent}, x{:}\mathtt{Lex}*\,.\,\mathtt{rlsn}(r,x) \leftrightarrow r \mapsto x\end{aligned}$$

references: LPBIN (3.28), SYN* (2.3), IDLP (3.34)

DCLP ensures that the extension of **lpok** from **Ent*** to **Syn*** allows pairs of projections wherever the corresponding pair of lexical entries was allowed. If, for some implementation G, $\mathsf{G} \sqsubseteq \mathsf{DEP} + \mathsf{DLPBIN} + \mathsf{ADJCND}$ (as, for instance, with the above implementations ELEX + EPREC and GLEX + GPREC of English and German), then $\mathsf{G}{+}\mathsf{DCIMD}{+}\mathsf{DCLP}{+}\mathsf{DREAL}{+}\mathsf{IDLP} \sqsubseteq \mathsf{WEQV}$: that is, the construction of DCIMD + DCLP gives the same realisations as the dependency treatment. But if $\mathsf{G} \not\sqsubseteq \mathsf{ADJCND}$ (as with the Dutch implementation NLEX + NPREC), then $\mathsf{G} + \mathsf{DCIMD} + \mathsf{DCLP} + \mathsf{DREAL} + \mathsf{IDLP} \not\sqsubseteq \mathsf{WEQV}$, and the construction of DCIMD and DCLP does not produce an account of the same data. For instance, in Dutch we would end up incorrectly allowing the string *Michael zag Harold zwemmen.*

6.5 Coda

The purpose of this chapter has been to exemplify the use of abstract specifications in a cross-linguistic application. We have produced a simple parameterised construction which need only be supplied a description of precedence to give (very simplistic) treatments for dependent clause order in English, Dutch, or German. Such treatments ought then to be susceptible to modular extension in order to extend coverage, say to main clauses. The treatment of topicalisation presented in Chapter 5 requires a notion of constituency. For the English and German cases, a notion of constituency does arise naturally out of the preceding dependency treatments, which would allow UIMP (5.19) to apply to them with little extra effort. Any imposition of constituency on the Dutch, however, is liable to be somewhat unsatisfactory. Of course, the basis of UIMP (5.19) in constituency is a reflection of the fact that it was from that starting point that the deliberations of Chapter 5 began. This suggests that reconsidering the construction in terms of dependency could be fruitful.

Chapter 7

Conclusions

The aim of this thesis has been to explore the use of various ideas taken from algebraic software specification in describing systems of models for human language (grammar). The most important of these ideas are loose specification and stepwise refinement, and modularisation and parameterisation.

In Chapter 1 I set out the background of the use of specifications in software engineering, and suggested some possible benefits one might obtain from the use of such disciplines in the description of grammar. I gave an introduction to language ASL and the algebraic concepts underlying it. Chapter 2 was devoted to the basics of models and specifications for constituent-style grammar. In Chapter 3 I moved on to consider how a richer intensional domain can be used to deal with matters such as agreement. I sketched how the model systems of PATR-II, LFG, GPSG and HPSG might be described by specifications in ordinary first order logic. Chapter 4 dealt with expanding our specification language to allow specifications written in different logics to be compared and combined. Chapter 5 contains the most detailed examples employing the range of techniques under consideration. I addressed the problem of topicalisation by developing a series of loose specifications which aim to capture the abstract idea of what it means to deal with topicalisation. This succession was informed by the experience of developing at each stage a parameterised implementation, capable of extending a core treatment by adding machinery to deal with topicalisation. The development of similar specifications to deal with wh-relative clauses was made much easier by the re-use of

modules developed when dealing with the related phenomenon of topicalisation. In Chapter 6 I considered how loose specification and parameterisation may be employed in developing a cross-linguistic treatment for some phenomenon — in this case, dependent clause order in English, German and Dutch. This involved specification of a grammar in the dependency tradition.

I believe that the central ideas under investigation — loose specification, stepwise refinement, modularisation, parameterisation — are appropriate to the linguistic domain, and capture informal ideas about methodology embodied, for instance, in the Principles and Parameters paradigm. I believe the use of precisely defined models is important in any science, and that ASL-style specifications allow us to avoid some of the pitfalls involved in such definitions, by giving us bookkeeping methods for breaking the task down into manageable chunks, in such a way that the independent pieces can be fitted together in different ways as our specification of a system evolves.

In retrospect, however, the choice of specification language used (ASL) may not have been ideal. A higher-level language may have been more appropriate in producing less cluttered specifications, although I am not very sure exactly what choice would have been more appropriate at the time. One possibility now is "Extended Prolog", as in Read and Kazmierczak (1992), a wide spectrum language aimed at the programming language of Sannella and Wallen (1987).

There is a fair bit of technical detail to be absorbed before one can read these specifications, and it takes some time and effort before they become sufficiently familiar to be really useful. While I feel that the introduction of a formal language for communicating these sort of ideas in important, it seems likely that some compromise is needed, perhaps involving a more specialised, high-level specification language, and a simplified presentation of those technical details actually required in using the specifications. One avenue toward a more specialised language might be the development of a better feature-value institution. Some consideration needs to be given to the relationship between the sort of presentation of feature-value logic in Section 4.3, and the employment of a calculus of minimal-model genera-

tion. Another (challenging) direction would be to consider what classes of models could be employed in an institution where GB-style models were to be initial.

