Institutional Approaches to Programming Language Specification

A Thesis submitted for the degree of PhD.

James Power BSc. MSc.
School of Computer Applications
Dublin City University

August, 1994.

Supervisor: Prof. Tony Moynihan

This thesis is based on the candidate's own work, and has not previously been submitted for a degree at any academic institution.

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of PhD. is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

James Power

February 9, 1995

Contents

Institutional Approaches to Programming Language Specification

James Power

Abstract

Formal specification has become increasingly important in software engineering, both as a design tool, and as a basis for verified software design. Formal methods have long been in use in the field of programming language design and implementation, and many formalisms, in both the syntactic and semantic domains, have evolved for this purpose.

In this thesis we examine the possibilities of integrating specifications written in different formalisms used in the description of programming languages within a single framework. We suggest that the theory of institutions provides a suitable background for such integration, and we develop descriptions of several formalisms within this framework. While we do not merge the formalisms themselves, we see that it is possible to relate modules from specifications in each of them, and this is demonstrated in a small example.

Acknowledgements

I want to thank Tony for being an ideal supervisor! Without his enthusiasm and encouragement over the last four years, it is extremely unlikely that I would ever have reached this stage.

All the staff in the School of Computer Applications have contributed to the production of this thesis by their interest and support; particularly I would like to thank Alan Smeaton (for getting me here in the first place) and, of course, (Dr.) John Murphy and (Dr.) John Waldron.

This document was prepared using Leslie Lamport's LaTeX document preparation system. The mathematical symbols and Z-style schema boxes were produced using Paul King's `oz.sty` style file, while the commutative diagrams were constructed with Paul Taylor's `diagrams.tex` package (version 3.8).

Chapter 1

Introduction

In this chapter we seek to motivate the work contained in this thesis, and to explain some of its background in the field of programming language design and implementation.

1.1 The Structure of a Programming Language

Broadly speaking, the description of programming languages can fall into the main categories used in linguistics for natural languages; specifically, we can speak of its:

- *syntax*, or the symbols used to denote specific concepts, and the correct grammatical form for their usage
- *semantics*, which is the method of assigning a meaning to some given text from the language

One of the simplest ways of providing a definition of a programming language's syntax and semantics is to write a compiler for it. This functions as a definition in two ways:

- it is a *recogniser* for the language: it will tell us which input programs belong to the language and which don't
- it gives a meaning to any correct program by translating it into some other language (such as assembly language)

In operational terms, these are often referred to as the *analytical* and *generative* phases of the compiler respectively.

Obviously the expectations which are nowadays associated with software engineering projects of any significance would suggest that some further description be given in addition to the final version of the working program. However, it will serve our purpose to consider a programming language definition in terms of the computations involved in its implementation.

Broadly speaking, compilation can be divided into four phases:

1. *Lexical analysis*, or scanning, which determines if the correct symbols have been used in the input; some basic grouping may occur here such as the formation of words from these symbols. Irrelevances such as whitespace and comments are usually removed at this stage.

2. *Syntax analysis*, or parsing, checks to see that the words identified in the input have been put together in the correct sequence. Such checking is free of global context, in that a phrase is checked only with reference to its immediate neighbours. Often this process proceeds iteratively from the output of the scanning phase, by grouping together larger and larger phrases until the entire input is structured hierarchically.

3. *Static semantics*, which involves conducting the remaining analysis operations on the source code which cannot be handled by the formalisms used for syntax analysis. Typically this phase will include checking details such as scope rules, type consistencies etc.

4. *Dynamic semantics*, or code generation, defines the (now fully-analysed) program in terms of some other formal system; for a compiler this will be some form of intermediate, object or target code.

These divisions are not, of course, absolute; for example

- many systems do not bother to differentiate between lexical and syntax analysis

- more advanced formalisms may incorporate some semantic operations (such as symbol-table maintenance) into the syntax definition

- often the static and dynamic semantics are merged (this turns the latter into a *partial* operation over syntactically well-formed sentences, instead of a *total* operation over semantically correct programs).

- The dynamic semantics may themselves be broken into several phases in order to facilitate optimisation, retargetting etc.

Even the division between syntax and semantics may be made less distinct by the presence of *ambiguity*, where some aspects of the syntactic analysis depend on semantic information.

The division of the process will depend on various factors such as the complexity of the language, the purpose of the definition and, not least, the nature of the formal specification method used (often itself a product of the intended use). Indeed, it is fair to say that the above division owes much to the (empirically justifiable) assumption of a context-free-grammar formalism for describing the syntax of a language.

Standard texts which describe the above phases in detail include [ASU86] and [PP92].

1.2 Specification of Programming Languages

Writing a compiler for a programming language is basically just another software engineering process, and may be expected to benefit from the techniques commonly associated with this field. A well-established practice in this area is the formulation of a formal specification prior to, and as a basis for, program implementation.

The formal specification of a compiler is what is more usually regarded as the definition of a particular programming language. More specifically, definitions are usually developed at a level of abstraction which is independent of a particular compiler implementation; maximum abstraction is desirable for a general-purpose language whose designers wish to encourage implementation on many platforms.

Many of the reasons for requiring a *formal* definition of a programming language are elaborations of the general case:

- Even if implementation is not directly considered, the formal specification of a project can be seen as worthy in itself, as it allows for the formal consideration of design decisions and the full exploration of the implications of the definition. The abstraction of implementation details allows the language designer to work in an environment more likely to lead to improvements in the nature of programming languages themselves, as opposed to just efficiency of operation.

- The automation of the program derivation process in general is highly desirable, as much of this work is detailed and error-prone. In a restricted domain of application such as programming languages it is reasonable to assume that the scope for such automation should be increased. Indeed, the compiler-generation tools lex and yacc [LMB92], and their many variants, provide some degree of functionality in this area. Attempts at automatically generating complete compilers have had various degrees of success, but all are at least noteworthy for their value as prototype implementations.

- The whole process of formal program derivation (or specification refinement) in an arbitrary domain of application depends on having a formal definition of the target programming language, so that the transformation from specification to implementation can take place within a homogeneous framework. While the definition of the specification and programming languages cannot always be guaranteed to be expressed using the *same* formalism, it should at least be possible to construct a suitable mapping between them.

- One of the major goals of formal specification as a tool in software engineer-

ing is the development of provably correct programs. In order for this to be meaningful, the result of a process of formal refinement, the program, should not then be subjected to software which itself has not been formally derived: if we cannot depend on the compiler, then the whose process is in doubt.

The need for formality in the description programming languages has given rise to a variety of specification languages; some of the main approaches include:

- Expressions

  This is the simplest way of describing any language, where we simply use the elements of a set, along with some given collection of operations over that set: one example is regular expressions, commonly used to describe a languages lexical syntax. A particular characteristic of such definitions is that they are "flat", providing no hierarchical structuring on the language

- Grammars

  The syntactic description formalism of choice is, unquestionably, the context-free grammar (CFG). Almost every programming language will at least have a formal description of its syntax either as a standard CFG, or using one of its variants such as BNF. More powerful grammars, such as context-sensitive or free grammars could be use to describe semantics, but in practice have not been found useful for this.

- Two-Level Grammars

  These seek to stick with the general concept of grammar-based approaches by augmenting ordinary CFGs with specific operations for dealing with semantics. Examples here include attribute grammars, affix grammars and van-Wijngaarden grammars. The first of these lies at the heart of most of the popular tools for compiler construction.

- Operational Semantics

  One of the earliest formalisms: here the concept of a "computation" is described formally (perhaps by using an abstract machine), and the constructs of the programming language are translated into these computations (see [Hen90]

for some examples). The Vienna Definition Language, an ancestor of VDM, was originally used for this purpose.

- Denotational Semantics

  Also called "Scott-Strachey semantics", or simply "mathematical" semantics, these seek to describe a program's components in terms of known mathematical constructs, many of which originate in category theory. Often the functional nature of such descriptions allows them to be quickly translated into programs, blurring the distinction with operational semantics somewhat. [Sch86] is a standard reference.

- Algebraic Semantics

  Here an abstraction of the program's syntax is taken to be an algebraic signature, and all other aspects of the language, including its concrete syntax and various aspects of its semantics, are seen as models of that signature. Formalisms based on this approach include the ASF of [Kli93].

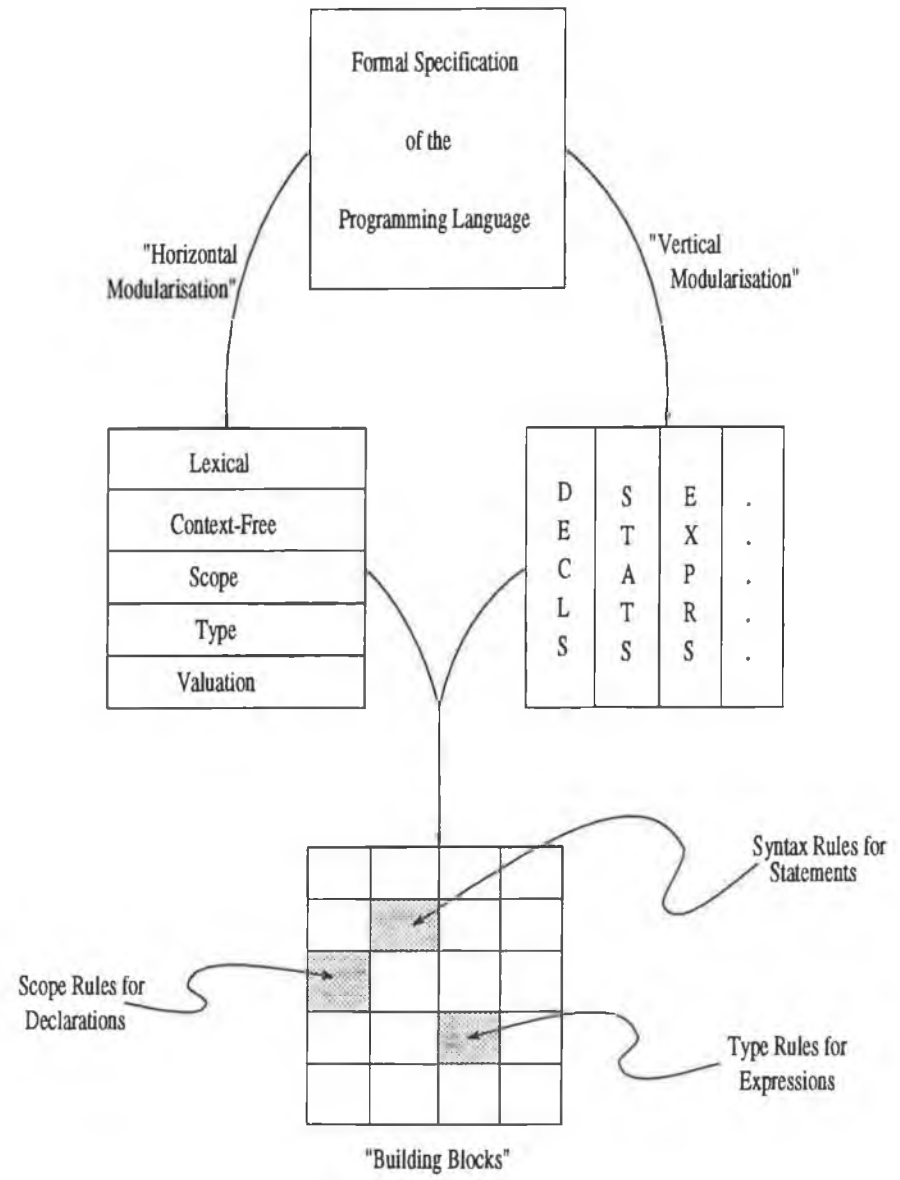
For a general overview of semantic formalisms, see [Pag81], [Wat91] or [vL90, Vol. 2].

1.3 Integrating Specification Formalisms

It is fundamental to our approach that we do not consider a programming language as a single specification entity, but as the result of combining specifications from a number of different formalisms. Each of these formalisms may be seen as a logic, possessing its own syntax and semantics: these should not be confused with the syntax and semantics of the programming language itself. Our goal then is to provide a framework where each of these individual specification formalisms can interact to provide, jointly, the definition of the programming language.

Roughly speaking we might regard the traditional decomposition of a programming language specification as being "horizontal" in nature with clear, well-defined boundaries between the different layers. As an alternative, we wish to incorporate

"vertical" slicing between the specifications, so that components from different specification languages which describe the same programming language concepts may be linked.



Combining horizontal and vertical modularisation

Note that we do not wish to provide one single specification language that can describe all aspects of a programming language: ordinary languages such as VDM [BJ82], Z [Spi89] or any of the algebraic languages (see particularly [BHK89]) will do nicely for this. Rather we wish to maintain the heterogeneity between the languages, on the grounds that this

- allows different aspects of the language to be described by formalisms specifically suited to that task thus, presumably, making the specification easier to construct and read

- permits different implementation strategies to be considered, some of which may possess greater optimality for specific tasks (e.g. using Finite-State Automata to implement regular definitions, rather than more powerful context-free parsing algorithms)

- facilitates the incorporation of existing descriptions using some of the formalisms mentioned above, or others, since a considerable body of such specifications already exists

- may allow the integration of specification for different programming languages, where the number of different formalisms involved may further increase

Hence we need some sort of structure which is abstract enough to incorporate existing formalisms at the object level. In general: we use programming languages to describe algorithms; we can use specification languages to describe programming languages; what we want is a language that will describe specification languages.

Our thesis is that the theory of *institutions* provides a suitable framework for this type of integration. This theory is based on category theory, a formalism which is increasingly being used to give high-level descriptions of algebraic and logic based languages. Indeed, much of the theory of institutions is based on a categorical semantics for CLEAR [BG80], as is much of the work in denotational semantics on which the semantics of Z as given in [Spi88] is based. Many of the higher-order type-theoretic formalisms which incorporate ordinary classical logic as a sub-components also look towards category theory for a formal definition (see e.g. [AL91] or [Cro93]).

1.4 Structure of the Thesis

In chapter 2 we describe the theory of institutions; while they are based in category theory, many of the concepts should look familiar to anyone with a background in

algebraic specification. As well as presenting the basic structures, we augment these slightly by providing a new construct, which we call a *restraint*, for linking specifications from different formalisms.

Chapters three, four and five contain institutional descriptions of six programming language specification formalisms: regular expressions, context-free grammars, attribute grammars, van Wijngaarden grammars, denotational semantics and axiomatic semantics. The purpose of this presentation is twofold:

1. to demonstrate the suitability of this framework for such descriptions
2. to present the basic results needed to incorporate the different formalisms within the theory of institutions.

In chapter 6 we present an example of a simple programming language, and demonstrate the application of our work by giving a modular, heterogeneous description of aspects of its syntax and semantics, and integrating these in the institutional framework.

Note that while the example is presented as a unit in chapter 6, it should also be read in conjunction with the previous chapters, as components of it will help illuminate the definitions given there.

Chapter 2

Institutions

If a language has a precise, formal semantics, then any sentences from that language constitute a formal specification. Typically, formal specification languages as used in computer science are thought of as working at a "higher" level than programming languages, in that the objects they describe need not be computable or algorithmic in nature. Such languages are generally based on abstract mathematical concepts such as set theory, first-order logic or algebra. In order to compare or integrate formal specification languages we thus require a framework which is general enough to be able to contain each of these, already quite general, formalisms.

Attempts to generalise the concept of a logic can be traced back to Tarski's original works on consequence relations, and emerges most notably in a category-theoretic framework in [Bar74]. Here, in answer to the question "What is a logic?" Barwise takes seven different types of logic and attempts to distill their common properties. While each of these logics has its own language, semantics and form of assertions, the relationships between the latter two under change of language form the basis of a *translation axiom* which asserts that logical consequence is preserved independently of the language used.

Around the same time, and again based on ideas from category theory, much work was being done on algebraic specification languages. The most common approach regarded a set of equations over some signature as denoting the (isomorphism class

of the) corresponding initial algebra (see e.g. [GTW78]). One alternative to this approach was that taken with the algebraic language Clear [BG81], where specifications were interpreted "loosely", in that any model which satisfied the equations was acceptable. The formal semantics of Clear in [BG80] made use of constructions which, it was found, could be generalised to specification languages *not* based on algebra. By parameterising out the algebraic content the remaining skeleton forms the basis of the theory of *institutions*, as described in a series of articles culminating in [GB92].

Closely related approaches include π-institutions [FS88], galleries [May85], foundations [Poi89], logical systems [HST89b] and general logics [Mes89].

2.1 Institutions

When attempting to formally define something we must first fix on a notation or set of terms with which to denote the objects we wish to work with. Once these have been listed out they must then be defined by specifying their relationship with the objects that they are supposed to denote. Next we must specify (usually by means of a grammar) how to form assertions with the terms, with the understanding that these sentences will describe properties of and relationships between the objects. Finally we must describe some way of giving meaning to the assertions so that their truth or falsehood can be worked out.

$$\begin{aligned} \textit{terms} &\longrightarrow \textit{objects} \\ \textit{sentences} &\longrightarrow \textit{truth values} \end{aligned}$$

The one condition we place on this structure is that if we change the notation being used then, since we have not changed the underlying objects, there should be some way of changing the sentences so that their denotation is also static. This is the property which is taken as the distinguishing feature of a logic; specifically:

"Truth is invariant under change of notation"

We note that this approach is entirely denotational in nature. The truth/falsehood

When the name of the institution is clear from the context it will be omitted; thus, assuming afixed institution, we might rephrase the satisfaction condition as:

$$m \models [\sigma]e' \qquad \Leftrightarrow \qquad [\![\sigma]\!]m \models e'$$

Further generalisation, not used here, would use **Cat** as the target of *Sen*, with the extra morphisms representing deduction; i.e. an arrow between objects A and B would imply that for any model in which A is true it will be the case that B is true also. Another possibility is to enhance the concept of satisfaction beyond a simple truth-valued answer, and allow something of the form $m \models e$ to denote an object from some chosen value category. In this context, the definition given above could be seen as using the category **2** as its value-category.

Based on the model-theoretic definition of satisfaction, we can define a syntactic notion of *consequence* which gives a relation between sentences. We say that a sentence e is a consequence of some set of sentences E iff it is satisfied in all models which satisfy E. Note that this is entirely defined in terms of satisfaction, and does not relate to a particular proof system. For any set of sentences E, we write $E^\bullet$ for the set of sentences which are the consequences of E; this echoes the original Tarki-style definition of consequence as developed in papers such as [Sco74] and [Avr91]. (As an alternative to institutions, the π-institutions of [FS88] treat the concept of consequence as primitive, and involve models only as a defined concept).

A number of institutions are described in [GB92]. The institution $\mathcal{EQ}$ of many-sorted equational logic has

- as signatures pairs of the form $\langle S, \Sigma \rangle$, where S is a set (of sort-names) and Σ is a S-indexed set of operators
- a model involves interpreting the sorts as sets and the operators as (appropriately-typed) functions over these sets.
- a sentence assumes the existence of some set of sort-indexed variables, and takes the form $(\forall X)t_1 = t_2$, where X is a list of variables, and t_1 and t_2 are terms formed from the operators and variables (in a sort-consistent manner).

- A sentence over a signature is then "satisfied" in a given model of that signature if for all possible assignments to the values of the variables, the interpretation of t_1 and t_2 yield the same object

With any such definition comes a number of proof obligations: it is necessary to show that the objects defined do in fact form categories and functors and, most importantly from the institutional point of view, that the satisfaction condition holds. The construction of this last proof can, in certain situations, be facilitated using the structures of charters and parchments; examples using the above institution can be found in [GB85].

The above institution can be extended to the institution of (many-sorted) first-order logic with equality, $\mathcal{FOEQ}$, by adding in predicate symbols to the signature, interpreting them as relations in the model, and allowing the use of the standard logical connectives such as conjunction, implication etc. in the sentences.

Other applications of the theory of institutions include:

- Horn-Clause Logic [GB92]
- Modal Logic [Ste92]
- Power algebras [Mos89]
- Logics for information hiding [RR92]
- Specification refinement/implementation [BV87]
- Algebras for dynamic systems [Reg90]

2.2 Working with Institutions

Once we have "set up" an institution for a given specification formalism we are ready to deal with specifications written in that language. At its simplest, a specification consists of a list of sentences over a given signature; in institutional terms, given some signature Σ, a specification in this language would be called a Σ-presentation.

Note that the set of sentences involved in a presentation is not necessarily finite, although this is clearly desirable in many cases.

The standard denotation of a given presentation is taken to be the collection of all those models which satisfy all the sentences in the presentation. For any given Σ-presentation, the collection of models which it denotes forms a full subcategory of $[\![\Sigma]\!]$. For any given presentation we can speak of its *closure* under the consequence operator. A theory is a presentation which is closed; in the absence of models it would not be unreasonable to take a theory as being the denotation of a given presentation.

We can define a category **Pres** of presentations whose objects are pairs of the form $\langle \Sigma, A \rangle$ for any set of Σ-sentences A. There is a morphism between any two objects $\langle \Sigma', A' \rangle$ and $\langle \Sigma, A \rangle$ if there is a signature morphism $\sigma\colon \Sigma' \to \Sigma$ such that $\sigma(A'^{\bullet}) \subseteq A^{\bullet}$. This has a full subcategory **The** of theories whose objects are of the form $\langle \Sigma, A \rangle$ such that A is closed. We note in addition the existence of a forgetful functor $\mathit{Sign}\colon \mathbf{Pres} \to \mathbf{Sign}$ sending presentations to their underlying signature.

It is not usual to build whole specifications from just a single presentation; usually we will want operations within the language which allow us to modularise the descriptions. Thus we envisage some kind of language for working with these "modules" in order to produce presentations; one of the most useful features of institutions is the ability to define an algebra for manipulating these modules in a manner which applies uniformly to a broad range of specification languages.

In the next section we present such a module algebra; as such it parallels closely the Clear specification language. In such definitions it is common to blur the distinction between presentations and components of the module algebra; however, to ensure clarity we will be somewhat pedantic in differentiating between these. [1]

[1] The matter is somewhat worse in Clear, as the standard name for a module is a "theory"!

2.2.1 Modularising Specifications

This section presents a notation for constructing "modules"; we assume that each such module can denote a presentation from any given institution. We will use the term "specification" in future to refer to any list of modules.

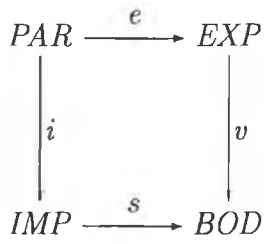
Recent work on modular algebraic specification languages such as [EM90] and [BEPP87] picture any module as consisting of four component specifications:

BOD the body of the specification

EXP along with a morphism v: $EXP \to BOD$, which specifies those components of the module visible to any module which imports it

IMP and a morphism s: $IMP \to BOD$, which specifies the imported modules

PAR specifying the parameters, and two morphisms e: $PAR \to EXP$ and i: $PAR \to IMP$.



For simplicity in our discussion we will omit dealing with any *EXP* presentation; we suggest that this can be compensated for by means of appropriate selection of inclusion morphisms between theories, renaming some of their elements so as to avoid clashes. Similarly, we will assume that the parameter and import modules are disjoint.

First of all we assert that any presentation is a module. To define some module, let us call it *Mod*, we will use a Z-like notation, and write:

Mod

[Signature Part]

[Sentences]

What actually appears in the definition will depend on the particular institution; we do not seek to fix any kind of notation for this. In situations where more than one institution is involved, we will join this to the name of the module, such as $\mathcal{INS} : Mod$.

The simplest way of combining two modules is to include one of them within another; the corresponding presentation then will contain the presentation of the included module as a sub-part. To include some module I within some other module M we will write:

Import I into M

Any module may import a number of other modules, and each of these may also contain (not necessarily disjoint) sets of imported modules. Based on this we can envisage, for any given specification, a graph-like structure where the nodes are presentations corresponding to modules and the edges correspond to (inclusion) presentation morphisms. The graph for any given module may be seen as a cone in **Pres** in which the presentation corresponding to the module itself is the apex, and those corresponding to its its imported modules form the diagram at the base.

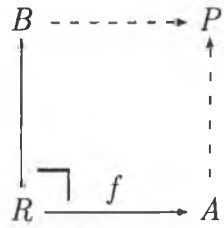
Given any two modules $M1$ and $M2$ we can also combine two modules on an "equal" basis - effectively taking their union. The most basic way of doing this would be, assuming the existence of sums in **Sign**, to define the presentation corresponding to the union of $M1$ and $M2$, which we write as $M1 + M2$ as containing the sum of their signatures and the union of their sentences. However, this disjoint summing is a rather blunt operation, since we will wish to equate common sub-modules. Thus the appropriate categorical construction here is to regard the meaning of $M1 + M2$ as the *co-limit* of the corresponding diagrams in the category **Pres**.

Another useful operation is *renaming*; given a signature morphism between two signatures Σ' and Σ, we can then apply this to any module M' with signature Σ' to get a module with signature Σ. This module will be written as:

Translate M' by σ

Numerous other operations may be specified over these modules but we will have

following diagram in **Pres**:



This gives a presentation containing both B and A in which their common elements, as specified by R, have been identified.

2.3 Combining Institutions

We have seen that one an institution has been constructed for a particular specification formalism it is possible to structure and combine different presentations from that formalism. Given that this framework is not specific to a particular institution, it seems natural to examine the possibilities for combining presentations from *different* institutions. To do this it will be necessary to specify the relationship between their components.

Since an institution consists of a category and two functors, any attempt to relate a pair of them will involve a functor and two natural transformations. Suppose we have two institutions $\mathcal{I} = \langle \mathbf{Sign}, Sen, Mod, \models_I \rangle$ and $\mathcal{I}^+ = \langle \mathbf{Sign}^+, Sen^+, Mod^+, \models_{I^+} \rangle$. The simplest way of relating these is to define a mechanism for translating $\mathcal{I}$ presentations into $\mathcal{I}^+$ (or vice-versa, depending on which is more suitable for the given instance). To do this [Mes89] defines the following:

Definition 2.2 *Institution Mapping*
Given two institutions $\mathcal{I}$ and $\mathcal{I}^+$ as above, we define an institution mapping $\Phi: \mathcal{I} \Rightarrow \mathcal{I}^+$ as consisting of:

1. *a functor Φ: $\mathbf{Sign} \rightarrow \mathbf{Sign}^+$*

2. *a natural transformation α: $Sen \Rightarrow (\Phi \fatsemi Sen^+)$*

a natural transformation $\beta: (\Phi ; Mod^+) \Rightarrow Mod$

that for each Σ *in* **Sign**, *the following condition holds:*

$$m^+ \models_{\Phi(\Sigma)} \alpha_\Sigma(e) \quad \Leftrightarrow \quad \beta_\Sigma(m^+) \models_\Sigma e$$

e m^+ *is a* $\Phi(\Sigma)$*-model from* $\mathcal{I}^+$ *and* e *is a set of* Σ*-sentences from* $\mathcal{I}$.

sort of operation is useful in a number of situations. Perhaps $\mathcal{I}$ represents
: weaker formalism (and which is thus easier to implement) in which part of a
lem has been defined, and this now needs to be linked in to the main body of
pecification. Another possibility is that the specification consists of a number
fferent formalisms, and $\mathcal{I}^+$ is some language which connects them all together.
as is quite common in software engineering, $\mathcal{I}$ could be "more abstract" than
and the mapping constitutes the basis of a refinement step (with β representing
nany-to-one relationship between concrete and abstract models).

iote that institutions and institution mappings form a category with the obvi-
dentity and composition of functors between signatures and α- and β-generated
ws in **Set** and $\mathbf{Cat}^{\mathbf{op}}$ respectively. If the signature categories of any two insti-
ins $\mathcal{I}$ and $\mathcal{I}^+$ allow, then we can conceive of structures such as product and sum
tutions etc.

.1 Constraints

definition of institution mappings given above would seem to be intuitively
ect; however an alternative version is presented in [GB92] in which the natural
sformations go in the *opposite* direction; these are called institution *morphisms.*

nition 2.3 *Institution Morphism*
n two institutions $\mathcal{I}$ *and* $\mathcal{I}^+$ *as before, we define an institution morphism* $\Phi: \mathcal{I} \Rightarrow$
is consisting of:

a functor Φ: **Sign** $\rightarrow$ **Sign**$^+$

from $\mathcal{I}^+$.

hat mappings en-
els), while the use

ormations can be
GB92]. Given an
luplex-constraint

ation morphism,

ences from $[\Sigma]$ -
treat constraints
ny $\sigma: \Sigma \rightarrow \Sigma'$ we

$\Phi(\sigma)\rangle$ via Φ

ures and models
lowed to include
nts as follows:

$\models_{\Phi(\Sigma)} P^+$

to allow $\mathcal{I}$ to
ntax present to

enable other aspects to be modelled (presumably) more successfully in $\mathcal{I}^+$ which is more suited to this purpose. We note that it can be shown that the co-completeness of the category of signatures in $\mathcal{I}$ is enough to also ensure the same in $\mathcal{D}(\Phi)$.

As noted in noted in [Mes89, §4.2], the definitions of an institution morphism and an institution mapping are not dual, and both may be needed to exploit the full power of translations between institutions in general.

2.3.2 Restraints

While the concept of a constraint is generally useful, it will not always suit our purposes here, particularly in relation to checking static semantics. Suppose, for example, that we are given some institution $\mathcal{SYN}$ which describes the (context-free) syntax of our language. We might then envisage some other set of institutions each of which describes some aspect of the static semantics of the language (such as scope rules, type rules, valuations etc.).

Clearly the structures as specified by $\mathcal{SYN}$, while being syntactically-correct programs, need not necessarily be semantically valid, and should thus be constrained by some presentation from each of the semantic institutions. To preserve orthogonality, it is desirable that these semantic descriptions be kept separate from each other: their only correspondence is via the syntactic institution.

Suppose then that we have some institution $\mathcal{SEM}$ specifying a static-semantic component. So, given a syntactic presentation *Syn* and a semantic presentation *Sem*, we want to relate them in some way, so that models of the former can be constrained to fit in with models of the latter. In order to use constraints on $\mathcal{SYN}$-models, we might try to construct an institution morphism $ST: \mathcal{SYN} \Rightarrow \mathcal{SEM}$, where:

- Φ: $\mathbf{Sign}_{\mathcal{SYN}} \rightarrow \mathbf{Sign}_{\mathcal{SEM}}$ which "upgrades" a syntactic signature to a semantic one whose specifically semantic component is empty

- $\alpha: (\Phi \fatsemi Sen_{\mathcal{SEM}}) \Rightarrow Sen_{\mathcal{SYN}}$ extracts from the semantic description the piece of syntax to which it refers

- $\beta: Mod_{\mathcal{SYN}} \Rightarrow (\Phi \fatsemi Mod_{\mathcal{SEM}})$ maps any syntactically correct program straight into the semantic domain, since a model of a signature in the range of Φ can impose no (semantic) constraints on it

We would then envisage constraining presentations in $\mathcal{SYN}$ which specify syntactically correct fragments, with presentations from $\mathcal{SEM}$ which restrict the models to those which are also semantically correct. The problem with this is that the satisfaction of a constraint by some model m from $\mathcal{SYN}$ is defined in terms of the model $\beta(m)$. By the construction this cannot carry semantic information (since then it would be unclear how to find a target for *every* model in $\mathcal{SYN}$), and thus its satisfaction or otherwise in $\mathcal{SEM}$ does not specify the sort of information we are looking for.

Thus it would seem that we must settle on a definition of β which goes in the opposite direction; however, if we try to construct an institution morphism from $\mathcal{SEM}$ to $\mathcal{SYN}$, we find ourselves constraining semantic models rather than syntactic ones.

What we need is to also reverse our concept of constraint; to do this we introduce *restraints*. The basic idea here is that given any two institutions $\mathcal{I}$ and $\mathcal{J}$ there will nearly always be a "natural" choice of morphism between them, based on their construction and on the intended use of the morphism. The problem is that, given such a choice, say from $\mathcal{I}$ to $\mathcal{J}$, if we use constraints we necessarily qualify $\mathcal{I}$-theories by those from $\mathcal{J}$: the end result, however, is still an $\mathcal{I}$-theory. Restraints, on the other hand, allow us to keep the morphism in the same direction, but this time qualify $\mathcal{J}$-theories by $\mathcal{I}$-theories: the result is still a $\mathcal{J}$-theory. (We require the *existence* of a suitable model in $\mathcal{I}$).

Formally we define:

Definition 2.4 *Restraints*

Given two institutions $\mathcal{I}$ and $\mathcal{I}^+$, some institution morphism $\Phi: \mathcal{I} \Rightarrow \mathcal{I}^+$, and some signature Σ^+ from $\mathcal{I}^+$, a Σ^+-restraint is of the form:

$$\mathsf{Restrain}\ \Sigma^+\ \mathsf{by}\ \langle P, \theta \rangle\ \mathsf{via}\ \Phi$$

where P is a presentation from $\mathcal{I}$ and $\theta: \Phi(Sign(P)) \to \Sigma^+$ is a signature morphism from $\mathcal{I}^+$.

We suggest that these can play the role of sentences from $\mathcal{I}^+$ in a similar manner to constraints, and, given any signature morphism $\sigma: \Sigma^+ \to \Sigma'$ from $\mathcal{I}^+$, we define:

$$[\sigma](\mathsf{Restrain}\ \Sigma^+\ \mathsf{by}\ \langle P, \theta \rangle\ \mathsf{via}\ \Phi) \quad = \quad \mathsf{Restrain}\ \Sigma'\ \mathsf{by}\ \langle P, (\theta;\ \sigma) \rangle\ \mathsf{via}\ \Phi$$

Most importantly, we can define satisfaction for these sentences:

Definition 2.5 *Satisfaction of Restraints*

Given an institution morphism and Σ^+-restraint as above, and some Σ^+-model m^+, we define satisfaction as:

$$m^+ \models_{\Sigma^+} \mathsf{Restrain}\ \Sigma^+\ \mathsf{by}\ \langle P, \theta \rangle\ \mathsf{via}\ \Phi$$

$$\Leftrightarrow$$

$$\exists\, m \in |[\![Sign(P)]\!]| \ \cdot\ m \models_{Sign(P)} P \ \wedge\ \beta(m) = [\![\theta]\!](m^+)$$

We suggest now that given any institution and institution morphism, we can construct a new institution by allowing restraints to appear as sentences; to verify this we need to prove the satisfaction condition:

Lemma 2.6 *Satisfaction condition for Restraints*

Given any institution morphism $\Phi: \mathcal{I} \Rightarrow \mathcal{I}^+$ as above, any signature morphism $\sigma: \Sigma^+ \to \Sigma'$ from $\mathcal{I}^+$, any Σ'-model m' and Σ^+-restraint r, we have:

$$[\![\sigma]\!](m') \models^+_\Sigma r \quad \Leftrightarrow \quad m' \models_{\Sigma'} [\sigma](r)$$

Proof:

Letting r be $\mathsf{Restrain}\ \Sigma^+\ \mathsf{by}\ \langle P, \theta \rangle\ \mathsf{via}\ \Phi$, the left-hand-side of the satisfaction condition tells us that:

$$\exists\, m \in |[\![Sign(P)]\!]| \ \cdot\ m \models_{Sign(P)} P \ \wedge\ \beta(m) = [\![\theta]\!]([\![\sigma]\!](m'))$$

Since $[\![\cdot]\!]$ is a functor (into $\mathbf{Cat}^{\mathbf{op}}$) we know that $([\![\sigma]\!];\ [\![\theta]\!]) = [\![\theta;\ \sigma]\!]$, and thus can assert that:

$$\exists\, m \in |[\![Sign(P)]\!]| \ \cdot\ m \models_{Sign(P)} P \ \wedge\ \beta(m) = [\![\theta;\ \sigma]\!](m')$$

which is exactly the definition for:

$$m' \models_{\Sigma'} \mathsf{Restrain}\ \Sigma'\ \mathsf{by}\ \langle P, (\theta;\ \sigma)\rangle\ \mathsf{via}\ \Phi$$

□

2.3.3 Presentation-Based Restraints

A wide number of variations on the basic concepts of constraint and restraints are possible; one more that we will require is that of a *presentation-based* restraint. The need for this occurs in situations when we cannot define a functor between the signature of the institutions that will suit our purpose - instead we wish to define a similar mapping in the context of some specific group of sentences involved. Thus we will define a mechanism for mapping presentations (i.e. signatures and sentences) from one institution into presentations in the other.

We have already noted the existence of a category **Pres** for any institution whose objects are presentations, and whose morphisms are presentation morphisms (all of which are based on signature morphisms). We can extend this to define a functor $ModP$: **Pres** $\rightarrow$ $\mathbf{Cat}^{\mathbf{op}}$, associating with any presentation P a category $\mathbf{ModP}(P)$ whose elements are all models of the presentation (this is a sub-category of $[\![Sign(P)]\!]$). We will overload our notation and write $\mathbf{ModP}(P)$ as $[\![P]\!]$; it should be clear from the context which functor is intended.

Using this functor, we can then define:

Definition 2.7 *Presentation-based mappings*

Given two institutions $\mathcal{I}$ and $\mathcal{I}^+$ we can construct a presentation-based mapping between them by specifying:

- *A functor* Φ: $\mathbf{Pres} \rightarrow \mathbf{Pres}^+$
- *A natural transformation* β: Φ; $ModP^+ \Rightarrow ModP$

such that for any presentations P *and* P^+ *in* $\mathcal{I}$ *and* $\mathcal{I}^+$ *respectively, and any* P^+*-model* m^+*, we have:*

$$m^+ \models \Phi(P) \qquad \Leftrightarrow \qquad \beta(m^+) \models P$$

We do not need to define a natural transformation α as before, since Φ will now take care of sentences as well. Based on this we can now restrain (models of) presentations in one institution by those in another:

Definition 2.8 *Presentation-based restraints*

Given some presentation-based mapping Φ: $\mathcal{I} \Rightarrow \mathcal{I}^+$*, and some presentation* P^+ *from* $\mathcal{I}^+$*, we can define a presentation-based restraint as being of the form:*

$$\mathsf{Restrain}\ P^+\ \mathsf{by}\ \langle P, \theta \rangle\ \mathsf{via}\ \Phi$$

where P *is a presentation from* $\mathcal{I}$*, and* θ: $\Phi(P) \rightarrow P^+$ *is a presentation morphism from* $\mathbf{Pres}^+$*.*

Any signature morphism is, by definition, consequence preserving and so, given some signature Σ and some Σ-presentation P^+, a signature morphism σ: $\Sigma \rightarrow \Sigma'$ naturally gives rise to a presentation morphism from P^+ into $[\sigma](P^+)$. If we denote this by $[\sigma]$ also, we can then regard constraints as sentences by defining:

$$[\sigma](\mathsf{Restrain}\ P^+\ \mathsf{by}\ \langle P, \theta \rangle\ \mathsf{via}\ \Phi) \qquad = \qquad \mathsf{Restrain}\ [\sigma](P^+)\ \mathsf{by}\ \langle P, \theta;\ [\sigma] \rangle\ \mathsf{via}\ \Phi$$

A $Sign(P^+)$-model satisfies a constraint such as the one above if it can be regarded as being both a model of P and of P^+ in the following way:

Definition 2.9 *Satisfaction of presentation-based constraints*

Given any presentation-based mapping Φ: $\mathcal{I} \Rightarrow \mathcal{I}^+$*, any presentation-based constraint* $\mathsf{Restrain}\ P^+\ \mathsf{by}\ \langle P, \theta \rangle\ \mathsf{via}\ \Phi$ *as above, and any* P^+*-model* m^+*, we define:*

$$m^+ \models_{Sign(P^+)} \mathsf{Restrain}\ P^+\ \mathsf{by}\ \langle P, \theta \rangle\ \mathsf{via}\ \Phi \qquad \Leftrightarrow \qquad \beta([\![\theta]\!](m^+)) \models_{Sign(P)} P$$

We can use such sentences just like any others over $Sign(P^+)$ since they are consistent with the satisfaction condition:

Lemma 2.10 *Satisfaction condition for presentation-based restraints*

Given any presentation-based institution mapping $\Phi: \mathcal{I} \Rightarrow \mathcal{I}^+$ *as above, any signature morphism* $\sigma: \Sigma \to \Sigma'$ *and any* Σ*-presentation* P^+ *from* $\mathcal{I}^+$*, any* $[\sigma](P^+)$*-model* $m^{\#}$ *and any presentation-based restraint* c *on* P^+*, we have:*

$$[\![\sigma]\!](m^{\#}) \models_{\Sigma} c \qquad \Leftrightarrow \qquad m^{\#} \models_{\Sigma'} [\sigma](c)$$

Proof:

The proof is similar to that for previous types of contraint; letting c be the constraint Restrain P^+ by $\langle P, \theta \rangle$ via Φ as above, we have

$$\begin{aligned}
[\![\sigma]\!](m^{\#}) \models_{\Sigma} & \text{ Restrain } P^+ \text{ by } \langle P, \theta \rangle \text{ via } \Phi \\
\Leftrightarrow \quad & \beta([\![\theta]\!]([\![\sigma]\!](m^{\#}))) \models_{Sign(P)} P \\
\Leftrightarrow \quad & \beta([\![\sigma;\ \theta]\!](m^{\#})) \models_{Sign(P)} P \\
\Leftrightarrow \quad & m^{\#} \models_{\Sigma'} \text{ Restrain } [\sigma](P^+) \text{ by } \langle P, [\sigma];\ \theta \rangle \text{ via } \Phi \\
\Leftrightarrow \quad & m^{\#} \models_{\Sigma'} [\sigma](\text{Restrain } P^+ \text{ by } \langle P, \theta \rangle \text{ via } \Phi)
\end{aligned}$$

□

2.4 Conclusions

In this chapter we have laid the basic foundation on which we propose to build and integrate specifications of programming language formalisms. We have introduced the theory of institutions, and fixed our notation for dealing with modules in an institutional specification. Additionally, we have added a new type of constraint to the theory: a *restraint*, which works in the opposite direction.

In so far as using the abstract syntax of a language as an initial algebra in its class of models characterises the "algebraic approach" to programming language semantics, the use of restraints in the above manner could be said to characterise the "institutional approach". We envisage a situation where a language's context-free syntax is restrained by its semantic definition, allowing them to be defined in

separate, but related institutions. As we still deal with context-free models with this strategy, the syntax may be restrained by a *number* of different semantic institutions. We note finally that since these are all linked back to the weaker institution they do not share semantic information, and so this method applies specifically to *static*, rather than dynamic, semantics.

Chapter 3

Syntax-Based Definitions

3.1 Introduction

In this chapter we begin the process of casting programming language specification formalisms into an institutional setting. We start with syntax, and with institutions for two of the formalisms most commonly used for defining syntax:

- $\mathcal{REG}$ the institution for regular languages
- $\mathcal{CFREE}$ the institution for context-free definitions

While regular expressions are not essential to language definition, they are used quite commonly, and lay much of the groundwork for dealing with context-free languages.

The reader may wish to refer to the examples given in chapter 6 while reading the definitions given here.

Before we define the actual institutions, we will first fix some concepts and notation from formal language theory.

3.1.1 Languages - Basic Definitions

The three most basic definitions in formal language theory are those of an alphabet, string and language:

- An *alphabet* is any set of symbols

- A *string* over some alphabet Σ is any sequence (including the empty sequence) of symbols from Σ.

- A *language* over an alphabet Σ is any set of strings over Σ (including the empty set).

Rather than adopt a generic notation for sequences, we shall follow convention and adopt the usual notation for strings. Thus for any given alphabet Σ, we suggest:

- ε_Σ denotes the (unique) empty string

- For any $a \in \Sigma$, "a" is the string of length 1 containing only the symbol a

- For any strings s and t, $s \cdot t$ will denote the concatenation of s and t.

The set Σ, along with concatenation as the distinguished binary operator and ε_Σ as its identity, forms a monoid.

Finally, given any two languages M and N over the same alphabet, we can define:

- $M \cup N$ to be the union of the two languages

- $M \frown N$ to be the language whose strings are of the form $m \cdot n$ for any $m \in M$ and $n \in N$.

- M^* to be the language whose strings are formed by taking the reflexive and transitive closure of the concatenation operation over the strings in M (this is known as the *Kleene closure* of M).

We are now ready to define our first institution.

3.2 The Regular Institution

A regular language is one which can be constructed from an alphabet of characters using only the operations of union, concatenation and Kleene closure. It forms the most basic of the levels in the Chomsky Hierarchy, and is distinguished by the simplicity of its iteration mechanism. The following section gives the basic definitions for $\mathcal{REG}$, the regular institution.

3.2.1 Basic Definitions

The most straightforward approach would be to have alphabets for signatures, languages for models and regular expressions for sentences. However, the operation of languages which deal with regular expressions, such as lex, is generally a little more subtle, in that they allow specific subsets of the defined language to be named, so that these names may be used in later parts of the compilation where they are referred to as *tokens*. Thus we will take the viewpoint that the purpose of a specification in the regular institution is to construct a mapping between language names and sets of symbols.