Appendix A

Bibliography

Backus, J. (1978) Can Programming be liberated from the von Neumann Style? *Communications of the ACM* **21**, 613–641.

Bresnan, J., ed. (1982) *The Mental Representation of Grammatical Relations.* Cambridge, Mass.: MIT Press.

Bresnan, J., R. M. Kaplan, S. Peters and A. Zaenen (1982) Cross-serial dependencies in Dutch. *Linguistic Inquiry* **13**, 613–635.

Chomsky, N. (1981) *Lectures on Government and Binding.* Dordrecht: Foris Publications.

Chomsky, N. (1986) *Knowledge of Language: Its Nature, Origin and Use.* New York: Praeger.

Colmerauer, A., H. Kanoui, R. Pasero and P. Roussel (1973) Un système de communication homme-machine en français. Rapport. Groupe d'Intelligence Artificielle, Université d'Aix-Marseille II.

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

Gazdar, G., E. Klein, G. Pullum and I. Sag (1985) *Generalized Phrase Structure Grammar.* London: Basil Blackwell.

Goguen, J. A. (1987) One, None, a Hundred Thousand Specification Languages. Report CSLI-87-96, Centre for the Study of Language and Information, Stanford, Ca.

Goguen, J. A. and R. M. Burstall (1985) Institutions: Abstract Model Theory for Computer Science. Report CSLI-85-30, Centre for the Study of Language and Information, Stanford, Ca.

Goguen, J. A. and R. M. Burstall (1986) A Study in the Foundations of Programming Methodology: Specifications, Institutions, Charters and Parchments. Report ECS-LFCS-86-10, Laboratory for the Foundations of Computer Science, Department of Computer Science, University of Edinburgh.

Goguen, J. A. and J. Meseguer (1987a) Order-Sorted Algebra Solves the Constructor-Selector, Multiple Representation and Coercion Problems. Report CSLI-87-92, Centre for the Study of Language and Information, Stanford, Ca.

Goguen, J. A. and J. Meseguer (1987b) Order-Sorted Algebra I: Partial and Overloaded Operators, Errors and Inheritance. Technical Report. Computer Science Laboratory, SRI International, Menlo Park, Ca.

Goguen, J. A., J. W. Thatcher and E. G. Wagner (1978) An Initial Algebra Approach to the Specification Correctness and Implementation of Abstract Data Types. In R. Yeh, ed., *Current Trends in Programming Methodology*, Vol. 4: *Data Structuring*, pp. 80–149. Englewood Cliffs, N.J.: Prentice-Hall.

Goguen, J. A., J. W. Thatcher, E. G. Wagner and J. B. Wright (1975) Abstract Data Types as Initial Algebras and the Correctness of Data Representations. In *Proceedings of the Conference on Computer Graphics, Pattern Recognition, and Data Structures*, pp. 89–93, Beverly Hills, Ca.

Gries, D. (1981) *The Science of Programming.* Berlin: Springer-Verlag.

Hopkin, D. and B. Moss (1976) *Automata.* London: Macmillan.

Jackendoff, R. S. (1977) *X-Bar Syntax: A Study of Phrase Structure.* Cambridge, Mass.: MIT Press.

Janssen, T. M. V. (1983) Foundations and Applications of Montague Grammar. Ph.D. thesis, Mathematisch Centrum, Universiteit van Amsterdam.

Kaplan, R. and A. Zaenen (1987) Long-distance Dependencies, Constituent Structure and Functional Uncertainty. Unpublished ms.

Kowalski, R. (1974) Predicate logic as a Programming Language. In *Proceedings of the IFIP*, pp. 569–574, Amsterdam.

Kowalski, R. (1979) Algorithm = Logic + Control. *Communications of the ACM* **22**, 424–436.

Lambek, J. (1958) The mathematics of sentence structure. *American Mathematical Monthly* **65**, 154–170.

Lloyd, J. W. (1984) *Foundations of Logic Programming.* Berlin: Springer-Verlag.

Maling, J. M. and A. Zaenen (1982) A phrase structure account of Scandinavian extraction phenomena. In P. Jacobson and G. K. Pullum, eds., *The Nature of Syntactic Representation*, pp. 229–282. Dordrecht: D. Reidel.

Martin-Löf, P. (1982) Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science*, Vol. 6: *Proceedings of the Sixth International Congress of Logic, Methodology and Philosophy of Science*, pp. 153–175. Amsterdam: North Holland.

Matthews, P. H. (1981) *Syntax.* Cambridge: Cambridge University Press.

Montague, R. (1970) Universal grammar. *Theoria* **36**, 373–398. Reprinted in R. H. Thomason, ed., *Formal Philosophy: Selected Papers of Richard Montague*, pp. 222–246. New Haven, Conn.: Yale University Press, 1974.