A signature then will consists of an alphabet, over which to define the regular expressions, and names for the sets which are defined by them. Formally we define:

Definition 3.1 *Signatures in* $\mathcal{REG}$

A regular signature Σ *consists of two sets:*

1. Σ_A, *a finite set of alphabet symbols*
2. Σ_N, *a finite set of names (tokens) for the languages being defined*

Morphisms consists of pairs of functions $\langle \sigma_A, \sigma_N \rangle$, one for each set; this is easily seen to form a category.

A model of an alphabet will involve mapping the elements of Σ_N to languages. To allow full flexibility we will not re-use the alphabet symbols in the model; instead we will interpret them into some new set, whose elements we shall refer to simply

as "characters". In addition we will allow one terminal symbol to be related to a number of such characters; the idea here is that we allow for the possibility that the specification mechanism may be "too blunt", and that some other formalism may constrain these values further. This is particularly important in the context-free case (an enhancement of the regular case), but we allow for it here to ease comparison.

Thus we define:

Definition 3.2 *Models in $\mathcal{REG}$*

For any signature $\langle \Sigma_A, \Sigma_N \rangle$, a model I is a triple of the form $\langle I_C, I_A, I_N \rangle$ where:

1. *I_C is some set of characters*
2. *$I_A: \Sigma_A \to \wp(I_C)$ is a function interpreting alphabet symbols as sets of characters*
3. *$I_N: \Sigma_N \to \wp(I_C^*)$ is a function associating with each name in Σ_N the language "corresponding to" that name.*

A morphism μ between two Σ-models I and J consists of a function $\mu_S: I_C \to J_C$, with its obvious extension defining J_A and J_N.

Given any **Sign**-morphism $\sigma: \Sigma' \to \Sigma$, we define:

$$[\![\sigma]\!](\langle I_C, I_A, I_N \rangle) = \langle I_C, \sigma_A \fatsemi I_A, \sigma_N \fatsemi I_N \rangle$$

A sentence in the institution will associate a name with a regular expression; we note that a number of different regular expressions may be associated with the same name.

Definition 3.3 *Regular Expressions*

Given some signature Σ, we can define REG_Σ, the set of regular expressions over the signature inductively as follows:

1. *Λ is a regular expression*
2. *For any $a \in \Sigma_A$, 'a' is a regular expression*

3. *If r is a regular expression, then so is $r*$*

4. *If q and r are regular expressions, then so is $q.r$*

5. *If q and r are regular expressions, then so is $q \mid r$*

Now we can define:

Definition 3.4 *Sentences in $\mathcal{REG}$*

For any signature Σ, the sentences over this signature are all of the form $(r : N)$, where $r \in REG_\Sigma$ and $N \in \Sigma_N$

Sentence morphisms are defined inductively over the components of the sentence:

Definition 3.5 *Sentence morphisms in $\mathcal{REG}$*

For any signature morphism σ, the corresponding sentence morphism is defined by:

$$[\sigma](r : N) = ([\sigma](r) : \sigma_N(N))$$

where $[\sigma](r)$ is defined as:

$$\begin{aligned}
[\sigma]\Lambda &= \Lambda \\
\forall\, a \in \Sigma \cdot \quad [\sigma]\text{`}a\text{'} &= \text{`}\sigma_A(a)\text{'} \\
\forall\, r \in [\Sigma] \cdot \quad [\sigma](r*) &= ([\sigma]r)* \\
\forall\, q, r \in [\Sigma] \cdot \quad [\sigma](q.r) &= ([\sigma]q).([\sigma]r) \\
\forall\, q, r \in [\Sigma] \cdot \quad [\sigma](q \mid r) &= ([\sigma]q) \mid ([\sigma]r)
\end{aligned}$$

A sentence of the form $(r : N)$ is satisfied in some model iff the regular language which corresponds to r is contained in the language assoicated with N. To make this precise, we define the language associated with a regular expression:

Definition 3.6 *$LAN_I(r)$, the regular language corresponding to r*

For any Σ-model I as above, we define:

$$\begin{aligned}
LAN_I(\Lambda) &= \{\varepsilon_{I_C}\} \\
\forall\, a \in \Sigma \cdot \quad LAN_I(\text{`}a\text{'}) &= \{\text{“}c\text{”} \in I_C^* \mid c \in I(a)\} \\
\forall\, r \in [\Sigma] \cdot \quad LAN_I(r*) &= (LAN_I(r))^* \\
\forall\, q, r \in [\Sigma] \cdot \quad LAN_I(q.r) &= (LAN_I(q)) \frown (LAN_I(r)) \\
\forall\, q, r \in [\Sigma] \cdot \quad LAN_I(q \mid r) &= (LAN_I(q)) \cup (LAN_I(r))
\end{aligned}$$

We note that this function LAN_I commutes in a natural way with signature morphisms:

Lemma 3.7 *For any Σ-model I, any $r \in REG_\Sigma$, and any $\mathbf{Sign}_{\mathcal{REG}}$-morphism $\sigma: \Sigma' \to \Sigma$, we have:*

$$LAN_I([\sigma]r) = LAN_{\sigma;I}\,(r)$$

Proof: *By induction over the regular expression r*

1. $r = \Lambda$

$$\begin{aligned} LAN_I([\sigma]\Lambda) &= LAN_I(\Lambda) \\ &= \{\varepsilon_{I_C}\} \\ &= LAN_{\sigma;I}(\Lambda) \end{aligned}$$

2. $r = \text{`}a\text{'}$

$$\begin{aligned} LAN_I([\sigma]\ \text{`}a\text{'}) &= LAN_I(\text{`}\sigma_A(a)\text{'}) \\ &= \{\text{“}c\text{”} \in I_C^* \mid c \in I(\sigma_A(a))\} \\ &= \{\text{“}c\text{”} \in (\llbracket\sigma\rrbracket(I_C))^* \mid c \in \llbracket\sigma\rrbracket(I)(a)\} \\ &= LAN_{\sigma;I}\,(\text{`}a\text{'}) \end{aligned}$$

3. $r = p*$

$$\begin{aligned} LAN_I([\sigma](p*)) &= LAN_I(([\sigma]p)*) \\ &= (LAN_I([\sigma]p))^* \\ &= (LAN_{\sigma;I}\,(p))^* \\ &= LAN_{\sigma;I}\,(p*) \end{aligned}$$

4. $r = p.q$

$$\begin{aligned} LAN_I([\sigma](p.q)) &= LAN_I(([\sigma]p).([\sigma]q)) \\ &= LAN_I([\sigma]p) \frown LAN_I([\sigma]q) \\ &= LAN_{\sigma;I}\,(p) \frown LAN_{\sigma;I}\,(q) \\ &= LAN_{\sigma;I}\,(p.q) \end{aligned}$$

5. $r = p \mid q$

$$\begin{aligned} LAN_I([\sigma](p \mid q)) &= LAN_I(([\sigma]p) \mid ([\sigma]q)) \\ &= LAN_I([\sigma]p) \cup LAN_I([\sigma]q) \\ &= LAN_{\sigma;I}\,(p) \cup LAN_{\sigma;I}\,(q) \\ &= LAN_{\sigma;I}\,(p \mid q) \end{aligned}$$

□

Finally we are ready to define satisfaction:

Definition 3.8 *Satisfaction in $\mathcal{REG}$*

For any $r \in [\Sigma]$,

$$\langle I_C, I_A, I_N \rangle \models_\Sigma (r : N) \qquad \Leftrightarrow \qquad LAN_I(r) \subseteq I_N(N)$$

The satisfaction condition now follows from the previous lemma:

Lemma 3.9 *In the insititution $\mathcal{REG}$, the satisfaction condition holds*

Proof:

The satisfaction condition states that for any Σ-model I, and $\mathbf{Sign}_{\mathcal{REG}}$-morphism $\sigma\colon \Sigma' \to \Sigma$ and any Σ'-sentence $r' : N'$,

$$\langle I_C, I_A, I_N \rangle \models_\Sigma [\sigma](r' : L') \qquad \Leftrightarrow \qquad [\![\sigma]\!]\langle I_C, I_A, I_N \rangle \models_{\Sigma'} (r' : N)$$

Applying the sentence and model morphisms, we see that this is:

$$\langle I_C, I_A, I_N \rangle \models_\Sigma ([\sigma]r' : \sigma_N(N')) \qquad \Leftrightarrow \qquad \langle I_C, \sigma_A \fatsemi I_A, \sigma_N \fatsemi I_N \rangle \models_{\Sigma'} (r' : N)$$

By the definition of satisfaction we can restate this as:

$$LAN_I([\sigma]r') \subseteq I_N(\sigma_N(N')) \qquad \Leftrightarrow \qquad LAN_{\sigma;\, I}\ (r') \subseteq (\sigma_N \fatsemi I_N)(N)$$

which is true, since $LAN_I([\sigma]r') = LAN_{\sigma;\, I}\ (r')$ by the lemma.

□

3.2.2 Properties of the Regular Institution

Any *presentation* in the regular institution is simply a list of pairs of regular expressions and language names. A model of this presentation is one which satisfies *every* sentence in the presentation. Thus while combining specifications is normally regarded as a "conjunction" operation in, say, first-order logic, it actually corresponds to the *union* operation here. We can state:

Lemma 3.10 *For any signature Σ, any Σ-model I, any token N, and any two regular expressions p and q, we have that:*

$$\frac{I \models_\Sigma (p : N), \qquad I \models_\Sigma (q : N)}{I \models_\Sigma (p \mid q) : N}$$

Proof:

By the definition of satisfaction we can rewrite this as:

$$\frac{LAN_I(p) \subseteq I_N(N), \qquad LAN_I(q) \subseteq I_N(N)}{(LAN_I(p) \cup LAN_I(q)) \subseteq I_N(N)}$$

which is easily seen to be true.

□

Choosing a model for a presentation basically involves choosing "big enough" languages for the tokens to hold all of the corresponding regular expressions. Using the above lemma, we can see that it is possible to merge all the sentences involving a particular token into just one sentence. Based on this, we can pick as our model the regular language which corresponds *exactly* to this regular expression. Doing this for each sentence, and assuming that I_A is bijective, yields an *initial* model for the presentation.

It is common to assume that presentations are of finite length, since the nature of a model can differ considerably for an infinite presentation. In fact we can see that an infinite presentation in the regular institution would simply allow us to list all strings in a given language. Since there are no restrictions on this, the language need not be regular; thus models may specify context-free or any other type of language. We note therefore that only finite length presentations give us the standard interpretation of a regular expression.

Since the initial model is minimal it maximally constrains the theory of the presentation. Thus, for a given token N, if we take all sentences of the form $(r : N)$ in the theory, and exclude those involving union or Kleene closure, we get exactly the language corresponding to N. For a finite presentation, this is always a regular

language.

As the objects of $\mathbf{Sign}_{\mathcal{REG}}$ are just sets, it is easily seen that this category has all colimits, using set-theoretic union in the ordinary way. We get the sum of two modules simply by taking the union of the corresponding regular languages; a parameter to a module specifies a minimal language that must be satisfied by any argument. Similarly, the process of actualising a parameter involves identifying the subcomponent specified by the formal parameter and taking the union of the modules for the argument and the body.

We are now ready to deal with the next formalism in line, that of context-free languages. Based on our definition above, we can see that there will be many similarities between a presentation in the regular institution and a context-free grammar. The main difference of course, and the reason why the context-free formalism is more powerful, is the possibility of recursion in the context-free rules.

3.3 Context-Free Grammars

Context-Free Grammars are sets of production rules involving terminal and non-terminal symbols, referred to collectively as the *vocabulary*. A production rule defines a rewrite equivalence between a non-terminal and any string of symbols from the vocabulary. For the duration of this section only, let us choose to allow any regular expression over the vocabulary to appear on the right-hand-side of a rule, and call such a string a *rightpart*.[1]

From an operational point of view, we *apply* a rule to a string by replacing its left-hand-side with the symbols on the right-hand-side (or an arbitrarily long sequence of them in the case of Kleene closure). Based on a grammar, one string is derivable from another if we can find a set of rules which, when applied, will rewrite the first string to the second.

[1]Such grammars are often called *right-part-regular* grammars, to distinguish them from ordinary context-free grammars which do not use Kleene closure.

3.3.1 Basic Definitions

Based on the definitions given for regular languages, it seems evident that terminal symbols should appear in the signature. However, we suggest that non-teminals should also appear here. The view taken is that they act not merely as placeholders (like variables) but act to define sub-components of the language. Hence, the inclusion of non-terminals at this level will allow greater flexability in terms of the modularisation of context-free specifications later. Thus:

Definition 3.11 *The category* $\mathbf{Sign}_{\mathcal{CFREE}}$

The category of context-free signatures has as objects pairs $\langle \Sigma_T, \Sigma_N \rangle$*; i.e. sets of terminals and non-terminals. Morphisms are the products of set-theoretic functions.*

In a similar manner to the last institution, we will interpret terminal symbols as sets of characters, and non-terminal symbols as languages. In addition, we will choose to formally denote some language as being *the* language defined by the model; this could be regarded as the language associated with the start symbol. When considering the regular institution we did not need this, since "the" language was effectively the union of the language for each individual name; here, because a context-free specification is hierarchical (whereas our list of regular expressions was basically "flat"), we need to make this distinction. Note that the language corresponding to any given non-terminal is not necessarily a sublanguage of this language.

Definition 3.12 *Models in* $\mathcal{CFREE}$

A model of a signature $\langle \Sigma_T, \Sigma_N \rangle$ *consists of four components:*

1. *A set of characters* I_C
2. *A function* $I_T : \Sigma_T \rightarrow \wp(I_C)$ *mapping terminal symbols to sets of characters*
3. *A function* $I_N : \Sigma_N \rightarrow \wp(I_C^*)$ *mapping non-terminal symbols to languages over* I_C
4. *A language* I_L *over* I_C*, being "the" language defined by the model*

For any two Σ-models, we can define a morphism between them by using a mapping on I_C, and extending this to the other components as for the regular case.

Given any $\mathbf{Sign}_{\mathcal{CFREE}}$-morphism σ with components $\langle \sigma_T, \sigma_N \rangle$, we define

$$[\![\sigma]\!](\langle I_C, I_T, I_N, I_L \rangle) = \langle I_C, (\sigma_T \fatsemi I_T), (\sigma_N \fatsemi I_N), I_L \rangle$$

The immediate choice for the $[\cdot]_{\mathcal{CFREE}}$ functor would be to map a vocabulary to the set of context-free production rules over the vocabulary. However, we will also allow a sentence to be any rightpart for that signature, the idea being that for any presentation these form a set of "given" strings, from which all the others are derived (*axioms* as opposed to rules).

Thus we extend the definition for the regular case

Definition 3.13 *Sentences in $\mathcal{CFREE}$*

As for regular expressions with the addition that for any non-terminal A, 'A' is a rightpart, and for any rightpart r, $(A \rightarrow r)$ is a sentence.
Morphisms are defined by adapting the definition for the regular case, replacing Σ with Σ_T, and adding the following rules:

$$\begin{array}{rlcl} \forall\, A \in \Sigma_N \cdot & [\sigma]\,`A' & = & `\sigma_N(A)' \\ \forall\, A \in \Sigma_N, r \in REG_\Sigma \cdot & [\sigma](A \rightarrow r) & = & ([\sigma](A) \rightarrow [\sigma](r) \end{array}$$

Satisfaction will describe derivability. As before we will need to define what is meant by a context-free language:

Definition 3.14 *$LAN_I(r)$, the context-free language associated with r:*
For any model I, we adapt the rules for the regular case thus:

$$\begin{array}{rlcl} \forall\, a \in \Sigma_T \cdot & LAN_I(`a') & = & \{ \text{“}c\text{”} \in I_C^* \mid c \in I_T(a) \} \\ \forall\, A \in \Sigma_N \cdot & LAN_I(`A') & = & I_N(A) \end{array}$$

Once again, this commutes appropriately with signature morphisms:

Lemma 3.15 *For any Σ-model $\langle I_C, I_T, I_N, I_L \rangle$, any rightpart r, and any signature morphism $\sigma: \Sigma' \to \Sigma$, we have:*

$$LAN_I([\sigma]r) = LAN_{\sigma;I}(r)$$

Proof: *By induction over the sentence r*
The proof is as for $\mathcal{REG}$, with only one new case:

6. $r =$ 'A' (for some $A \in \Sigma_N$)

$$\begin{aligned} LAN_I([\sigma]\text{`}A\text{'}) &= LAN_I(\text{`}\sigma_N(A)\text{'}) \\ &= I_N(\sigma_N(A)) \\ &= LAN_{\sigma;I}(A) \end{aligned}$$

□

We are now ready to define satisfaction:

Definition 3.16 *Satisfaction in $\mathcal{CFREE}$:*

1. *For any rightpart r, we define:*

$$\langle I_C, I_T, I_N, I_L \rangle \models_\Sigma r \quad \Leftrightarrow \quad LAN_I(r) \subseteq I_L$$

2. *For any rule $(A \to r)$ we define:*

$$\langle I_C, I_T, I_N, I_L \rangle \models_\Sigma (A \to r) \quad \Leftrightarrow \quad LAN_I(r) \subseteq LAN_I(A)$$

Note that, as we would expect, satisfaction of a production rule is defined in model-theoretic rather than proof-theoretic terms; we do not need to explicitly state the intuitive version of replacing one string with another. This has the effect of making the appearance of Kleene closure operations in a rightpart more natural and, of course, not committing us to a bottom-up or top-down parsing strategy.

Verification of the satisfaction condition follows from the regular case:

Lemma 3.17 *In the institution $\mathcal{CFREE}$, the satisfaction condition holds*

Proof:

The satisfaction condition for $\mathcal{CFREE}$ may be stated as follows: For any $\sigma: \Sigma' \to \Sigma$, any Σ-model $\langle I_C, I_T, I_N, I_L \rangle$, and any Σ'-sentence e', we have:

$$\langle I_C, I_T, I_N, I_L \rangle \models_\Sigma [\sigma]e' \quad \Leftrightarrow \quad [\![\sigma]\!]\langle I_C, I_T, I_N, I_L \rangle \models_{\Sigma'} e'$$

By the definition of satisfaction, we can break this into two cases; for any rightpart r' and any non-terminal A' from Σ',

$$(LAN_I([\sigma]r') \subseteq I_L) \quad \Leftrightarrow \quad (LAN_{\sigma;I}(r') \subseteq I_L)$$

and

$$(LAN_I([\sigma]A') \subseteq LAN_I([\sigma]r')) \quad \Leftrightarrow \quad (LAN_{\sigma;I}(A') \subseteq LAN_{\sigma;I}(r'))$$

Both of these follow directly from the previous lemma.

□

3.3.2 Properties of the Context-Free Institution

Any presentation in this institution is effectively a context-free grammar. We have not explicitly provided for a start symbol; however, the collection of rightparts in any presentation may be taken to represent the right-hand-sides of start rules. Thus we need only pick some "new" non-terminal S as the start symbol, add it to the signature, and add in a rule of the form $(S \to r)$ for every rightpart r in the presentation. Any presentation which *doesn't* have any standalone rightparts (i.e. consists only of rules) will have the empty language as its initial model.

We note that neither the models of a grammar or a non-terminal are required to be context-free. However, we can see that the initial model will construct those languages which minimally satisfy the rules, and this will give rise to mappings into context-free languages for finite-length presentations. As before, the possibility of

infinitely-long presentations allows for models which are not context-free.

Given any presentation, its corresponding *theory* will contain rules and rightparts:

1. The rules in the theory represent all possible derivation steps, or equivalently, every possible node that could be found in any parse tree
2. The rightparts are usually called *sentential forms*; those sentential forms which only contain terminal symbols are called *sentences*, and correspond to the context-free language generated by the grammar.

Again the conjunction of two sentences is effectively represented internally by the union operation; i.e. :

$$\frac{I \models_\Sigma (A \rightarrow r_1), \qquad I \models_\Sigma (A \rightarrow r_2)}{I \models_\Sigma (A \rightarrow r_1 \mid r_2)}$$

The proof is almost identical to the regular case.

We can prove a similar (expected) result for Kleene closure:

Lemma 3.18 *For any signature Σ, any Σ-model I, non-terminal A and rightpart r*

$$\frac{I \models_\Sigma (A \rightarrow \Lambda), \qquad I \models_\Sigma (A \rightarrow r.A)}{I \models_\Sigma (A \rightarrow r*)}$$

Proof:

By the definition of satisfaction; we can rewrite the statement as:

$$\frac{LAN_I(\Lambda) \subseteq LAN_I(A), \qquad LAN_I(r.A) \subseteq LAN_I(A)}{LAN_I(r*) \subseteq LAN_I(A)}$$

This is the same as:

$$\frac{\{\varepsilon_{\mathbb{C}}\} \subseteq LAN_I(A), \qquad (LAN_I(r) \frown LAN_I(A)) \subseteq LAN_I(A)}{LAN_I(r)^* \subseteq LAN_I(A)}$$

Let $LAN_I(r)^i$ represent the maximal subset of $LAN_I(r)^*$ in which no string has length greater than i. The proof proceeds by induction over i.

- Base case: $i = 0$

  In this case $LAN_I(r)^i = \{\varepsilon_{\mathbb{C}}\}$, which is a subset of $LAN_I(A)$ by the first assumption

- Inductive case: Assume $LAN_I(r)^i \subseteq LAN_I(A)$
  By the second assumption $(LAN_I(r) \frown LAN_I(A)) \subseteq LAN_I(A)$,
  thus $(LAN_I(r) \frown LAN_I(r)^i) \subseteq LAN_I(A)$,
  which is $LAN_I(r)^{i+1} \subseteq LAN_I(A)$

□

Thus, for any presentation in the institution, we will be able to formulate another presentation which does not make use of union or Kleene closure, but which has exactly the same theory. To ease relating $\mathcal{CFREE}$ to other institutions, we will assume from this point on that all rules involve only union and concatenation, but that the results proved are extendible to the full institution as specified above.

We also note that we can demonstrate the implicative nature of production rules by proving a version of *modus ponens*:

Lemma 3.19 *For any signature Σ, model I, non-terminal A and rightpart r*

$$\frac{I \models_\Sigma (A \to r), \qquad I \models_\Sigma A}{I \models_\Sigma r}$$

Proof:
The proof follows directly from the definition of satisfaction; we can rewrite the statement as:

$$\frac{LAN_I(r) \subseteq LAN_I(A), \qquad LAN_I(A) \subseteq I_L}{LAN_I(r) \subseteq I_L}$$

□

Similarly (and equally strightforwarly), we can verify that the more operational proof rule for context-free grammars also holds:

Lemma 3.20 *For any signature Σ, model I, non-terminals A and B and rightparts x, y and z*

$$\frac{I \models_\Sigma (A \to x.\text{`}B\text{'}.z), \qquad I \models_\Sigma (B \to y)}{I \models_\Sigma A \to x.y.z}$$

Proof:

Once again we need only use the definition of satisfaction to verify that this is valid:

$$\frac{LAN_I(x) \frown LAN_I(B) \frown LAN_I(z)) \subseteq LAN_I(A), \qquad LAN_I(y) \subseteq LAN_I(B)}{LAN_I(x) \frown LAN_I(y) \frown LAN_I(z)) \subseteq LAN_I(A)}$$

□

Since the signature consists of a pair of sets we assert the presence of co-limits, and thus we can transform, sum and parameterise context-free presentations as required.

3.4 Relating Regular and Context-Free Definitions

While it is possible to describe the entire sytnax of a programming language using a context-free grammar, it is quite common to break this into a two-step process, using regular expressions to specify some of the allowable words from the language, and then using context-free grammars to specify the allowable combinations of these words. One benefit of this approach is that simpler (or more efficient) algorithms may then be used to implement recognisers for the regular parts of the specification.

To describe this institutionally, we will need to be able to relate the regular and context-free institutions; that is, we will need to define an institution morphism between them. The choice as regards the direction of the morphism is easily made by noting that the natural transformation α can really only go in one direction, as we cannot hope to translate context-free grammar rules back to regular expressions. Thus we construct:

Definition 3.21 *The institution morphism* $CFtoR: \mathcal{CFREE} \Rightarrow \mathcal{REG}$

The three components of the institution morphism are:

1. *The functor* $\Phi: \mathbf{Sign}_{\mathcal{CFREE}} \rightarrow \mathbf{Sign}_{\mathcal{REG}}$ *which sends any context-free signature of the form* $\langle \Sigma_T, \Sigma_N \rangle$ *to a regular signature with* Σ_T *as the alphabet characters, and* Σ_N *as the tokens*

2. *The natural transformation* α: $(\Phi \fatsemi Sen_{\mathcal{REG}}) \Rightarrow Sen_{\mathcal{CFREE}}$ *taking any sentence from the regular institution of the form* $(r : N)$ *and mapping it to* $(N \rightarrow r)$

3. *The natural transformation* β: $Mod_{\mathcal{CFREE}} \Rightarrow (\Phi \fatsemi Mod_{\mathcal{REG}})$ *mapping a context-free model of the form* $\langle I_C, I_T, I_N, I_L \rangle$ *to a model in* $\mathcal{REG}$ *of the form* $\langle I_C, I_T, I_N \rangle$

Verifying that this is in fact an institution morphism is straightforward:

Lemma 3.22

CFtoR: $\mathcal{CFREE} \Rightarrow \mathcal{REG}$, *as defined above, is an institution moprhism*

Proof:

We must show that for every $\mathcal{REG}$-signature Σ, every sentence $(r : N)$ in $[\Sigma]$, and every context-free model I in $[\![\Phi(\Sigma)]\!]$ we have:

$$\langle I_C, I_T, I_N, I_L \rangle \models_{\Phi(\Sigma)} \alpha(r : N) \qquad \Leftrightarrow \qquad \beta(\langle I_C, I_T, I_N, I_L \rangle) \models_{\Sigma} (r : N)$$

By the definition of α and β this becomes:

$$\langle I_C, I_T, I_N, I_L \rangle \models_{\Phi(\Sigma)} (N \rightarrow r) \qquad \Leftrightarrow \qquad \langle I_C, I_T, I_N \rangle \models_{\Sigma} (r : N)$$

which is easily seen to be true via the definition of satisfaction in each institution. □

Thus for any regular module R, and any context free module C, we can construct modules of the form:

$$\textsf{Constrain } C \textsf{ by } \langle R, \theta \rangle \textsf{ via } CFtoR$$

This is a sentence in $\mathcal{CFREE}$, and thus specifes models in that institution.

The signature morphism θ: $Sign(R) \rightarrow Sign(C)$ above establishes the relationship between the tokens and symbols from the regular module and the context-free grammar symbols. This morphism can be used in situations where a number of different lexical symbols correspond to just a single non-terminal from the context-free grammar. While this in itself is quite common in such mappings, it may also be useful if a number of different regular modules are involved.

Chapter 4

Two-Level Grammars

4.1 Introduction

The basic reason for the inability of context-free grammars to describe semantic features is indicated by their name – they cannot deal with context-sensitive information. Semantic analysis depends crucially on such information, since e.g. type and scope correctness are usually determined in the context of having previously processed some sort of declaration which presents the relevant information. Of course this could be remedied by just using a context-sensitive or free grammar, but these yield unintuitive descriptions and are rarely used.

Another grammar-based approach seeks to enhance context-free grammars in other ways so as to allow them deal with contextual information. By far the most popular such method is that of *attribute grammars*, which augment ordinary context-free grammar rules with assertions or statements from some completely different language to specify semantics. (See [DJ90] or [AM91] for surveys). A closely related formalism is that of *affix grammars* ([Kos91] contains a comparison)

An alternative approach is to provide a homogeneous framework for the specification of context: i.e. to use another, different, grammar. This is the view taken by *van-Wijngaarden grammars*, which consists of two "levels" of rules, with components of the lower level being restricted by rules from the higher level (in much the

same way as second-order logic can be used to define meta-concepts from first-order logic). Descriptions and examples of this sort of grammar can be found in [Pag81] and [GJ90].

In the following sections then we present two institutions:

- $\mathcal{ATG}$ for attribute grammars
- $\mathcal{VANW}$ for van-Wijngaarden grammars

and relate them both back into $\mathcal{CFREE}$. As before, examples of their use can be found in chapter 6.

4.2 Attribute Grammars

Compiler design tools such as yacc allow context-free definitions to be enhanced with semantic details to give a full description of a programming language. Generally this is based on the use of *attributes*, the name given to special values associated with (certain) terminal and non-terminal symbols in the grammar. The rules of the grammar are then extended with additional rules governing the relationships between the attribute values of the symbols in the grammar rule. In yacc these rules are expressed by using constructs from the programming language C, and evaluating these as the parse takes place. More sophisticated parser generators incorporate a special language for attributes, whose evaluation phase, often by necessity, takes place separately from parsing.

It is common to distinguish two types of attribute - *synthesised* attributes which involve transferring information from the rightpart of a rule to the non-terminal on the left, and *inherited* attributes which flow in the opposite direction. As might be expected, the choice of attribute type can be strongly influenced by the parsing strategy.

Since the main task of the semantic rules in an attributed grammar will be to control the values of those attributes, we will assume that all attributes are in equational

form. We will not impose any restriction on the flow of values around the grammar, since this would be inconsistent with the parse-strategy-independent view of grammars taken previously. Also, for simplicity, (and based on lemmas proved earlier) we will assume that the rightparts of rules do not use the union or Kleene closure operations.

We will define the institution of attributed grammars by expanding $\mathcal{CFREE}$ with attribute-handling capabilities. In general terms this means that sentences will contain attribute evaluation information as well as grammar rules, and that models can refer to values as well as sentences.

4.2.1 Basic Definitions

We can now formally define the components of $\mathcal{ATG}$, the institution of attributed grammars.

Definition 4.1 *Signatures in* $\mathcal{ATG}$

A signature Σ *consists of four sets:*

1. Σ_S, *a set of sort names,*
2. Σ_O *a set of operators indexed by sort-sequences*
3. Σ_T *a set of terminal symbols,*
4. Σ_N *a set of sort-indexed non-terminal symbols*

The purpose of the sort-index of each non-terminal symbol is to give the type of the attribute-values that can be associated with that symbol. The operators form a very simple algebraic-style language for forming expressions over the attributes. In order to give full meaning to these operators, we would envisage them being constrained by theories from some suitably more powerful institution, such as e.g. $\mathcal{EQ}$.

Morphisms are the four-fold products of set morphisms, with the conditions that for any such morphism σ, the indexing is preserved, i.e. :

- For any operator $op : S_1 \times \ldots \times S_n \rightarrow S$, we insist that $\sigma(op)$ is an operator of sort $\sigma(S_1) \times \ldots \times \sigma(S_n) \rightarrow \sigma(S)$
- If non-terminal symbol N is of sort S, then non-terminal $\sigma(N)$ is of sort $\sigma(S)$

To construct a model of an attributed signature will will again extend the context-free case – this time we need to allow for the presence of attributes. One option would be to use parse trees whose non-leaf nodes have been annotated with attribute values in a manner consistent with the grammar rules. However, as before, we will try to avoid the use of parse trees and look for less "operational" models.

Thus we will choose to model non-terminals as sets whose elements are pairs of the form $\langle s, v \rangle$, where s is a string, and v is some value, sort-consistent with the attribute-type of the non-terminal. The interpretation of this is that, based on the grammar, the non-terminal can derive the string s and, when it does, the resulting attribute value associated with it will be v. Despite appearances, this should not be seen as a necessarily "bottom-up" approach, since the application of the attribute equations will take place (in both directions) later when we define satisfaction for rules.

We can now define a model of an attributed signature as consisting of six components:

Definition 4.2 *Models in $\mathcal{ATG}$*

- The attribute part, consisting of

1. *A function I_S mapping sort names from Σ to sets*
2. *A function I_O mapping operator symbols into functions over the appropriate sets (operators of arity zero being mapped to constants).*

- The context-free part, consisting of

3. *Some countable set I_C of characters*
4. *A function I_T which, which will map any terminal symbol into a subset of I_C*

5. *A function I_N which, which will map any non-terminal symbol of the form $A^{:S}$ into a set whose elements are of the form $\langle s, v \rangle$ where $s \in I_C^*$ and $v \in [\![S]\!]$.*

- The actual language defined by the model

6. *Some set I_L whose elements are all of the form $\langle s, v \rangle$, being the language and evaluation defined by the grammar.*

We note that the images of I_N and I_L are relations between strings and values - they are not necessarily functional in either direction. It is quite possible that different derivations would cause different values to be associated with a non-terminal for the same string (this would indicate that grammars corresponding to the model can be ambiguous).

Since not every non-terminal need have an attribute, I_N could be seen as a partial function. However we shall not pursue this issue here; we suggest that such partiality can be dealt with by whatever formalism the attribute equations will ultimately be mapped into. Where necessary, we shall denote the lack of an attribute by using the value $\perp$.

It is common to have a number of different attributes, whether synthesised or inherited associated with a grammar symbol, representing different semantic features. Even though we have only allowed one specific variable in the above description, it is still possible for the sort of this variable to be a product, thus allowing for arbitrarily many components of a variable to be considered (in the manner of fields in a record). Thus we have not lost any generality in this respect from the standard definition.

Given a signature morphism $\sigma: \Sigma' \to \Sigma$, we can construct a model of Σ' by composing σ with each of I_S, I_O, I_T, and I_N, and by leaving I_C and I_L the same.

To construct sentences in the institution we will need to assume some ordered set of attribute variables, which we will denote $\{\$\$, \$1, \$2, \ldots\}$. The attribute variable

$$ will denote the value of the attribute associated with the left-hand-side of the grammar rule; one of the form $\$i$ refers to attribute associated with the i^{th} symbol on the right-hand-side of the rule.

We will use the term *attribute expression* to describe any well-typed expression formed from the operator symbols and the attribute variables. (For the moment, we assume that "well-typed" means "consistently-typed", with type inferencing begin used for the variables). Given a model I, and some sequence of (correctly typed) values to substitute in for the variables, we can evaluate an expression in this context. Formally, for any expression η, we define its evaluation $EVAL_I(\eta)$ as:

Definition 4.3 *Evaluating an attribute expression*

For some $n \in \mathbb{N}$, and any sequence of values $\tilde{a}$

$$\begin{aligned} EVAL_I(\$\$, \tilde{a}) &= \pi^1(\tilde{a}) \\ EVAL_I(\$i, \tilde{a}) &= \pi^{i+1}(\tilde{a}) \\ EVAL_I(op(\eta_1, \ldots, \eta_n), \tilde{a}) &= I_O(op)(EVAL_I(\eta_1, \tilde{a}), \ldots, EVAL_I(\eta_n, \tilde{a})) \end{aligned}$$

where op is an operator, and $\eta_1, \ldots, \eta_n$ are attribute expressions

We note at this point that signature morphisms can be extended to attribute expressions (componentwise), and that these interact with evaluation in a natural way:

Lemma 4.4 *For any attribute expression η over a signature Σ, any correctly typed sequence of values $\tilde{a}$, any model I, and any signature morphism σ into Σ,*

$$EVAL_I(\sigma(\eta), \tilde{a}) = EVAL_{\sigma;I}(\eta, \tilde{a})$$

Proof: *By induction over the size of an expression*

We consider the two cases from the previous definition:

1. $$\begin{aligned} EVAL_I(\sigma(\$\$), \tilde{a}) &= EVAL_I(\$\$, \tilde{a}) \\ &= \pi^i(\tilde{a}) \\ &= EVAL_{\sigma;I}(\$\$, \tilde{a}) \end{aligned}$$

2. $$\begin{aligned} EVAL_I(\sigma(\$i), \tilde{a}) &= EVAL_I(\$i, \tilde{a}) \\ &= \pi^{i+1}(\tilde{a}) \\ &= EVAL_{\sigma;I}(\$i, \tilde{a}) \end{aligned}$$

3. $$\begin{aligned} EVAL_I(\sigma(op(\eta_1, \ldots, \eta_n)), \tilde{a}) &= EVAL_I(\sigma(op)(\sigma(\eta_1), \ldots, \sigma(\eta_n)), \tilde{a}) \\ &= I_O(\sigma(op))(EVAL_I(\sigma(\eta_1), \tilde{a}), \ldots, EVAL_I(\sigma(\eta_n), \tilde{a})) \\ &= I_O(\sigma(op))(EVAL_{\sigma;I}(\eta_1, \tilde{a}), \ldots, EVAL_{\sigma;I}(\eta_n, \tilde{a})) \\ &= EVAL_{\sigma;I}((op(\eta_1, \ldots, \eta_n)), \tilde{a}) \end{aligned}$$

□

Sentences in the institution will be context-free sentences extended with equations over attribute expressions:

Definition 4.5 *Sentences in $\mathcal{ATG}$*

Sentences are of the form:

$$\begin{array}{lll} 1. & (A \to r_1 \ldots r_n) & \{p_1 = q_1, \ldots, p_m = q_m\} \\ & & \textit{or just} \\ 2. & r_1 \ldots r_n & \{p_1 = q_1, \ldots, p_m = q_m\} \end{array}$$

where A is a non-terminal, each r_i is a terminal or non-terminal, $m \geq 0$, and the p_j and q_j are attribute-expressions which can contain a variable $\$i$ only if the corresponding grammar symbol occurs at position i in the rule (with A being at position 0).

Signature morphisms can be extended to sentence morphisms componentwise in the expected way, acting as the identity on the attrbiute variables.

In order to deal with rightparts in a uniform way, we will define an simple auxiliary function. For any given signature, any model I, and any single grammar symbol γ, we can define $ATT_I(\gamma)$ to be the set of attributions for γ, each element of which consists of a string and an attribute value.

Definition 4.6 *$ATT_I(\gamma)$, the attribution associated with γ*

$$\begin{aligned} & ATT_I(\Lambda) &=& \{\langle \varepsilon_{I_C}, \perp \rangle\} \\ \forall\, a \in \Sigma_T \cdot \quad & ATT_I(\text{`}a\text{'}) &=& \{\langle \text{``}c\text{''}, \perp \rangle \mid c \in I_T(a)\} \\ \forall\, A \in \Sigma_N \cdot \quad & ATT_I(\text{`}A\text{'}) &=& I_N(A) \end{aligned}$$

We note that ATT_I commutes with signature morphisms, that is we assert that $ATT_I(\sigma(r)) = ATT_{\sigma;I}(r)$.

Now we are ready to define satisfaction in the institution; this will have two components: the language part, and the attribute part:

Definition 4.7 *Satisfaction in $\mathcal{ATG}$*

Satisfaction is defined as follows:

$$I \quad \models_\Sigma \quad (A \rightarrow r_1 \ldots r_n)\ \{p_1 = q_1, \ldots, p_m = q_m\}$$

$$\Leftrightarrow$$

$$\forall \langle s_1, v_1 \rangle \in ATT_I(r_1), \ldots, \forall \langle s_n, v_n \rangle \in ATT_I(r_n) \cdot \exists \langle s_0, v_0 \rangle \in ATT_I(A) \cdot$$

$$s_1 \cdots s_n = s_0 \quad \wedge$$

$$\bigwedge_{j=1}^{j=m} EVAL_I(p_j, \langle v_0, v_1, \ldots, v_n \rangle) =_{I_S(s_j)} EVAL_I(q_j, \langle v_0 v_1, \ldots, v_n \rangle)$$

or just

$$I \quad \models_\Sigma \quad r_1 \ldots r_n\ \{p_1 = q_1, \ldots, p_m = q_m\}$$

$$\Leftrightarrow$$

$$\forall \langle s_1, v_1 \rangle \in ATT_I(r_1), \ldots, \forall \langle s_n, v_n \rangle \in ATT_I(r_n) \cdot \exists \langle s_0, v_0 \rangle \in I_L \cdot$$

$$s_1 \cdots s_n = s_0 \quad \wedge$$

$$\bigwedge_{j=1}^{j=m} EVAL_I(p_j, \langle v_0, v_1, \ldots, v_n \rangle) =_{I_S(s_j)} EVAL_I(q_j, \langle v_0 v_1, \ldots, v_n \rangle)$$

where we assume that S_j is the sort associated with p_j and q_j, and "$\_ =_{I_S(s_j)} \_$" denotes the (strict[1]) identity relation over this set. (In the case where $m = 0$ we assume that the equation part is trivially satisfied).

The satisfaction condition follows from the definitions of $EVAL_I$ and ATT_I and their commutation with σ; we state:

Lemma 4.8 *Satisfaction Condition for $\mathcal{ATG}$*

For any signatures Σ' and Σ, any signature morphism $\sigma: \Sigma' \to \Sigma$, any Σ-model I, and Σ'-sentence s,

$$I \models_\Sigma [\sigma]s \qquad \Leftrightarrow \qquad [\![\sigma]\!]I \models_{\Sigma'} s$$

Proof:

The proof breaks into two cases; we shall just prove the first, as the second is almost identical.

$$I \quad \models_\Sigma \quad (\sigma(A) \to \sigma(r_1) \ldots \sigma(r_n))\ \{\sigma(p_1) = \sigma(q_1), \ldots, \sigma(p_m) = \sigma(q_m)\}$$

$$\Leftrightarrow$$

$$\forall \langle s_1, v_1 \rangle \in ATT_I(\sigma(r_1)), \ldots, \forall \langle s_n, v_n \rangle \in ATT_I(\sigma(r_n)) \cdot \exists \langle s_0, v_0 \rangle \in ATT_I(\sigma(A)) \cdot$$
$$s_1 \cdots s_n = s_0 \qquad \wedge$$
$$\bigwedge_{j=1}^{j=m} EVAL_I(\sigma(p_j), \langle v_0, v_1, \ldots, v_n \rangle) =_{\sigma;\, I_S(s_j)} EVAL_I(\sigma(q_j), \langle v_0, v_1, \ldots, v_n \rangle)$$

$$\Leftrightarrow$$

$$\forall \langle s_1, v_1 \rangle \in ATT_{\sigma;\, I}(r_1), \ldots, \forall \langle s_n, v_n \rangle \in ATT_{\sigma;\, I}(r_n) \cdot \exists \langle s_0, v_0 \rangle \in ATT_{\sigma;\, I}(A) \cdot$$
$$s_1 \cdots s_n = s_0 \qquad \wedge$$
$$\bigwedge_{j=1}^{j=m} EVAL_{\sigma;\, I}(p_j, \langle v_0, v_1, \ldots, v_n \rangle) =_{\sigma;\, I_S(s_j)} EVAL_{\sigma;\, I}(q_j, \langle v_0, v_1, \ldots, v_n \rangle)$$

$$\Leftrightarrow$$

$$[\![\sigma]\!]I \quad \models_{\Sigma'} \quad (A \to r_1 \ldots r_n)\ \{p_1 = q_1, \ldots, p_m = q_m\}$$

□

[1]That is, we assume the operation fails if any of the components involved are $\perp$

4.2.2 Properties of the Attribute Institution

Whereas with the syntactic institutions we were able to pick out a distinguished canonical model (the minimal one in each case), we cannot necessarily do so here. The reason for this is that the sorts in Σ_S cannot be fully specified within the formalism, and a minimal interpretation here would omit much of the intuitively expected structure. Thus the specifications here are genuinely "loose", and, as mentioned, we would expect to tie them down by constraints from other insitutions.

By direct substitution of their definitions, we can immediately prove some straightforward results such as:

$$\frac{I \models (A \rightarrow r)\{p_1 = q_1\}, \qquad I \models (A \rightarrow r)\{p_2 = q_2\}}{I \models (A \rightarrow r)\{p_1 = q_1, p_2 = q_2\}}$$

and intuitively-correct results such as:

$$\frac{I \models (A \rightarrow x.\text{`}B\text{'}.z)\{\$\$ = \$i\}, \qquad I \models (B \rightarrow y)\{\$\$ = c\}}{I \models (A \rightarrow x.y.z)\{\$\$ = c\}}$$

(where we assume that the non-terminal B is at position i in $x.\text{`}B\text{'}.z$)

Once again co-limits of signatures stem from set-theoretic union, so we are free to sum, parameterise and constrain attribute grammars.

4.3 Relating Attribute and Context-Free Grammars

As we have seen, the attribute grammar is built "on top of" a context-free grammar. However, in certain circumstances, it may be desirable to keep the actual description of the context-free syntax of the language separate from the semantics, perhaps because

- The context-free grammar is more "concrete" than the grammar on which the attributes are built, containing low-level details about the program text which are irrelevant at the semantic stage

- The attribute grammar describes constructs at a level of granularity which is not required to give a valid syntactical description of the language

- It is intended to implmement the parsing and attribution processes separately using different algorithms, and this is reflected at the specification level in both

- The attribute grammar takes care of a self-contained part of the anaysis process, and we wish to limit its interaction with other parts of the specification; any links will be made via the context-free grammar

For whichever of the above reasons, we envisage specifying the context-free syntax, and then making (parts of) this subject to additional specification from the attribute grammar. To do this we will construct an institution morphism from $\mathcal{ATG}$ back into $\mathcal{CFREE}$, and seek to restrain context-free grammar modules by sections of the attribute grammar.