Montague, R. (1973) The proper treatment of quantification in ordinary English. In J. Hintikka, J. M. E. Moravcsik and P. Suppes, eds., *Approaches to Natural Language.* Dordrecht: D. Reidel. Reprinted in R. H. Thomason, ed., *Formal*

Philosophy: Selected Papers of Richard Montague, pp. 247–270. New Haven, Conn.: Yale University Press, 1974.

Paxton, B. (1992) The Implementation of a Modular Prolog System Based on Standard ML Modules. 4th Year Project Report. Departments of Artificial Intelligence and Computer Science, University of Edinburgh.

Pereira, F. C. N. and S. M. Shieber (1984) The Semantics of Grammar Formalisms Seen as Computer Languages. In *Proceedings of the 10th International Conference on Computational Linguistics and the 22nd Annual Meeting of the Association for Computational Linguistics*, pp. 123–129, Stanford University, Stanford, Ca.

Pierce, B. C. (1990) A Taste of Category Theory for Computer Scientists. Technical Report CMU-CS-90-113R, Computer Science Department, Carnegie Mellon University, Pittsburgh.

Pollard, C. and I. A. Sag (1987) *Information-Based Syntax and Semantics*, Vol. 1: *Fundamentals.* Stanford, Ca.: Centre for the Study of Language and Information.

Pollard, C. and I. A. Sag (1992) *Information-Based Syntax and Semantics*, Vol. 2: *Topics in Binding and Control.* Stanford, Ca.: Centre for the Study of Language and Information. To appear.

Radford, A. (1988) *Transformational Grammar: A First Course.* Cambridge: Cambridge University Press.

Read, M. and E. Kazmierczak (1992) Formal program development in modular Prolog: a case study. In *Proceedings of a Workshop on Logic Program Synthesis and Transformation.* Berlin: Springer-Verlag. To appear.

Reape, M. (1990) A Theory of Word Order and Discontinuous Constituency in West Continental Germanic. In E. Engdahl and M. Reape, eds., *Parametric Variation in Germanic and Romance: Preliminary Investigations*, pp. 25–39. Edinburgh: Centre for Cognitive Science. Report R1.1.A of DYANA, Esprit Basic Research Action 3175.

Reape, M. and E. Engdahl (1990) Parametric Variation as a Research Strategy. In E. Engdahl and M. Reape, eds., *Parametric Variation in Germanic and Romance: Preliminary Investigations*, pp. 1–7. Edinburgh: Centre for Cognitive Science. Report R1.1.A of DYANA, Esprit Basic Research Action 3175.

Rizzi, L. (1982) *Issues in Italian Syntax.* Dordrecht: Foris Publications.

Robinson, J. A. (1965) A Machine-oriented Logic Based on the Resolution Principle. *Journal of the ACM* **12**, 23–41.

Rogers, H. (1967) *Theory of Recursive Functions and Effective Computability.* New York: McGraw Hill.

Sannella, D. (1986) Formal Specification of ML Programs. Report ECS-LFCS-86-15, Laboratory for the Foundations of Computer Science, Department of Computer Science, University of Edinburgh.

Sannella, D. (1989) Formal program development in Extended ML for the working programmer. Report ECS-LFCS-89-102, Laboratory for the Foundations of Computer Science, Department of Computer Science, University of Edinburgh.

Sannella, D. and A. Tarlecki (1985) Specifications in an Arbitrary Institution. Report CSR-184-85, Department of Computer Science, University of Edinburgh.

Sannella, D. and A. Tarlecki (1986) Extended ML: an institution-independent framework for formal program development. Report ECS-LFCS-86-16, Laboratory for the Foundations of Computer Science, Department of Computer Science, University of Edinburgh.

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

Sannella, D. and A. Tarlecki (1988) Toward Formal Development of Programs from Algebraic Specifications: Implementations Revisited. *Acta Informatica* **25**, 233–281.

Sannella, D. and A. Tarlecki (1992) Toward Formal Development of Programs from Algebraic Specifications: Model-Theoretic Foundations. Report ECS-LFCS-92-204, Laboratory for the Foundations of Computer Science, Department of Computer Science, University of Edinburgh.

Sannella, D. and L. A. Wallen (1987) A Calculus for the Construction of Modular Prolog Programs. In *IEEE 4th Symposium on Logic Programming*, San Francisco.

Sannella, D. and M. Wirsing (1983) A kernel language for algebraic specification and implementation. Report CSR-131-83, Department of Computer Science, University of Edinburgh.

Sells, P. (1985) *Lectures on Contemporary Syntactic Theories.* Stanford, Ca.: Centre for the Study of Language and Information.

Shieber, S. M. (1986) *An Introduction to Unification-based Approaches to Grammar.* Stanford, Ca.: Centre for the Study of Language and Information.

Stoy, J. E. (1977) *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* Cambridge, Mass.: MIT Press.

Tarlecki, A. (1984) Quasi-Varieties in Abstract Algebraic Institutions. Report CSR-173-84, Department of Computer Science, University of Edinburgh.