To make things slightly easier, we will assume that the right-hand sides of context-free grammars involve only the union of concatenated vocabulary symbols. This does not in any way restrict the power of specification of the grammar (and indeed could be formally specified by using an institution mapping from $\mathcal{CFREE}$ to itself).

Now we can define:

Definition 4.9 *The institution morphism AGtoCF:* $\mathcal{ATG} \Rightarrow \mathcal{CFREE}$

We define the three components as:

1. *The functor* Φ: $\mathbf{Sign}_{\mathcal{ATG}} \rightarrow \mathbf{Sign}_{\mathcal{CFREE}}$ *which takes any attribute signature of the form* $\langle \Sigma_S, \Sigma_O, \Sigma_T, \Sigma_N \rangle$ *and "loses" the sorts and operators giving the context-free signature* $\langle \Sigma_T, \Sigma_N \rangle$

2. *The natural transformation* α: $(\Phi \fatsemi Sen_{\mathcal{CFREE}}) \Rightarrow Sen_{\mathcal{ATG}}$ *takes any context-free rightpart or grammar rule and promotes it directly to the attribute institution (without change), since this is just an attribute grammar rule whose equations are trivially satisfied*

3. *The natural transformation* β: $Mod_{\mathcal{ATG}} \Rightarrow (\Phi \fatsemi Mod_{\mathcal{CFREE}})$ *takes an attribute-grammar-model of the form* $\langle I_S, I_O, I_C, I_T, I_N, I_L \rangle$ *and sends it to the context-free model* $\langle I_C, I_T, J_N, J_L \rangle$, *where* J_N *and* J_L *are the result of removing the value-component from the pairs in* I_N *and* I_L *respectively*

Since there has been so little actual change to the sentences and models (other than simplification), verification that this is in fact an institution morphism causes no problems:

Lemma 4.10

AGtoCF: $\mathcal{ATG} \Rightarrow \mathcal{CFREE}$, *as defined above, is an institution morphism*

Proof:

We must show that for every $\mathcal{CFREE}$-signature Σ, every context-free sentence of the form $(A \rightarrow r)$ in $[\Sigma]$ (sentences involving only rightparts follow the obvious simplification), and every attributed model I in $[\![\Phi(\Sigma)]\!]$ we have:

$$\langle I_S, I_O, I_C, I_T, I_N, I_L \rangle \models_{\Phi(\Sigma)} \alpha(A \rightarrow r) \quad \Leftrightarrow \quad \beta\langle I_S, I_O, I_C, I_T, I_N, I_L \rangle \models_{\Sigma} (A \rightarrow r)$$

By the definition of α and β this becomes:

$$\langle I_S, I_O, I_C, I_T, I_N, I_L \rangle \models_{\Phi(\Sigma)} A \rightarrow r\ \{\} \quad \Leftrightarrow \quad \langle I_C, I_T, J_N, J_L \rangle \models_{\Sigma} (A \rightarrow r)$$

(with J_T and J_N as earlier), which is easily seen to be true, since the attribute-grammar rule makes no use of the actual attribute values.

□

Thus, given two specifications $\langle \Sigma_C, S_C \rangle$ from $\mathcal{CFREE}$ and $\langle \Sigma_A, S_A \rangle$ from $\mathcal{ATG}$ we would envisage adding a sentence of the form

$$\textsf{Restrain } \Sigma_C \textsf{ by } \langle \langle \Sigma_A, S_A \rangle, \theta \rangle \textsf{ via } AGtoCF$$

into the context-free specification to make sure that the syntactically-correct models of $\langle \Sigma_C, S_C \rangle$ were also semantically correct.

Here, the morphism θ: $\Phi(\Sigma_A) \rightarrow \Sigma_C$ has two purposes:

2. The "upper"-level rules are called *meta-rules*, and their purpose is to define valid arguments which may be substituted in for the meta-notions. This is achieved by presenting a context-free grammar in which the meta-notions are the non-terminals, and correspond to languages of proto-notions.

Thus a context-free rule can be derived from a van-W grammar by taking any hyper-rule and substituting the meta-notions with one of their corresponding string of proto-notions, as defined by the meta-rules. This must work as for a parameterisation, in that the substitution for a particular meta-notion must be uniform within a given hyper-rule.

As a notion sequence may eventually correspond to either a terminal or non-terminal grammar symbol when regarded as a unit, it is usual to distinguish those which correspond to terminals by appending the word "`symbol`" to them. We will generalise this slightly and assume that some arbitrary syntactic differentiation is possible.

4.4.1 Basic Definitions

With this in mind, we are ready to define the format of a signature in the institution $\mathcal{VANW}$ of van-W grammars:

Definition 4.11 *Signatures in* $\mathcal{VANW}$

Any object in the category of signature from $\mathcal{VANW}$ *has four components:*

1. Σ_M, *the (finite) set of meta-notions*
2. Σ_P, *the (finite) set of proto-notions*
   Thus a "notion-sequence" is any element of $(\Sigma_M \cup \Sigma_P)^+$
3. *A finite set* $\Sigma_T \subseteq (\Sigma_M \cup \Sigma_P)^+$, *disjoint from* Σ_N, *consisting of those notion-sequences which can correspond to terminal symbols*
4. *A finite set* $\Sigma_N \subseteq (\Sigma_M \cup \Sigma_P)^+$, *consisting of those notion-sequences which can correspond to non-terminal symbols*

This definition may appear to be somewhat more restrictive than necessary, since we might just have specified Σ_M and Σ_P, along with some predicate over notion-sequences which recognises (non-)terminal grammar symbols. However, the above definition will make the process of constructing mappings into other formalisms considerably easier.

We shall choose to write any notion sequence n from either Σ_N or Σ_T as though it were indexed by the elements of Σ_M thus: $n^{:m_1,\ldots,m_k}$, where each $m_i \in \Sigma_M$ occurs somewhere in n. Note that this indexing is derived, and that all such notion-sequences n are unique, independently of their indices.

A signature morphism $\sigma: \Sigma' \to \Sigma$ consists of any pair of set theoretic morphisms $\sigma_M: \Sigma'_M \to \Sigma_M$ and $\sigma_A: \Sigma'_P \to \Sigma_A$, such that their extension to strings in the usual way maintains the division between terminals and non-terminals i.e. $\sigma(\Sigma_N) \cap \sigma(\Sigma_T) = \varnothing$. By its definition, this extension preserves the indexing on the notion-sequences in a homomorphic way.

When defining the functor $[\![\cdot]\!]$, a little care is required in dealing with the proto-notions, since they play a role in both the syntactic and semantic aspect of the grammar. As we have seen, we will use these symbols to build up hyper-rules, which are sentences. However, the language defined by the meta-rules will consist of strings over these symbols, so they may also be regarded as components of the model. It is common in denotational definitions to gloss over such differences, but it will serve us well to be pedantic here, since model-morphisms induced by signature morphisms will change Σ_P, but will also be expected to preserve the basic structure of the model. The net result of all this is the inclusion of an alphabet in the model into which the proto-notions are interpreted.

The other components of the model will involve interpreting each meta-notion as a language over proto-notions. We will keep with the "parameter" analogy, and interpret the notion-sequences as functions which, when provided with arguments of

the appropriate type, will yield languages (for non-terminals) or characters (for terminals). As usual, we shall also provide form some set which defines "the" language described by the model.

Definition 4.12 *Models in $\mathcal{VANW}$*

Given any signature Σ, a model I of Σ consists of two "levels":

- *The meta-level:*

1. *A finite set of proto-notion "alphabet characters", I_P*
2. *A mapping, I_A, sending each element of Σ_P to an element of I_P.*
3. *A mapping, I_M, sending each element of Σ_M to a language over I_P*

- *The hyper-level:*

4. *A finite set of characters, I_C*
5. *A mapping, I_T, sending each element from Σ_T of the form $t^{:m_1,\ldots,m_k}$ into some function of type $(\Sigma_M(m_1) \times \ldots \times \Sigma_M(m_n) \rightarrow \wp(I_C))$ yielding (parameterised) sets of characters*
6. *A mapping, I_N, sending each element from Σ_N of the form $n^{:m_1,\ldots,m_k}$ into some function of type $(\Sigma_M(m_1) \times \ldots \times \Sigma_M(m_n) \rightarrow \wp(I_C^*))$, yielding (parameterised) languages*
7. *A language, I_L, over the alphabet I_C, being "the" language defined by the grammar*

For some fixed signature Σ, we can construct a morphism between Σ-models straightforwardly by taking set-theoretic morphisms with its components.

Given any signature morphism $\langle \sigma_M, \sigma_A \rangle$ as above, and any Σ-model I, we define:

$$[\![\sigma]\!]\langle I_P, I_A, I_M, I_C, I_T, I_N, I_L\rangle = \langle I_P, (\sigma_P;\ I_A), (\sigma_M;\ I_M), I_C, (\sigma;\ I_T), (\sigma;\ I_N), I_L\rangle$$

We note that this implies, for example,

$$\forall\, n^{:m_1,\ldots,m_k} \in \Sigma_N \ \cdot\ ([\![\sigma]\!] I_N)(n^{:m_1,\ldots,m_k}) = I_N(\sigma(n)^{:\sigma(m_1),\ldots,\sigma(m_k)})$$

which has type $(\Sigma_M(\sigma(m_1)) \times \ldots \times \Sigma_M(\sigma(m_n)) \rightarrow \wp(I_C^*))$

As we have stated, sentences in the institution will consist of meta-rules and hyper-rules. It is usual with van-W grammars to use different notation for concatenation and union in each of these rule-sets. While this is not strictly necessary (and indeed is somewhat of a syntactic burden), we will adopt it here for ease of reference. The following table gives the standard symbols we have been using in context-free rules, and the corresponding symbols for the van-W rules: [2]

Context-Free	$\rightarrow$	.	|
Meta-Rule	::	.	;
Hyper-Rule	:	,	;

We note that since the hyper-rules are dealing with concepts *defined* by the meta-rules - strings over the proto-notions - the syntax of the hyper-rules will use elements from the semantics of the meta-rules. Thus there will in fact be two versions of concatenation on the right-hand-side of a hyper-rule: the syntactic ",", and the semantic ".". (We did not have to "look closely enough" to see the latter when dealing with ordinary context-free grammars.)

We can now give a formal definition of meta- and hyper-rules. The meta-rules are, for all practical purposes, identical to ordinary context-free grammar rules:

Definition 4.13 *Format of meta-rules*

Given any signature $\Sigma = \langle \Sigma_M, \Sigma_A, \Sigma_T \rangle$, *a meta-rule over* Σ *is of the form:*

$$`m' :: MR.$$

where $m \in \Sigma_M$, *and MR is a meta-rightpart, defined as:*

1. Λ *is a meta-rightpart*

2. *For any* $a \in \Sigma_A$, *'a' is a meta-rightpart*

[2] As with context-free rules, concatenation is usually denoted by juxtaposition in meta-rules: here we will explicitly represent this by a "."

3. *For any* $m \in \Sigma_M$, *'m' is a meta-rightpart*

4. *If* m_1 *and* m_2 *are meta-rightparts, then so are* $m_1.m_2$ *and* $m_1;\ m_2$

Hyper-rules are similar to context-free grammars where Σ_N and Σ_T are used as the vocabularly. For simplicity here we will restrict the format of right-hand sides to the simplest possible, and not allow arbitrary mixing of union and concatention.

Definition 4.14 *Format of hyper-rules*

Given any signature Σ, *a hyper-rule over* Σ *is of the form:*

$$`p' : HR.$$

where $p \in \Sigma_N$, *and HR is a hyper-rightpart, defined as:*

1. Λ *is a hyper-rightpart*

2. *Given any finite set of notion-sequences* $\{n_1, \ldots, n_k\} \subseteq (\Sigma_N \cup \Sigma_T)$, *then* $`n_1', \ldots, `n_k'$ *is a hyper-rightpart*

3. *If* h_1 *and* h_2 *are hyper-rightparts, then so is* $h_1;\ h_2$

As with the context-free institution, we will allow hyper-rightparts to act as "axioms", collectively defining the start symbol. In this case, a collection of hyper-rightparts can define a (possibly infinite) set of start symbols.

Definition 4.15 *Sentences in* $\mathcal{VANW}$

Given a signature Σ *from* $\mathcal{VANW}$, *the set of sentences* $[\Sigma]$ *may be partitioned into three subsets:*

- $[\Sigma]_M$, *a set of meta-rules*
- $[\Sigma]_H$, *a set of hyper-rules*
- $[\Sigma]_{HR}$, *a set of hyper-rightparts.*

We assert that with Σ_P as terminals and Σ_M as non-terminals, we can formulate the same auxiliary definitions (including a version of LAN_I) and independently verify the satisfaction condition for meta-rules as we did for the context-free institution. We will not deal with this in any more detail here.

Hyper-rules are evaluated in the context of the definitions of the meta-notions. Since these are effectively free variables, we must have some concept of an "environment" (or "value-assignment") which will allow us to plug in values, and evaluate notion sequences. Thus, given any set of meta-notions $M \subseteq \Sigma_M$, and any Σ-model I as above, we define an I-consistent *environment* for M as:

$$ENV_I(M) \quad = \quad \{e: M \to I_P^* \mid \forall\, m \in M \cdot e(m) \in I_M(m)\}$$

In the context of these environments, it is now possible to define a language (over I_C) which will correspond to a (terminal or non-terminal) notion sequence. Since I_M can yield an arbitrarily large language for any meta-notion, the number of languages over I_C corresponding to a notion-sequence may be infinite.

Definition 4.16 *$LAN_I(n)$, the language associated with a notion-sequence*

Given any signature Σ and any Σ-model I, we can define the function LAN_I to evaluate a notion-sequence from Σ_N or Σ_T in the context of an appropriate environment as follows:

1. $$\forall\, n^{:m_1,\ldots,m_k} \in \Sigma_N \cdot \forall\, e \in ENV_I(\{m_1,\ldots,m_k\}) \cdot \\ LAN_I(n)(e) = I_N(n(e(m_1),\ldots,e(m_k)))$$

2. $$\forall\, t^{:m_1,\ldots,m_k} \in \Sigma_T \cdot \forall\, e \in ENV_I(\{m_1,\ldots,m_k\}) \cdot \\ LAN_I(t)(e) = \{\text{“}c(e(m_1),\ldots,e(m_k))\text{”} \mid c \in I_T(t)\}$$

This can, of course, be extended to hyper-rightparts by interpreting the concatenation and union operators in the usual manner.

The important result here is that there is a method of changing environments with a signature change that preserves the language defined by a notion-sequence; indeed, this can take place in either direction.

Lemma 4.17

For any signature morphism $\sigma: \Sigma' \to \Sigma$, *any (terminal or non-terminal) notion-sequence of the form* $n'^{:m'_1,\ldots,m'_k}$ *from* Σ', *we have that:*

$$\forall\, e \in ENV_I(\{\sigma(m'_1),\ldots,\sigma(m'_k)\}) \cdot LAN_{[\sigma](I)}(n')(\sigma;\ e) = LAN_I(\sigma(n'))(e)$$

Proof:

The proof follows directly from the definition of LAN_I. □

Satisfaction is defined as for the context-free case.

Definition 4.18 *Satisfaction in* $\mathcal{VANW}$

For any signature Σ, *and any* Σ*-model* I, *we consider the three possibilities for an element of* $[\Sigma]$*:*

1. *The sentence is a meta-rule:*

$$I \models \text{`}m\text{'} :: MR. \quad \Leftrightarrow \quad LAN_I(MR) \subseteq LAN_I(\text{`}m\text{'})$$

2. *The sentence is a hyper-rule; then:*

$$\begin{array}{l}
(a)\ I \models n_0 : N_1; \ldots; N_k. \quad \Leftrightarrow \\
\qquad I \models n_0 : N_1. \ \wedge \ldots \wedge \ I \models n_0 : N_k. \\
(b)\ I \models n_0^{:M_0} : n_1^{:M_1}, \ldots, n_k^{:M_k}. \quad \Leftrightarrow \\
\qquad \forall\, e \in ENV_I(M_0 \cup M_1 \cup \ldots \cup M_k) \cdot \\
\qquad\quad LAN_I(n_1^{:M_1})(e) \frown \ldots \frown LAN_I(n_k^{:M_k})(e) \subseteq LAN_I(n_0^{:M_0})(e) \\
(c)\ I \models \Lambda \quad \Leftrightarrow \quad \forall\, e \in ENV_I(M_0) \cdot \varepsilon_{I_C} \in LAN_I(n_0^{:M_0})(e)
\end{array}$$

3. *The sentence is a hyper-rightpart; then*

$$\begin{array}{l}
(a)\ I \models N_1; \ldots; N_k \quad \Leftrightarrow \\
\qquad I \models N_1 \ \wedge \ldots \wedge \ I \models N_k \\
(b)\ I \models n_1^{:M_1}, \ldots, n_k^{:M_k} \quad \Leftrightarrow \\
\qquad \forall\, e \in ENV_I(M_1 \cup \ldots \cup M_k) \cdot \\
\qquad\quad LAN_I(n_1^{:M_1})(e) \frown \ldots \frown LAN_I(n_k^{:M_k})(e) \subseteq I_L \\
(c)\ I \models \Lambda \quad \Leftrightarrow \quad \varepsilon_{I_C} \in I_L
\end{array}$$

Notice that in the definition of satisfaction for the hyper-rules, we take an environment over the union of the meta-notions on the right- and left-hand side. This is

the "consistent substitution" rule associated with van-W grammars, which requires that each meta-notion be substitued uniformly accross the rule before it is applied.

Lastly, the work done to date establishes the following:

Lemma 4.19 *Satisfaction Condition for $\mathcal{VANW}$*

For any signature morphism $\sigma: \Sigma' \rightarrow \Sigma$, any Σ-model I, and Σ'-sentence s,

$$I \models_\Sigma [\sigma]s \qquad \Leftrightarrow \qquad [\![\sigma]\!]I \models_{\Sigma'} s$$

Proof: *(Sketch)*
The proof for the first case follows from $\mathcal{CFREE}$, and that for the third case closely resembles that for the second.

Expanded out, the satsifaction condition for hyper-rules whose right-hand-sides consists of notion-sequences is:

$$\begin{array}{l}\forall\, e \in ENV_I(\sigma(M_0) \cup \sigma(M_1) \cup \ldots \cup \sigma(M_k)) \cdot \\ \quad LAN_I(\sigma(n_1)^{:\sigma(M_1)})(e) \frown \ldots \frown LAN_I(\sigma(n_k)^{:\sigma(M_k)})(e) \subseteq LAN_I(\sigma(n_0)^{:\sigma(M_0)})(e)\end{array}$$

$$\Leftrightarrow$$

$$\begin{array}{l}\forall\, e \in ENV_{[\![\sigma]\!](I)}(M_0 \cup M_1 \cup \ldots \cup M_k) \cdot \\ \quad LAN_{[\![\sigma]\!](I)}(n_1^{:M_1})(e) \frown \ldots \frown LAN_{[\![\sigma]\!](I)}(n_k^{:M_k})(e) \subseteq LAN_{[\![\sigma]\!](I)}(n_0^{:M_0})(e)\end{array}$$

To prove this by contradiction, we negate the statement, yielding the disjunction:

$$\begin{array}{l}(\forall\, e \in ENV_I(\sigma(M_0) \cup \sigma(M_1) \cup \ldots \cup \sigma(M_k)) \cdot \\ \quad LAN_I(\sigma(n_1)^{:\sigma(M_1)})(e) \frown \ldots \frown LAN_I(\sigma(n_k)^{:\sigma(M_k)})(e) \subseteq LAN_I(\sigma(n_0)^{:\sigma(M_0)})(e)\end{array}$$

$$\wedge$$

$$\begin{array}{l}\exists\, e \in ENV_{[\![\sigma]\!](I)}(M_0 \cup M_1 \cup \ldots \cup M_k) \cdot \\ \quad LAN_{[\![\sigma]\!](I)}(n_1^{:M_1})(e) \frown \ldots \frown LAN_{[\![\sigma]\!](I)}(n_k^{:M_k})(e) \not\subseteq LAN_{[\![\sigma]\!](I)}(n_0^{:M_0})(e))\end{array}$$

$$\vee$$

$$\begin{array}{l}(\forall\, e \in ENV_{[\![\sigma]\!](I)}(M_0 \cup M_1 \cup \ldots \cup M_k) \cdot \\ \quad LAN_{[\![\sigma]\!](I)}(n_1^{:M_1})(e) \frown \ldots \frown LAN_{[\![\sigma]\!](I)}(n_k^{:M_k})(e) \subseteq LAN_{[\![\sigma]\!](I)}(n_0^{:M_0})(e)\end{array}$$

$$\wedge$$

$$\exists\, e \in ENV_I(\sigma(M_0) \cup \sigma(M_1) \cup \ldots \cup \sigma(M_k)) \cdot$$
$$LAN_I(\sigma(n_1)^{:\sigma(M_1)})(e) \frown \ldots \frown LAN_I(\sigma(n_k)^{:\sigma(M_k)})(e) \not\subseteq LAN_I(\sigma(n_0)^{:\sigma(M_0)})(e))$$

Both components of this are seen to be falsified by the previous lemma, thus proving the original hypothesis. □

4.5 Relating van-W and Context-Free Grammars

For much the same reasons as for attribute grammars we might want to relate van-W grammars back down to the corresponding syntactic descriptions in $\mathcal{CFREE}$. The procedure we will use here is essentially the same as that for attribute grammars, with the only major difference being the fact that we cannot distinguish the "semantic" parts of a van-W grammar as easily. Thus we will make an arbitrary abstraction, where all of the syntactic and semantic details contained in a notion-sequence are bundled into a single context-free vocabulary symbol. The mappings for signatures and models are once again of the "forgetful" type, in that their purpose is to lose semantic details.

The first two components of the morphism can be defined easily; thus we state:

Definition 4.20 *The institution morphism* $VWtoCF: \mathcal{VANW} \Rightarrow \mathcal{CFREE}$
We define the three components as:

1. *The functor* Φ: $\mathbf{Sign}_{\mathcal{VANW}} \rightarrow \mathbf{Sign}_{\mathcal{CFREE}}$ *which takes a van-W signature of the form* $\langle \Sigma_M, \Sigma_P, \Sigma_T, \Sigma_N \rangle$ *and forgets about the internals of the notion-sequences, giving the context-free signature* $\langle \Sigma_T, \Sigma_N \rangle$, *where the elements of each set are now non-decomposable.*

2. *The natural transformation* α: $(\Phi \fatsemi Sen_{\mathcal{CFREE}}) \Rightarrow Sen_{\mathcal{VANW}}$ *maps context-free grammar rules into hyper-rules; that is, a context-free sentence of the form* $(A \rightarrow r)$ *is mapped to* $A : r.$, *with "," and ";" being used in place of "." and "|"*

3. *The natural transformation* $\beta: Mod_{VANW} \Rightarrow (\Phi \fatsemi Mod_{CFREE})$ *takes a van-W model and merges together the sets of languages corresponding to the notion sequences. Thus given a van-W model of the form* $\langle I_P, I_A, I_M, I_C, I_T, I_N, I_L \rangle$, *we can construct the context-free model* $\langle J_C, J_T, J_N, J_L \rangle$, *where ...*

The mapping for models will not be so simple however. Basically, each individual I-consistent environment defines its own context-free model – what we want to do is to merge these models. Thus a first attempt might let $J_C = I_C$ and $J_L = I_L$, and define the following mappings for terminals and non-terminals:

$$\forall\, t^{m_1,\ldots,m_k} \in \Sigma_T \cdot$$
$$J_T(t) = \bigcup v_i \in I_M(M_i) \cdot I_T(t^{m_1,\ldots,m_k})(v_1, \ldots, v_k)$$

$$\forall\, n^{m_1,\ldots,m_k} \in \Sigma_N \cdot$$
$$J_N(n) = \bigcup v_i \in I_M(M_i) \cdot I_N(n^{m_1,\ldots,m_k})(v_1, \ldots, v_k)$$

To see how this might work, suppose we have a van-W signature Σ, two notion-sequences ls and rs both from Σ_T. Let us assume that these contain some meta-notion M which can correspond to one of two proto-notion sequences: p or q. Then, if we are given some some Σ-model I which defines:

$$I_T(ls) = \{p \mapsto c, q \mapsto d\}$$
$$I_T(rs) = \{p \mapsto c, q \mapsto d\}$$

we will translate this as:

$$J_T(ls) = \{c, d\}$$
$$J_T(rs) = \{c, d\}$$

Thus, as we expect, the context-free sentence $(ls \rightarrow rs)$ is satisfied in $\beta(I)$ because

$$\{\text{“}c\text{”}, \text{“}d\text{”}\} \subseteq \{\text{“}c\text{”}, \text{“}d\text{”}\}$$

and its its α-image $ls : rs.$ is satisfed in I, since

$$\{\text{“}c\text{”}\} \subseteq \{\text{“}c\text{”}\} \qquad \wedge \qquad \{\text{“}d\text{”}\} \subseteq \{\text{“}d\text{”}\}$$

However, this does not always work both ways, since if we take some other model, call it I' defined as:

$$I'_T(ls) = \{p \mapsto c, q \mapsto d\}$$
$$I'_T(rs) = \{p \mapsto d, q \mapsto c\}$$

we get exactly the same value for $\beta(I')$ as before, except that now while this satisfies $(ls \rightarrow rs)$, its α-image $ls : rs.$ translates under I' to:

$$\{\text{“}d\text{”}\} \subseteq \{\text{“}c\text{”}\} \qquad \wedge \qquad \{\text{“}c\text{”}\} \subseteq \{\text{“}d\text{”}\}$$

which is false, and thus we have not defined an insitution morphism.

In order to correct this we must cause the context-free sentence to be falsified; the problem is that we have been too general with our “forgetting” of the environments. Noting that we have only a finite set of meta-notions, we suggest that any environment (a finite mapping) could be mapped to a unique character over some new alphabet[3]. We propose to index each of the symbols in the β image of a van-W model by these symbols.

In the above example, if we let the two environments be called $E1$ and $E2$, then the sentence $(ls \rightarrow rs)$ translates under the model $\beta(I)$ to

$$\{\text{“}c^{E1}\text{”}, \text{“}d^{E1}\text{”}\} \subseteq \{\text{“}c^{E1}\text{”}, \text{“}d^{E1}\text{”}\}$$

which is true, but under $\beta(I')$ it translates to:

$$\{\text{“}c^{E1}\text{”}, \text{“}d^{E1}\text{”}\} \subseteq \{\text{“}c^{E2}\text{”}, \text{“}d^{E2}\text{”}\}$$

which is false, as required.

So, if we suppose that we have some new set of character symbols $\mathbb{C}$, and a function Ψ_I mapping I-consistent environments into $\mathbb{C}$, then we can proceed to define the components of our model. They are as follows:

1. $J_C = (I_C \times \mathbb{C})$

2. $\forall\, t^{m_1,\ldots,m_k} \in \Sigma_T \cdot$

   $J_T(t) = \bigcup e \in ENV_I(\Sigma_M) \cdot mark(I_T(t^{m_1,\ldots,m_k})(e(m_1),\ldots,e(m_k)),\ \Psi_I(e))$

[3]One method: impose an arbitrary ordering on the meta notions, and list out the environment meta-notion, proto-notion pairs in order. This yields a finite-length string over $(\Sigma_M \cup \Sigma_P)$, which we may regard as a new character

3. $\forall n^{m_1,\ldots,m_k} \in \Sigma_N \cdot$

$$J_N(n) = \bigcup e \in ENV_I(\Sigma_M) \cdot mark(I_N(n^{m_1,\ldots,m_k})(e(m_1),\ldots,e(m_k)),\ \Psi_I(e))$$

4. $J_L = \bigcup e \in ENV_I(\Sigma_M) \cdot mark(I_L, \Psi_I(e))$

We assume that the function

$$mark\colon \wp(I_C^*) \times \mathbb{C} \to \wp(I_C \times \mathbb{C})^*$$

denotes the annotation of each character of each string in the language with the symbol corresponding to the environment.

From this construction it can readily be seen that the construction is now an insitution morphism, since most of the structure of the van-W model has been copied across (and internalised) into the context-free model. Thus we assert:

Lemma 4.21

VWtoCF: $\mathcal{VANW} \Rightarrow \mathcal{CFREE}$, *as defined above, is an institution moprhism*

Proof: *Sketch*

We must show that for every $\mathcal{CFREE}$-signature Σ, every context-free sentence of the form $(A \to r)$ in $[\Sigma]$, and every van-W model I in $[\![\Phi(\Sigma)]\!]$ we have:

$$\langle I_S, I_O, I_C, I_T, I_N, I_L\rangle \models_{\Phi(\Sigma)} A : r. \qquad \Leftrightarrow \qquad \langle J_C, J_T, J_N, J_L\rangle \models_{\Sigma} (A \to r)$$

which, based on the above construction, can be seen to be true. □

This then allows us to set up restraints on context-free signatures in exactly the same way as for attribute grammars.

4.6 Conclusion

In this chapter we have extended our syntactic institutions by examining two grammar-based formalisms for semantic definition. We note that while conceptually similar, the models from $\mathcal{ATG}$ display one specific feature not present in $\mathcal{VANW}$, in that they associate *values* with languages. This greatly facilitates their role as a generative, rather than solely analytical, tool in programming language specification.

We have linked both formalisms back to $\mathcal{CFREE}$, thus permitting context-free description to be restricted by semantic specifications from either institution; an example of this is presented in chapter 6.

Chapter 5

Semantic Definitions

5.1 Introduction

In this chapter we examine some formalisms for describing programming language semantics which are not directly grammar-based. The two we concentrate on here are

- denotational semantics and
- axiomatic semantics

Many other formalisms exist; we might mention in particular *operational* and *algebraic* semantics. Denotational semantics may be regarded as fitting into the general "algebraic" style, which involves specifying an abstract syntax and defining the semantics "structurally" (also called "compositionally") over the terms of this syntax. The reason this approach may be seen as algebraic is that if the abstract syntax is expressed as a signature, then the collection of all programs forms an initial algebra (a sort of Herbrand-expansion of the terms), and maps to semantic elements form algebraic homomorphisms from this syntactic algebra. Modern approaches to operational semantics are also quite similar to this, basically just involving a change in the nature of the target semantic elements.

In this chapter we first present denotational semantics; we suggest that the definition could be modified straightforwardly to deal with algebraic or operational semantics

by changing the functor $[\![\cdot]\!]$ to deal with sets or some abstract definition of "computations". As with previous formalisms, we relate our definition back to $\mathcal{CFREE}$ so that specification of other aspects of a language may be tied in. Examples of modules from this institution can be found in chapter 6.

We deal with axiomatic definitions quite differently to denotational ones however, in that we see them as the amalgamation of some existing compositional-style semantics, and some suitable logic of assertions; specifically we choose denotational semantics and first-order logic.

5.2 Denotational Semantics

Of all the formalisms we will be considering, the denotational style is the one most naturally associated with category theory. While the term "denotational" has quite a general meaning, in programming language semantics it is usually taken to indicate the definition of a language in terms of equations over some category of *domains* which are Cartesian closed. For our purposes here we will take the simplest form of domain - i.e. a chain-complete partially-ordered set, or CPO.

The definition of an institution for denotational-style specifications, call it $\mathcal{DEN}$, will involve a fairly straightforward modification of the definition for general algebra. To make it easier to relate to other formalisms, we will split each denotational definition up into three components:

1. The *abstract syntax*, which defines the syntactic domains
2. The *semantic algebra*, which constraints the semantic domains
3. The *meaning functions*, which relate the syntactic to the semantic domains

Based on this we can define the components of a denotational signature:

Definition 5.1 *Signatures in $\mathcal{DEN}$*

We can define any signature as consisting of:

1. *A set of syntactic domains* Σ_{SYD}

2. *A set of* Σ_{SYD}*-indexed syntactic operators* Σ_{SYO}

3. *A set of semantic domains* Σ_{SED}

4. *A set of* Σ_{SED}*-indexed semantic operators,* Σ_{SEO}

5. *A set of meaning functions* Σ_M *indexed by the elements of* Σ_{SYD}

A signature morphism consists of a map for the syntactic and semantic domains, and a type-consistent map for the operators and meaning functions.

A model of any signature will involve mapping its components into either domains or continuous functions over those domains; thus we define:

Definition 5.2 *Models in* $\mathcal{DEN}$

Given any signature Σ*, a* Σ*-model* I *consists of:*

1. *A function* I_D *mapping the elements of* Σ_{SYD} *and* Σ_{SED} *into CPOs*

2. *A function* I_O *mapping the operators in* Σ_{SYO}, Σ_{SEO} *and* Σ_M *into continuous functions over these domains, in a manner which is type-consistent with their indexing*

To support this we must assume that the category $[\![\Sigma]\!]$ has products; it is common to also allow exponents and sums and other, more specialised operations. For simplicity we omit specific operations for these CPOs, but suggest that their incorporation would not pose any major theoretical difficulties.

Sentences in the institution will consist of equations over the domains defining the semantic algebra and the meaning functions. We will not need to specify equations for the abstract syntax, since all the information we require is provided by the indexing of the operators. To construct sentences, let us assume the existence of some infinite set of Σ_D-indexed variables, while elements we will write as v_i, and define

Definition 5.3 *Sentences in $\mathcal{DEN}$*

Given any signature Σ, the set $[\Sigma]$ can be partitioned into two subsets of closed equations:

1. *The set $[\Sigma]_S$ of semantic equations; these take the form*

$$\forall\, v_1 : D_1, \ldots, v_n : D_n \;\cdot\; e_1 = e_2$$

where each $D_i \in \Sigma_{SED}$, and e_1 and e_2 are expressions built from the semantic operators in Σ_{SEO}

2. *The set $[\Sigma]_M$ of equations defining the meaning functions; all of these are of the form*

$$\forall\, v_1 : D_1, \ldots, v_n : D_n \;\cdot\; M\langle\langle se\rangle\rangle e_1 = e_2$$

where each D_i can be from either Σ_{SYD} or Σ_{SED}, $M \in \Sigma_M$, se is an expression over the syntactic domains (of the appropriate type for M), and e_1 and e_2 are expressions over the syntactic and semantic domains. [1]

Sentence morphisms are the obvious extension of syntax morphisms to terms, preserving the divide between the semantic algebra and the meaning equations.

A sentence of either form is satisfied in some model I if and only if the interpretations of the expressions are equal under all possible assignments of values (from the appropriate CPO) to the universally quantified variables.

As noted in [GB85], the proof of the satisfaction condition for such structures can be seen as a modification of the algebraic institution; we will not deal with it further here.

[1] We will use the angular double-brackets $\langle\langle\cdot\rangle\rangle$ in the denotational definitions to denote pieces of syntax since the more conventional square brackets $[\![\cdot]\!]$ are already in use.

5.3 Relating Denotational and Context-Free Descriptions

Usually when we come to write a denotation definition we assume that the syntactic aspects of the specification have been taken care of by some other, more suitable formalism. Hence, the abstract syntax component of the a denotational specification is really just a base for a linkage to some more concrete syntax, specified elsewhere. Thus, as we have done for other institutions, we will want to be able to constrain the abstract syntax so that it corresponds to some language specified in the context-free institution.

This mapping is slightly less straightforward than the others, since the information describing the language is carried in the *signature* part of a module from $\mathcal{DEN}$, while roughly the same information requires specification by *sentences* in $\mathcal{CFREE}$. Clearly what is needed here is a presentation-based mapping. We note also that models of syntactic signatures in $\mathcal{DEN}$, while possibly representing a more abstract language, actually carry *more* information than models in $\mathcal{CFREE}$. We have the obvious correspondence between non-terminals and syntactic domains, but each domain element that corresponds to some ground term over the operators will implicitly carry with it parsing information which is not present in a string from a context-free language. This suggests that we have little choice over the direction of model-mappings, since the natural transformation β will be many-to-one from terms in $\mathcal{DEN}$ to strings in $\mathcal{CFREE}$.

This then leads us to a definition of Φ for presentations in the opposite direction. Here we envisage any production rule giving rise to an operator representing a function from the (domains corresponding to the) non-terminals on the right-hand-side to the (domain corresponding to the) non-terminal on the left. For this to be accurate we have to make sure that each operator gets a unique name; since we are dealing with a given set of rules, this poses no real problems. Let us use the delimiters $\lfloor\cdot\rfloor$ to denote this function; thus, for example, we might have an operator

of the form $\lfloor S \to \textbf{while } E \textbf{ do } S \textbf{ od} \rfloor : E \times S \to S$

Definition 5.4 *The functor* Φ

Given any context-free signature Σ*, and some* Σ*-presentation, we construct its image under* Φ *by*

1. *letting* Σ_{SYD} *in* $\mathcal{DEN}$ *be* Σ_N*, along with some new sort, let us call it* ℓ *which is not in* Σ_N

2. *letting* Σ_{SYO} *be the set of sentences in the presentation. For each rule, the corresponding operator is indexed as a function from the non-terminals on the left to the non-terminal on the right; for rightparts, the corresponding operators are indexed as function from the non-terminals it contains into* ℓ*.*

A model in some Φ-image of a context-free definition is effectively an abstract syntax tree whose nodes are indexed by the concrete syntax rules that can be used to derive it. To translate this back to a string from the language, we simply traverse the tree and apply the rules. To perform this mapping we define a function Υ_Σ which will translate expressions to strings:

Definition 5.5 *Mapping abstract to concrete syntax*

For any Σ*-presentation* P *in* $\mathcal{CFREE}$*, we define the function* $\Upsilon_\Sigma : |[\![\Phi(P)]\!]| \to |[\![P]\!]|$ *as:*

$$\Upsilon_\Sigma(\lfloor A \to r \rfloor(E_1, \ldots, E_j)) = \upsilon_\Sigma(r)(E_1, \ldots, E_j)$$

where $\lfloor A \to r \rfloor \in \Sigma_{SYD}$*,* $E_1, \ldots, E_j \in |[\![\Phi(P)]\!]|$ *and we define* υ_Σ *as:*

$$\upsilon_\Sigma(\text{`}n_1\text{'} \ldots \text{`}n_k\text{'})(E_1, \ldots, E_j) = \begin{cases} \Lambda & \text{if } k = 0 \\ \upsilon_\Sigma(\text{`}n_2\text{'} \ldots \text{`}n_k\text{'})(E_1, \ldots, E_j) & \text{if } n_1 = \Lambda \\ \text{“}n_1\text{”} \cdot \upsilon_\Sigma(\text{`}n_2\text{'} \ldots \text{`}n_k\text{'})(E_1, \ldots, E_j) & \text{if } n_1 \in \Sigma_T \\ \Upsilon_\Sigma(e_1) \cdot \upsilon_\Sigma(\text{`}n_2\text{'} \ldots \text{`}n_k\text{'})(E_2, \ldots, E_j) & \text{if } n_1 \in \Sigma_N \end{cases}$$

We note that Υ_Σ is a partial function, since there may be elements of the domain which are not mapped to by the interpretation (unless we insist on a minimal model). However, this causes no problems, since we are only interested in those elements to which Υ_Σ applies; the others can be “lost” in the translation. Based on this function we can now define the translation between models:

Definition 5.6 *The natural transformation β For any Σ-presentation P in $\mathcal{CFREE}$, and any model $I \models \Phi(P)$ from $\mathcal{DEN}$, we define*

$$\beta_\Sigma(I) = \langle \Sigma_T,\ \lambda\, t : \Sigma_T \cdot t,\ \lambda\, n : \Sigma_N \cdot \Upsilon_\Sigma(I_D(n)),\ \Upsilon_\Sigma(I_D(\ell)) \rangle$$

Lemma 5.7 *The functor Φ and natural transformation β as defined above constitute a presentation-based mapping CFtoDEN:$\mathcal{CFREE} \Rightarrow \mathcal{DEN}$. That is, for any presentations D in $\mathcal{DEN}$ and C in $\mathcal{CFREE}$, and any D-model d, we have:*

$$d \models \Phi(C) \qquad \Leftrightarrow \qquad \beta(d) \models C$$

Proof: *(Outline)*
Since the presentation $\Phi(C)$ has no sentences (by our construction), the left-hand-side of this biconditional is vacuously true[2] Thus we need only show that:

$$d \models \Phi(C) \qquad \Rightarrow \qquad \beta(d) \models C$$

To prove this we need to take any sentence $A \rightarrow r$ in C, and show that $LAN_\beta(d)(r) \subseteq LAN_\beta(d)(A)$.

By the definition of Υ we can associate with any rightpart a set of parse trees which have that rightpart at their root. It is straightforward to show that the strings defined by the leaves of these parse trees correspond to the language associated with that rightpart.

Hence since $A \rightarrow r$ generates an operator $\lfloor A \rightarrow r \rfloor$ which converts parse trees from r into ones for A, the strings corresponding to r must also correspond to A.

□

We emphasize again that his mapping is different from the other mappings concerning $\mathcal{CFREE}$, since they all left us with context-free models, whereas *CFtoDEN* keeps us in $\mathcal{DEN}$. The understanding is that we have carried out the analytical part of the description, and now wish to transform it into some other form.

[2]In fact any statement of the form ... $\models \Phi(\ldots)$ that is correctly typed (with respect to the signatures involved) will be true.

5.4 Axiomatic Definitions

One of the earliest approaches to dealing with programs in a formal way was *axiomatic semantics.* This involved annotating a program text with boolean statements (called *assertions*), the inference being that the condition specified by the statement should always be true when control passed through that point in the program. Much work has been done in establishing formal software derivation techniques based on this form of annotation.

In meta-logical terms it is common to regard the program text and the assertions as belonging to two different languages, with some degree of identification between variables. Thus many texts which deal with axiomatic semantics regard the assertions as being general statements from first-order logic, and quite separate from the actual program itself. It should also be noted that axiomatic definitions tend to be more "abstract" than their operational or denotational counterparts, since they are only concerned with making assertions about the program, rather than providing a specific model of its semantics or operation.

While axiomatic definitions can be used to describe a language, it is more common to regard them as an additional structure on top of some existing formal semantics. For example, in giving a formal definition of axiomatic semantics (similar to that for first-order logic) [Cou90] assumes the existence of a *relational* semantics for the language, and builds the definition on top of this. At the other end of the scale, [MA86] interpret the assertions as guards, enabling the whole annotated program to be described homogeneously using their partially additive categories.

In general terms then we can regard the concept of an "axiomatic definition" as nothing more than the formulation of a specification in some suitable logic institution, which is then constrained to apply to the programming language via an institution morphism into some semantic institution. The standard choice here would be to use $\mathcal{FOEQ}$, the institution of first-order equational logic of [GB92], for the assertions, and $\mathcal{DEN}$ as the semantic institution; other choices might include:

- Using a logical institution specifically suited to the type of programming language involved: perhaps $\mathcal{EQ}$ for functional programming languages, or $\mathcal{HCL}$, the institution of Horn-Clause logic, for Prolog-like languages

- Using the institution of modal logic as described in [Ste92]; this might be useful for languages involving non-deterministic features, where we envisage the semantics being extended with some suitable structure (such as power-domains).

- We could replace $\mathcal{DEN}$ with some other semantic definition formalism; perhaps we could use $\mathcal{ATG}$ to give a more operational-style semantics, or $\mathcal{UNI}$ the institution of unified algebras [Mos89], and attempt a combination with Mosses' action semantics.

Taking the standard approach then, let us assume that we wish to write axiomatic definitions in first-order logic, and give them meaning by association with existing denotational specifications. To do this we would define:

Definition 5.8 *Relating axiomatic and denotational definitions The components of the institution mapping* $DENtoAX: \mathcal{DEN} \Rightarrow \mathcal{FOEQ}$ *are:*

1. *The functor* $\Phi: \mathbf{Sign}_{\mathcal{DEN}} \rightarrow \mathbf{Sign}_{\mathcal{FOEQ}}$ *takes any denotational signature* Σ*, and performs the mapping:*

$$\Phi(\Sigma) \quad = \quad \langle (\Sigma_{SYD} \cup \Sigma_{SED}), (\Sigma_{SYO} \cup \Sigma_{SEO} \cup \Sigma_{M}), \varnothing \rangle$$

2. *The natural transformation* $\alpha: Sen_{\mathcal{DEN}} \Rightarrow \Phi$*;* $Sen_{\mathcal{FOEQ}}$ *is the identity, since denotational sentences are just quantified equations, and are valid first-order logic sentences*

3. *The natural transformation* α*: Phi;* $Mod_{\mathcal{FOEQ}} \Rightarrow Mod_{\mathcal{DEN}}$ *simply involves splitting back up the mappings for the distinguished subsets of sorts and operators; this is valid since any set may be regarded as a discrete domain.*

Proof that this is in fact an institution mapping (and commutes appropriately with satisfaction) is trivial, since both of the natural transformations make almost no

changes.

Thus we can take any standard Hoare-style definition of a programming language, and rewrite it as a module from $\mathcal{FOEQ}$, with the understanding that its meaning is really only fully understood by appropriate restraints along $DENtoAX$. This may be seen as specifying further conditions on specific programs over and above their denotational definition; thus their main effect will be to nullify the denotational model. This might be seen as something similar to assertions in the C programming language, which cause a particular program's execution to abort if they are falsified when processed during that execution.

5.5 Conclusion

In this chapter we turned our attention to the generative aspects of programming language specification. Here we were no longer interested in further constraining specifications in $\mathcal{CFREE}$, but rather sought to translate these specifications into some other formalisms which seeks to given them a "meaning" in the usual, translational, sense. We chose to describe denotational definitions, and relate these back into context-free specifications, and forwards into axiomatic-style definitions.

We have noted that the connection from $\mathcal{CFREE}$ was different to that presented for other formalisms, as it left us in $\mathcal{DEN}$. In terms of programming language definition the relation to axiomatic definitions is different too, since it is not usual for such specifications to give us extra information about a language's semantics. Instead we may see it as a starting point for a mapping into some other formalism, such as a suitable *refinement calculus*, or some "software engineering" specification language.

Chapter 6

A Small Example

The work in the preceding chapters has established the basic framework in which programming language specifications can be constructed and integrated. In this chapter we give a small example of using these formalisms within the structure of institutions, by specifying the syntax and semantics of a simple imperative block-structured language.

Rather than burden ourselves with excess syntactic baggage we will share nomenclature between modules in the same institution without pointing out specifically the intended shared sub-theories. However, sharing between modules from different institutions will be noted explicitly in the specification.

6.1 A Small Language

The language which we will be specifying consists of nested blocks, each of which consists of declarations followed by statements. Declarations consists of either integer or boolean (scalar) variables. Statements can be assignments, if-then-else constructs or while loops. A variable is in scope in the block where it is declared, and in any sub-blocks defined within that block. Type-checking is done statically based on the declarations seen to date.

The full syntax of the language is as follows:

BLOCK	::=	**begin** *DECL** *STAT** **end**
DECL	::=	**int** *IDENT* | **bool** *IDENT*
STAT	::=	*IDENT* := *EXPR*
	|	**if** *EXPR* **then** *STATS* **else** *STATS* **fi**
	|	**while** *EXPR* **do** *STATS* **od**
	|	*BLOCK*
EXPR	::=	*EXPR BINOP EXPR*
	|	*UNOP EXPR*
	|	*CONST*
	|	*IDENT*
CONST	::=	*NUM* | **true** | **false**
NUM	::=	*DIGIT DIGIT**
IDENT	::=	*LETTER* (*LETTER* | *DIGIT*)*
LETTER	::=	$A \mid \ldots \mid Z \mid a \mid \ldots \mid z$
DIGIT	::=	$1 \mid \ldots \mid 9 \mid 0$
BINOP	::=	$+ \mid - \mid * \mid \div \mid \wedge \mid \vee$
UNOP	::=	$+ \mid - \mid \neg$

We propose to use specifications in all the institutions constructed to date as follows:

- $\mathcal{REG}$ to specify (parts of) the lexical syntax
- $\mathcal{CFREE}$ to specify the remainder of the syntax
- $\mathcal{VANW}$ to give the scope rules
- $\mathcal{ATG}$ to perform type checking
- $\mathcal{DEN}$ to give the basic dynamic semantics

The core of the specification will be the modules from $\mathcal{CFREE}$. As we have discussed in previous chapters, we propose to constrain these by specifications from $\mathcal{REG}$, to restrain them by the static semantics as specified in $\mathcal{VANW}$ and $\mathcal{ATG}$, and then to map this to $\mathcal{DEN}$ to give its dynamic semantics. Implicit in this strategy is the assumption that information (other then validity) gained from the scope- and type-checking is not passed on to the dynamic semantics. This strict delineation between static analysis and the dynamic definition may not always be suitable for more complex examples (such as those involving dynamic typing, for example). For such cases we might still employ the same morphisms between institutions, but use *constraints* (specified into $\mathcal{CFREE}$) which would allow us to stay in one of the semantic institutions.

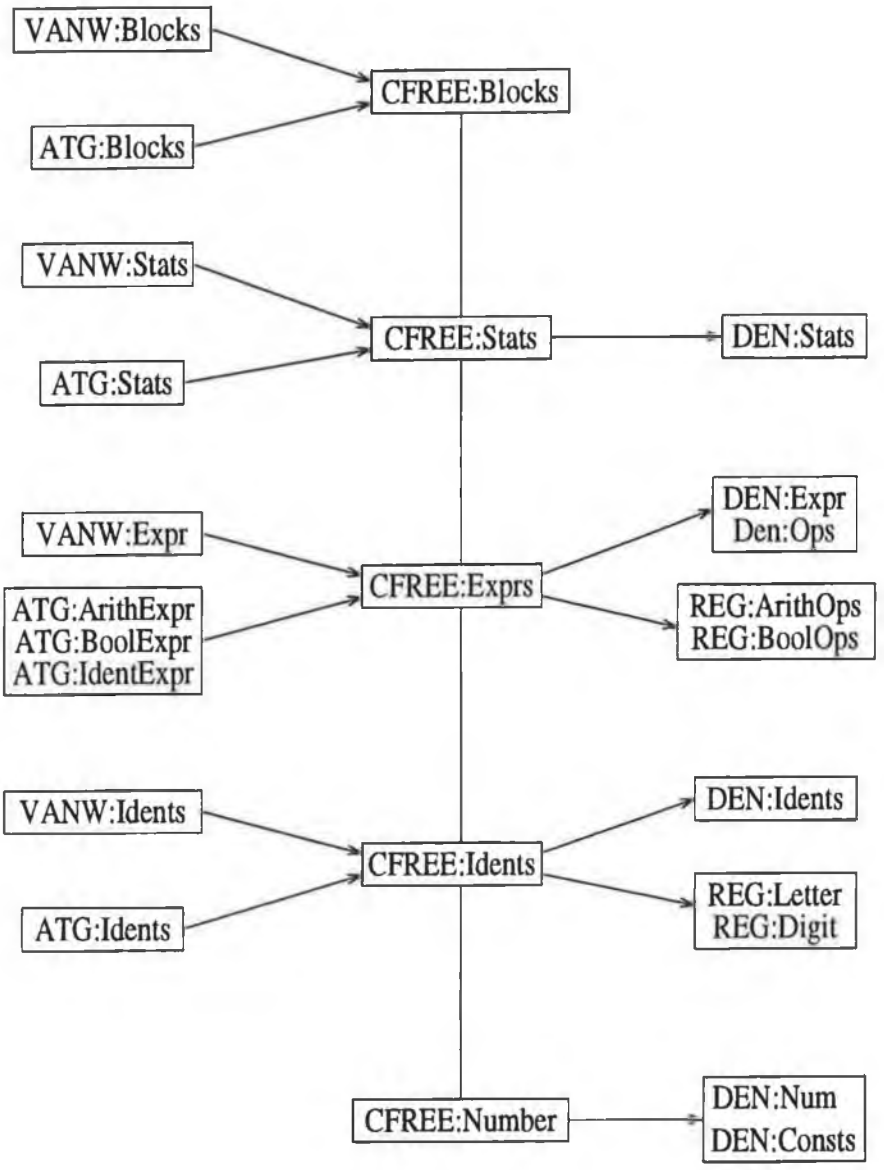
The diagram below gives an overview of the main modules we will be defining; the actual definitions constitute the reaminder of this chapter.

6.2 Preliminaries

One of the key features of a semantic definition is that of the *context*; this is obviously not a feature of the context-free syntactic definition. The context usually contains details about the user-defined names in a program (such as variables, constants, functions etc.); the exact content depends on the type of semantics being defined. Traditionally in compiler design the context is represented by a data structure known as a *symbol table*; the more common term used in semantics, and the one which we shall adopt here, is *environment*.

Since it must give details about the identifiers used in a program, the environment is usually represented as a function from these identifiers into some sort of "record". What is held in this record (if anything) depends on what sort of information we will need to perform the analysis; four our purposes here we suggest:

- For scope checking in $\mathcal{VANW}$ we need only know the names of the variables



The basic modules and their interconnections

declared to date (i.e. currently "in scope")

- For type checking in $\mathcal{ATG}$ we must record the type of each declared variable
- For dynamic semantics in $\mathcal{DEN}$ we will need to associate some value with each identifier that may be referenced

For van-W grammars it is clear that an environment will have to be a meta-level concept; the hyper-rules may then use such environments, and thus represent an infinity of context-free rules, with one (derived) grammar for each possible environment. As we have noted, an environment at this stage need only consist of a list of identifiers currently in scope; thus we define:

$\mathcal{VANW} : \mathit{Envs}$

$\Sigma_M = \{ENV\}$
$\Sigma_P = \Sigma_N = \Sigma_T = \{\}$

ENV :: ENV IDENT; Λ.

The attribute grammars will need to enhance this environment by allowing identifiers be mapped to types. Unlike the van-W grammars, we do not assume that the mechanism for dealing with the semantic aspects is entirely built into $\mathcal{ATG}$, but expect to flesh out these definitions in some other institution. Thus we define the shell:

$\mathcal{ATG} : \mathit{Envs}$

$\Sigma_S = \{\mathit{Name}, \mathit{Type}, \mathit{Env}\}$

$$\Sigma_O = \left\{ \begin{array}{rcl} \text{int}, \text{bool} & : & \mathit{Type}, \\ \text{empty} & : & \mathit{Env}, \\ \text{combine} & : & \mathit{Env} \rightarrow \mathit{Env} \end{array} \right\}$$

$\Sigma_T = \Sigma_N = \{\}$

This simply declares the names of the types and operators that we will be using in our attribute language. We would then expect to constrain this by a module in, say, the algebraic institution (or in some set-theory based notation such as Z) which would assert *Env* as a synonym for $(\mathit{Name} \rightarrow \mathit{Type})$, and impose suitable axioms on *empty* and *combine* so that the former represents an empty function, and the latter represents function overriding. We assume that *Name* is mapped to some

representation for strings.

Finally, we turn to the denotational definition. This time an environment will hold dynamic information: the value associated with a particular variable at some stage of the execution. Let us assume that we have defined the domains *Dval* of denotable values, and *Name* of variable names; then we can complete the definition as:

$\mathcal{DEN}$: *SemAlg*

$\Sigma_{SYD} = \Sigma_{SYO} = \{\}$

$\Sigma_{SED} = \{Denv, Bool, Name, Dval\}$

$$\Sigma_{SEO} = \left\{ \begin{array}{rcl} \text{true}, \text{false} & : & Bool, \\ update & : & Denv \times Name \times Dval \rightarrow Denv, \\ cond & : & Bool \times Denv \times Denv \rightarrow Denv \end{array} \right\}$$

$\Sigma_M = \{\}$

$cond(\text{true}, e1, e2) = e1$

$cond(\text{false}, e1, e2) = e2$

$update(e, n, v) = \lambda\, n' : Name \cdot cond(n = n', v, e(n'))$

(Here we use *Denv* as an abbreviation for the domain $Name \rightarrow Dval$.)

This type of specification in a denotation definition is usually referred to as (part of) the "semantic algebra". It is distinguished by not referring to any syntax (Σ_{SYD} or Σ_{SYO}) or any meaning functions (Σ_M).

6.3 Blocks

A block is the basic program unit, and consists of a sequence of declarations, followed by a sequence of statements. We can express this straightforwardly in the context-free institution as:

$\mathcal{CFREE}$: *Blocks*

$\Sigma_T = \{\textbf{begin}, \textbf{end}, \textbf{int}, \textbf{bool}, ;\ \}$

$\Sigma_N = \{BLOCK, DECLS, DECL, STATS, IDENT\}$

$BLOCK \rightarrow$ **begin** *DECLS STATS* **end**

$DECLS \rightarrow DECL\ DECLS \mid \Lambda$

$DECL \rightarrow$ **int** *IDENT* | **bool** *IDENT*

We have used recursion here rather than Kleene closure in order to make the mapping to other formalisms a little easier.

The corresponding definition in $\mathcal{VANW}$ will need to change the environment in order to take account of the new declarations. Thus we declare two new environments *PRE* and *POST* using meta-rules; in our definition *PRE* is changed to *POST* by the addition of an identifier:

$\mathcal{VANW}$: *Blocks*

$\Sigma_M = \{PRE, POST\}$

PRE :: ENV.
POST :: ENV.
block starting with PRE :
    begin, decls mapping PRE to POST, stats with POST, **end.**
decls mapping PRE to POST :
    declare IDENT, decls mapping PRE IDENT to POST.
decls mapping PRE to PRE : Λ.

The statements are dealt with in the context of *POST*, which is just *PRE* with *all* the declarations of *DECLS* added. Note that we have abstracted away the type details since we do not require them for scope checking.

To link this to the context free definition we will use the institution morphism *VWtoCF*; let us represent the translation by mapping e.g. the notion sequence *stats with POST* to the non-terminal symbol *stats.with.POST*. The we must define some signature morphism which we shall call θ_{BVC} into the context-free signature, thus:

θ_{BVC}: $\Phi(Sign(\mathcal{VANW} : Blocks)) \rightarrow \mathcal{CFREE} : Blocks$

block.starting.with.PRE	$\mapsto$	*BLOCK*
decls.mapping.PRE.to.POST	$\mapsto$	*DECLS*
decls.mapping.PRE.IDENT.to.POST	$\mapsto$	*DECLS*
decls.mapping.PRE.to.PRE	$\mapsto$	*DECLS*
declare.IDENT	$\mapsto$	*DECL*
stats.with.POST	$\mapsto$	*STATS*

Type checking in $\mathcal{ATG}$ will follow a similar pattern, except that this time we must additionally store type information. To do this, we will assume that the non-

terminals *DECLS*, *DECL*, *STATS* and *STAT* all have attribute-type *Env*. Although it is not necessary to specify an evaluation strategy, we may regard the attribute as being synthesised for declarations and inherited for statements.

$\mathcal{ATG}$: *Blocks*

$\Sigma_S = \{Env\}$
$\Sigma_O = \{\text{int}, \text{bool}, \text{combine}, \text{empty}\}$
$\Sigma_T = \{\textbf{begin}, \textbf{end}, \textbf{int}, \textbf{bool}, ;\ \}$
$\Sigma_N = \{BLOCK, DECLS, DECL, STATS, IDENT\}$

BLOCK	$\rightarrow$	**begin** *DECLS STATS* **end**	$\{\$2.env = \text{combine}(\$\$.env, \$1.env)\}$
DECLS	$\rightarrow$	*DECL DECLS*	$\{\$\$.env = \text{combine}(\$1.env, \$2.env)\}$
	$\mid$	Λ	$\{\$\$.env = \text{empty}\}$
DECL	$\rightarrow$	**int** *IDENT*	$\{\$\$.env = \langle \$2.name \mapsto \text{int}\rangle\}$
	$\mid$	**bool** *IDENT*	$\{\$\$.env = \langle \$2.name \mapsto \text{bool}\rangle\}$

In all cases we have spelled out the name of the attribute, even though it is not strictly necessary for types which do not involve a product. The grammar rules here correspond one-to-one with those in $\mathcal{CFREE}$, so we take it that the Φ components of *AGtoCF* strips away the equations, and the signature morphism θ_{BAC} is just the identity.

The denotational definition is the simplest of all at this stage, since we do not need to pick up *any* information from the declarations: hence we do not need a module specifically for blocks!

6.4 Statements

Next we turn to statements; once again the context-free definition is straightforward:

$\mathcal{CFREE}$: *Stats*

$\Sigma_T = \{$**if**, **then**, **else**, **fi**, **while**, **do**, **od**, :=$\}$
$\Sigma_N = \{BLOCK, STATS, STAT, EXPR, IDENT\}$

$$
\begin{array}{rcl}
STATS & \rightarrow & STAT\ STATS \mid \Lambda \\
STAT & \rightarrow & IDENT := EXPR \\
 & \mid & \textbf{if}\ EXPR\ \textbf{then}\ STATS\ \textbf{else}\ STATS\ \textbf{fi} \\
 & \mid & \textbf{while}\ EXPR\ \textbf{do}\ STATS\ \textbf{od} \\
 & \mid & BLOCK
\end{array}
$$

To scope check this we will need to check the identifier in the assignment statement, and allow for some sort of checking of expressions; in both cases we just make sure to pass the current environment down to the next level of definition.

$\mathcal{VANW}$: *Stats*

$\Sigma_M = \{ENV\}$

stats with ENV :
    stat with ENV, stats with ENV;
    Λ.
stat with ENV :
    IDENT from ENV, **assign**, expr in ENV;
    if, expr in ENV, **then**, stats with ENV, **else**, stats with ENV, **fi**;
    while, expr in ENV, **do**, stats with ENV, **od**;
    block starting with ENV .

This is in one-to-one correspondence with the context-free definition, and the morphisms necessary to enable constraints are the obvious ones (mapping *expr.in.ENV* to *EXPR* and so on).

The type-checking follows almost exactly the same pattern, with the additional task of checking that the expressions used in the if and while statements are of boolean type, and that the assignment is type compatible. If we assume the additional attribute types of $EXPR : Env \times Type$ and $IDENT : Name$, we can write:

$\mathcal{ATG}$: *Stats*

$\Sigma_S = \{Env, Name, Type\}$
$\Sigma_O = \{\}$
$\Sigma_T = \{\textbf{if}, \textbf{then}, \textbf{else}, \textbf{fi}, \textbf{while}, \textbf{do}, \textbf{od}, :=\}$
$\Sigma_N = \{BLOCK, STATS, STAT, EXPR, IDENT\}$

STATS	→	*STAT STATS*	{\$1.*env* = \$\$.*env*, \$2.*env* = \$\$.*env*}
	|	Λ	
STAT	→	*IDENT* := *EXPR*	{\$3.*env* = \$\$.*env*, \$\$.*env*(\$1.*name*) = \$3.*type*}
	|	**if** *EXPR* **then** *STATS* **else** *STATS* **fi**	{\$2.*env* = \$\$.*env*, \$4.*env* = \$\$.*env*, \$6.*env* = \$\$.*env*, \$2.*type* = "*bool*"}
	|	**while** *EXPR* **do** *STATS* **od**	{\$2.*env* = \$\$.*env*, \$4.*env* = \$\$.*env*, \$2.*type* = "*bool*"}
	|	*BLOCK*	{\$2.*env* = \$\$.*env*}

One again these rules are in exact correspondence with the context-free syntax, and so we will not spell out the mapping.

Next the denotational definition. As is common for denotational definitions, in this and subsequent modules we will leave out the explicit quantification of the variables, assuming that this is evident from the context in which they are used.

Note that the following module defines both the (abstract) syntax and the semantics of statements.

$\mathcal{DEN}$: *Stats*

$\Sigma_{SYD} = \{STATS, STAT, EXPR, IDENT\}$

$$\Sigma_{SYO} = \left\{ \begin{array}{rcl} join & : & STAT \times STATS \to STATS, \\ none & : & STATS, \\ assign & : & IDENT \times EXPR \to STAT, \\ if & : & EXPR \times STATS \times STATS \to STAT, \\ while & : & EXPR \times STATS \to STAT, \\ more & : & STATS \to STAT \end{array} \right\}$$

$\Sigma_{SED} = \{Denv, Bool, Name, Dval\}$

$\Sigma_{SEO} = \{update, cond\}$

$$\Sigma_M = \left\{ \begin{array}{rcl} M_{STATS}, M_{STAT} & : & Denv \to Denv \\ M_{IDENT} & : & Name \\ M_{EXPR} & : & Denv \to Dval \end{array} \right\}$$

$M_{STATS}\langle\!\langle join(stat, stats)\rangle\!\rangle e = M_{STATS}\langle\!\langle stats\rangle\!\rangle(M_{STAT}\langle\!\langle stat\rangle\!\rangle e)$

$M_{STATS}\langle\!\langle none\rangle\!\rangle e = e$

$M_{STAT}\langle\!\langle assign(ident, expr)\rangle\!\rangle e =$

$\quad update(e, M_{IDENT}\langle\!\langle ident\rangle\!\rangle e, M_{EXPR}\langle\!\langle expr\rangle\!\rangle e)$

$M_{STAT}\langle\!\langle if(expr, stats1, stats2)\rangle\!\rangle e =$

$\quad cond(M_{EXPR}\langle\!\langle expr\rangle\!\rangle e, M_{STATS}\langle\!\langle stats1\rangle\!\rangle e, M_{STATS}\langle\!\langle stats2\rangle\!\rangle e)$

$M_{STAT}\langle\!\langle while(expr, stats)\rangle\!\rangle e =$

$\quad cond(M_{EXPR}\langle\!\langle expr\rangle\!\rangle e, M_{STATS}\langle\!\langle stats\rangle\!\rangle e, e)$

$M_{STAT}\langle\!\langle more(stats)\rangle\!\rangle e = M_{STATS}\langle\!\langle stats\rangle\!\rangle e$

This needs to be connected to the context-free institution via the presentation-based mapping *CFtoDEN*; to do this we need to define the morphism θ required in the constraint. If we assume that all the Σ_N components are mapped to the obvious corresponding syntactic domain, with *BLOCK* being mapped to *STATS*, we can write:

θ_{SCD}: $\Phi(\mathcal{CFREE}$: *Stats*$) \to \mathcal{DEN}$: *Stats*

$$\begin{array}{rcl} \lfloor STATS \to STAT\ STATS \rfloor & \mapsto & join \\ \lfloor STATS \to \Lambda \rfloor & \mapsto & none \\ \lfloor STAT \to IDENT := EXPR \rfloor & \mapsto & assign \\ \lfloor STAT \to \textbf{if}\ EXPR\ \textbf{then}\ STATS\ \textbf{else}\ STATS\ \textbf{fi} \rfloor & \mapsto & if \\ \lfloor STAT \to \textbf{while}\ EXPR\ \textbf{do}\ STATS\ \textbf{od} \rfloor & \mapsto & while \\ \lfloor STAT \to BLOCK \rfloor & \mapsto & more \end{array}$$

This completes the definitions for statements.

6.5 Expressions

The last major section of the language is the set of expressions; we allow binary and unary operations, identifiers and constants. Syntactically, we define:

$\mathcal{CFREE}$: *Expr*

$\Sigma_T = \{\}$

$\Sigma_N = \{EXPR, BINOP, UNOP, CONST, IDENT\}$

$$
\begin{aligned}
EXPR \rightarrow\ & EXPR\ BINOP\ EXPR \\
|\ & UNOP\ EXPR \\
|\ & CONST \\
|\ & IDENT
\end{aligned}
$$

Defining scope details involves passing down the environment to sub-expressions; the base cases involve constants, which have no scope, and identifiers, which are checked separately.

$\mathcal{VANW}$: *Expr*

$\Sigma_M = \{ENV\}$

```
expr in ENV :
    expr in ENV, binop, expr in ENV;
    unop, expr in ENV;
    const;
    IDENT from ENV.
```

When doing type checking however we have some extra flexibility with the grammar rules. It is usually possible to distinguish syntactically different types of expressions; however, since this is not generally applicable, it is also usual to ignore this in the specification. Since we have separated syntactic and semantic description, we may add back in this possibility, and define three modules for expressions.

The first defines expressions which are obviously arithmetical:

$\mathcal{ATG}$: *ArithExpr*

$\Sigma_S = \{Env, Type\}$
$\Sigma_O = \Sigma_T = \{\}$
$\Sigma_N = \{EXPR, ABINOP, AUNOP, NUM\}$

$$
\begin{aligned}
EXPR \;\rightarrow\; & EXPR\ ABINOP\ EXPR \\
& \{\$1.env = \$\$.env,\ \$3.env = \$\$.env, \\
& \$\$.type = \text{“}int\text{”}, \$1.type = \text{“}int\text{”}, \$3.type = \text{“}int\text{”}\} \\
\mid\; & AUNOP\ EXPR \quad \{\$2.env = \$\$.env, \\
& \$\$.type = \text{“}int\text{”}, \$2.type = \text{“}int\text{”}\} \\
\mid\; & NUM \quad \{\$\$.type = \text{“}int\text{”}\}
\end{aligned}
$$

The second defined those which are definitely boolean:

$\mathcal{ATG}$: *BoolExpr*

$\Sigma_S = \{Env, Type\}$
$\Sigma_O = \Sigma_T = \{\}$
$\Sigma_N = \{EXPR, BBINOP, BUNOP, BOOLVAL\}$

$$
\begin{aligned}
EXPR \;\rightarrow\; & EXPR\ BBINOP\ EXPR \\
& \{\$1.env = \$\$.env,\ \$3.env = \$\$.env, \\
& \$\$.type = \text{“}bool\text{”}, \$1.type = \text{“}bool\text{”}, \$3.type = \text{“}bool\text{”}\} \\
\mid\; & BUNOP\ EXPR \quad \{\$2.env = \$\$.env, \\
& \$\$.type = \text{“}bool\text{”}, \$2.type = \text{“}bool\text{”}\} \\
\mid\; & BOOLVAL \quad \{\$\$.type = \text{“}bool\text{”}\}
\end{aligned}
$$

And lastly we have those which cannot be identified specifically as either (not grammatically, anyway):

$\mathcal{ATG}$: *IdentExpr*

$\Sigma_S = \{Env, Name, Type\}$
$\Sigma_O = \Sigma_T = \{\}$
$\Sigma_N = \{EXPR, IDENT\}$

$$
EXPR \;\rightarrow\; IDENT \quad \{\$\$.type = \$\$.env(\$1.name)\}
$$

This approach has the advantage of considerably simplifying the specification; it might also be of use should we wish to extend the language later by adding in new data types (we could perhaps also modularise the declarations section for this).

To map this back to the context-free case we perform an abstraction. The functor Φ for *AGtoCF* will lose the equations, and letting

$$\mathcal{ATG} : Expr \triangleq \mathcal{ATG} : ArithExpr + \mathcal{ATG} : BoolExpr + \mathcal{ATG} : IdentExpr$$

we can define the signature morphism θ_{EAC} as follows:

$\theta_{EAC}: \Phi(Sign(\mathcal{ATG} : Expr)) \rightarrow \mathcal{CFREE} : Expr$

$$\begin{aligned} EXPR &\mapsto EXPR \\ ABINOP, BBINOP &\mapsto BINOP \\ AUNOP, BUNOP &\mapsto UNOP \\ NUM, BOOLVAL &\mapsto CONST \\ IDENT &\mapsto IDENT \end{aligned}$$

The denotational definition, relieved of type information, just passes on the environment to the lower levels:

$\mathcal{DEN} : Expr$

$\Sigma_{SYD} = \{EXPR, BINOP, UNOP, CONST, IDENT\}$

$$\Sigma_{SYO} = \left\{ \begin{array}{rcl} apply & : & BINOP \times EXPR \times EXPR \rightarrow EXPR, \\ apply & : & UNOP \times EXPR \rightarrow EXPR, \\ a & : & CONST \rightarrow EXPR, \\ an & : & IDENT \rightarrow EXPR \end{array} \right\}$$

$\Sigma_{SED} = \{Denv, Dval, Name\}$

$\Sigma_{SEO} = \{\}$

$\Sigma_M = \{M_{EXPR}, M_{CONST}, M_{IDENT}, M_{BOP}, M_{UOP}\}$

$M_{EXPR}\langle\langle apply(binop, expr1, expr2)\rangle\rangle e =$
$\quad M_{BOP}\langle\langle binop\rangle\rangle(M_{EXPR}\langle\langle expr1\rangle\rangle e, M_{EXPR}\langle\langle expr2\rangle\rangle e)$
$M_{EXPR}\langle\langle apply(unop, expr)\rangle\rangle e = M_{UOP}\langle\langle unop\rangle\rangle(M_{EXPR}\langle\langle expr\rangle\rangle e)$
$M_{EXPR}\langle\langle a(const)\rangle\rangle e = M_{CONST}\langle\langle const\rangle\rangle$
$M_{EXPR}\langle\langle an(ident)\rangle\rangle e = e(M_{IDENT}\langle\langle ident\rangle\rangle)$

This depends on the following link from the *CFtoDEN* image of the context-free description:

$\theta_{ECD}: \Phi(\mathcal{CFREE} : Expr) \rightarrow \mathcal{DEN} : Expr$

$$\begin{aligned} \lfloor EXPR \rightarrow EXPR\ BINOP\ EXPR \rfloor &\mapsto apply \\ \lfloor EXPR \rightarrow UNOP\ EXPR \rfloor &\mapsto apply \\ \lfloor EXPR \rightarrow CONST \rfloor &\mapsto a \\ \lfloor EXPR \rightarrow IDENT \rfloor &\mapsto an \end{aligned}$$

6.6 Operators

We have made use of unary and binary operators in our specifications; we can give them a precise definition at the lexical level:

$\mathcal{REG}$: *ArithOps*

$\Sigma_A = \{+, -, *, \div\}$
$\Sigma_L = \{AUNOP, ABINOP\}$

$(+ \mid - \mid * \mid \div) : ABINOP$
$(+ \mid -) : AUNOP$

$\mathcal{REG}$: *BoolOps*

$\Sigma_A = \{\wedge, \vee, \neg\}$
$\Sigma_L = \{BUNOP, BBINOP\}$

$(\wedge \mid \vee) : BBINOP$
$\neg : BUNOP$

We take a number of approaches to linking these with the various expression modules.

The link with the context-free syntax involves using the institution morphism *CFtoR*, with $\mathcal{CFREE}$: *Expr* being constrained by the sum of the above modules (which we call $\mathcal{REG}$: *Ops*) via the $\mathcal{REG}$ signature morphism:

θ_{OCR}: $Sign(\mathcal{REG} : Ops) \rightarrow \Phi(\mathcal{CFREE} : Expr)$

$ABINOP, BBINOP \mapsto BINOP$
$AUNOP, BUNOP \mapsto UNOP$

We do not need to worry about the definition in $\mathcal{VANW}$, since the mapping of $binop \mapsto BINOP$ and $unop \mapsto UNOP$ will cause the appropriate constraints to be applied to the proto-notions.

However, we make specific use of the difference in type of the operators at the type checking stage, and so we would envisage constraining $\mathcal{ATG}$: *ArithExpr* by $\mathcal{REG}$: *ArithOps* and $\mathcal{ATG}$: *BoolExpr* by $\mathcal{REG}$: *BoolOps*. This extra level of precision at the type checking stage may seem to "bypass" the context-free definition, but in reality all it does is to further limit the models in $\mathcal{ATG}$ by which the context-free

models are constrained (in an appropriate manner, of course).

The denotational definition will also need to make use of the actual operator symbols in its translation of them into functions over domains.

$\mathcal{DEN}$: *Ops*

$$\Sigma_{SYD} = \{BINOP, UNOP\}$$

$$\Sigma_{SYO} = \left\{ \begin{array}{rcl} cross, dash, star, over, and, or & : & BINOP, \\ cross, dash, not & : & UNOP \end{array} \right\}$$

$$\Sigma_{SED} = \{Dval\}$$

$$\Sigma_{SEO} = \left\{ \begin{array}{rcl} add, subtract, mult, div, conj, disj & : & Dval \times Dval \rightarrow Dval, \\ negate, lnegate & : & Dval \rightarrow Dval \end{array} \right\}$$

$$\Sigma_M = \{M_{BOP}, M_{UOP}\}$$

$$M_{BOP}\langle\langle cross\rangle\rangle = \lambda\, v_1, v_2 : Dval \cdot add(v_1, v_2)$$
$$M_{BOP}\langle\langle dash\rangle\rangle = \lambda\, v_1, v_2 : Dval \cdot subtract(v_1, v_2)$$
$$M_{BOP}\langle\langle star\rangle\rangle = \lambda\, v_1, v_2 : Dval \cdot mult(v_1, v_2)$$
$$M_{BOP}\langle\langle over\rangle\rangle = \lambda\, v_1, v_2 : Dval \cdot div(v_1, v_2)$$
$$M_{BOP}\langle\langle and\rangle\rangle = \lambda\, v_1, v_2 : Dval \cdot conj(v_1, v_2)$$
$$M_{BOP}\langle\langle or\rangle\rangle = \lambda\, v_1, v_2 : Dval \cdot disj(v_1, v_2)$$
$$M_{UOP}\langle\langle cross\rangle\rangle = \lambda\, v_1 : Dval \cdot v_1$$
$$M_{UOP}\langle\langle dash\rangle\rangle = \lambda\, v_1 : Dval \cdot negate(v_1)$$
$$M_{UOP}\langle\langle not\rangle\rangle = \lambda\, v_1 : Dval \cdot lnegate(v_1)$$

We will not quote here the rather obvious definitions of the above semantic operators, or the standard link from the syntactic definition of the syntactic operator symbols.

6.7 Constants

Constants can either be numbers, or the two boolean constants true and false; we define the lexical syntax:

$\mathcal{REG}$: *BoolConst*

$$\Sigma_A = \{\mathsf{true}, \mathsf{false}\}$$
$$\Sigma_L = \{BOOLVAL\}$$

$$(\mathsf{true} \mid \mathsf{false}) : BOOLVAL$$

$\mathcal{REG}$: *Digit*

$$\Sigma_A = \{1, \ldots, 9, 0\}$$
$$\Sigma_L = \{DIGIT\}$$

$$(1 \mid \ldots \mid 9 \mid 0) : DIGIT$$

We can constrain the *CONST* in $\mathcal{CFREE}$: *Expr* with a signature morphism that takes *BOOLVAL* $\mapsto$ *CONST*; this can be further widened by summing in the rules:

$\mathcal{CFREE}$: *Number*

$\Sigma_T = \{\}$
$\Sigma_N = \{CONST, NUM, DIGIT\}$

CONST $\rightarrow$ *NUM*

NUM $\rightarrow$ *DIGIT DIGIT**

As before, the specifications in $\mathcal{VANW}$ can safely ignore constants. The definitions in $\mathcal{ATG}$ can be separately constrained by the regular definition of *BOOLVAL*, and the context-free definition of *NUM*, to ensure that the constants are still separated on type. And lastly, the denotational definition will need to interpret constants into actual values:

$\mathcal{DEN}$: *Consts*

$\Sigma_{SYD} = \{CONST, NUM, DIGIT\}$
$\Sigma_{SYO} = \{a : NUM \rightarrow CONST, truth, falsehood : CONST\}$
$\Sigma_{SEM} = \{Bool, Dval\}$
$\Sigma_{SEO} = \{\}$
$\Sigma_M = \{M_{CONST}\}$

$M_{CONST}\langle\langle a(num)\rangle\rangle = M_{NUM}\langle\langle num\rangle\rangle$
$M_{CONST}\langle\langle truth\rangle\rangle = \text{true}$
$M_{CONST}\langle\langle falsehood\rangle\rangle = \text{false}$

We omit the obvious definition of M_{NUM}.

6.8 Identifiers

Finally, we come to the definition of identifiers. We can define them in the usual way syntactically:

$\mathcal{CFREE}$: *Idents*

$\Sigma_T = \{\}$
$\Sigma_N = \{IDENT, LETTER, DIGIT\}$

IDENT $\rightarrow$ *LETTER* (*LETTER* | *DIGIT*)*

We will define the concepts of letter and digit at the lexical level:

$\mathcal{REG}$: *Letter*

$\Sigma_A = \{A, \ldots, Z, a \ldots, z\}$
$\Sigma_L = \{LETTER\}$

$(A \mid \ldots \mid Z \mid a \mid \ldots \mid z) : LETTER$

Here we assume that $\mathcal{CFREE}$: *Expr* is

$\mathcal{REG}$: *Digit*

$\Sigma_A = \{1, \ldots, 9, 0\}$
$\Sigma_L = \{DIGIT\}$

$(1 \mid \ldots \mid 9 \mid 0) : DIGIT$

Imposing this as a constraint on the context-free grammar is straightforward.

Next we define identifiers from the scope point of view, with a built-in check to see if they are in the current environment (this is a variable reference, as opposed to a variable definition):

$\mathcal{VANW}$: *Idents*

$\Sigma_M = \{PRE, POST\}$

PRE :: ENV.
POST :: ENV.
IDENT from PRE IDENT POST : Λ.

The attribute grammar will require some method of extracting an element of *Name* from a variables syntactic appearance; we do not spell this out here, and just include the following shell for completeness:

$\mathcal{ATG}$: *Idents*

$\Sigma_S = \{Name\}$
$\Sigma_O = \Sigma_T = \{\}$
$\Sigma_N = \{IDENT, LETTER, DIGIT\}$

$IDENT \rightarrow LETTER\ (LETTER \mid DIGIT)^* \quad \{\$\$.name = \ldots\}$

The denotational definition for M_{IDENT} will have to perform an analogous operation, which we again omit.

6.9 Conclusion

In this chapter we have given a small example of a programming language specification built from modules in five different specification languages, related using institution mappings and morphisms. Apart from the general advantages of modularisation noted in previous chapters, we can see that it has been possible to omit modules for language components from some of the institutions altogether, and to significantly simplify others, compared with standard definitions.

Chapter 7

Conclusions

The goal of this thesis was to demonstrate the applicability of the theory of institutions to the specification and integration of those formalisms used to describe (various aspects of) programming languages. Here we wish to reflect on the work done and on its implications.

7.1 What's been done?

In the preceding chapters we have taken five programming language specification formalisms and constructed five institutions:

1. $\mathcal{REG}$ for regular expressions
2. $\mathcal{CFREE}$ for context-free descriptions
3. $\mathcal{ATG}$ for attribute grammars
4. $\mathcal{VANW}$ for van Wijngaarden grammars
5. $\mathcal{DEN}$ for denotational semantics

We have suggested that a sixth formalism, axiomatic semantics, does not require a separate institution but can be seen as the integration of a suitable semantic and logic institution.

We have performed four basic tasks with each of these institutions:

1. We have shown how the formalism can be incorporated into the institutional framework

2. We have verified the necessary results, such as the satisfaction condition and the existence of co-limits in the category of signatures, to ensure that these are, in fact, valid institutions

3. We have discussed and provided examples of their use

4. We have provided institution mappings relating these to each other, most particularly to $\mathcal{CFREE}$

Chapters three through five develop the individual institutions; an integrated example of their use was presented in chapter six.

In all cases the design of each institution was the crucial step. We found this to be an iterative process which started with a basic definition which was then modified as we:

- attempted to prove the satisfaction condition

- began to formulate and use actual modules from the institution

- attempted to integrate it with other institutions

It was quite often the case that each of these steps resulted in important, and occasionally quite substantial, modification of the initial definitions. Generally this involved working on the concept of a model within the institution, but occasionally modifications to the concept of signature and, less frequently, the sentences were required. We believe this reflects the general nature of an institutional description as being "denotational" in style: we already know what the formalism looks like in practice (as represented by the sentences): our task is to provide a suitable model for its meaning.

We have already mentioned that often one of the main tasks in constructing an institution is proving that the satisfaction condition holds. Indeed, work has been carried out (in [GB85]) on specific structures to facilitate this, using *charters* and *parchments* as tools to construct institutions in which the satisfaction condition definitionally holds. On the other hand, the proofs of the satisfaction condition for the institutions in this thesis were not complex. While some of this was undoubtedly due to the inductive nature of the structures being dealt with, it should be pointed out that in constructing these institutions originally, attempting (and failing) to prove the satisfaction condition provided a key "checkpoint" in their development. It was often the case that the insight gained in checking this condition resulted in substantial modifications being made to the concepts of models and sentences in the institution, with this in turn facilitating the proof of the satisfaction condition.

It might be noted that the models are set-theoretic, rather than category-theoretic in nature (this was perhaps contrary to our own initial expectations!). It should be remembered that the role of category theory here is as a background to the semantics of institutions themselves, and that this does not imply that the actual components of a given institution need be categorical. Indeed, one of the most common example of an institution, $\mathcal{FOEQ}$ for first-order logic, is set-theoretic in nature, despite the availability of alternatives in category theory (as in e.g. [LS86]). This does not, of course, preclude the definition of more categorical alternatives to our own and their incorporation within the framework.

On a similar theme, it should be pointed out that some of the power of the "categorical" aspect of institutions has not been exploited here. Since the collection of signatures in any institution is a category we are concerned not only with the signatures themselves, but also with the morphisms between them; this leads towards mechanisms for providing modularity and parameterisation in modules. While this is necessarily in the background here in our use of information sharing between modules, we have not pursued the issue fully. A possible development of this work might include examining the role of parameterised modules in the institutions we have

presented, and also the interaction between parameterised modules from different institutions.

7.2 What use is it?

We suggest that the contribution made by this work may be dealt with under four main headings:

1. The theory of institutions

   While we have not altered the basics of institution theory in any fundamental way, we have added the new concept of a *restraint* to the theory, which allows for presentations in one specification to be "shadowed" by those in another in a manner not previously possible. In addition, we have introduced the concept of a *presentation-based mapping*, which provides for the translation of signatures in the context of a particular set of sentences.

2. The application of institutions

   With any new theory much of the initial work in using it involves a process of familiarisation with the style and goals of that theory. One important resource in this process is the existence of a pool of previous work in the area. We have augmented this by providing examples of the design and definition of five new institutions, and demonstrated their application in a small example.

3. Formalisms for describing programming languages

   In order to understand the various formalisms used to describe programming languages, it is necessary to have some common frame of reference within which to compare them. We suggest that the institutional descriptions, along with the mappings and morphisms that we have defined, provide such a reference point by building a homogeneous framework within which they can be compared and contrasted

4. Programming language specification

   As we have previously noted, the approach that we have taken has not involved arbitrary integration of formalisms, but a deliberate policy of constraining the

concrete syntax with static semantic descriptions, and then imposing this on the abstract syntax on which the dynamic definition is based. As such we suggest that this is a style of definition which differs from standard algebraic approaches, and could be seen as an "institutional" approach to programming language specification.

7.3 What next?

As with any work of this nature, further extensions are possible. We have attempted to provide a reasonable cross-sample of the types of formalisms used in programming language definition; one obvious extension to our work would be the incorporation of yet more specification languages.

Two other possibilities come to mind, both under the heading of "implementation", though at the object- and meta-level respectively.

- Implementing the specifications

  In his work on general logics, [Mes89] incorporates the concept of *proof calculi* within the institutional system, providing an operational aspect to the definition. In our terms, these proof calculi correspond to parsing strategies, type-checking algorithms, and methods for prototyping semantic definitions. Given our institutional descriptions, it might be useful to investigate their operational side; this would involve

  - Formulating institutional descriptions of the calculi involved
  - Attempting to investigate their possible integration along the line of the institution mappings that we have specified

  From the software engineering point of view, we might envisage a refinement calculus being associated with some or all of the languages; it could be relevant to investigate the possibilities of carrying out such refinements "in tandem" between formalisms connected by mappings or morphisms

- Implementing the framework

  The theory of institutions itself forms a specification language, and would benefit from the availability of the types of tools associated with other specification languages. In particular, some automatic assistance in the verification of the satisfaction condition would be a considerable help. The prospects here are quite tangible: something close to this already exists with the logical framework LF [HST89a], and, with a suitable framework for describing category theory, it should be possible to build up a suitable theory of institutions in some meta-logical framework such as Isabelle [Pau90] or Coq [DF+93].

Both of these are worthy of further study and would, we believe, help to underline the usefulness of the theory of institutions for the specification of programming languages.

Appendix A

Category Theory

Here we define some of the basic concepts from category theory used in this document. The standard reference for category theory is [Mac71], for a computing-related introduction [BW90] is particularly comprehensive; other references include [RB88], [AL91] and [Pie91]. A general overview of the relevance of various categorical concepts in computing can be found in [Gog89].

Category

A category consists of:

- a collection of *objects*
- a collection of *arrows* (also called *morphisms*), indexed by two objects (called its source and destination); we write $f\colon A \to B$ if f is a morphism from object A to object B

such that

- For any objects A, B and C, and arrows $f\colon A \to B$ and $g\colon B \to C$ there is an arrow $f;\ g\colon A \to BC$, called their *composition*, such that for any arrows f, g and h which can be composed, we have

$$f;\ (g;\ h) = (f;\ g);\ h$$

- For each object B there is an arrow id_B, such that for any morphisms $f\colon A \to B$ and $g\colon B \to C$ we have

$$f;\ id_A = f \qquad \text{and} \qquad id_A;\ g = g$$

The set of objects of any category **C** is usually written as $|C|$.

Duality

The dual of any category **C** has the same objects as **C**, with the arrows going in the *opposite* direction; it is written $\mathbf{C}^{\mathbf{op}}$

Functor

Given two categories **C** and **D**, we define a functor $F\colon \mathbf{C} \to \mathbf{D}$ as consisting of:

- A function F_O mapping **C**-objects to **D**-objects
- A function F_A mapping **C**-arrows to **D**-arrows, where if $f\colon A \to B$ is an arrow in **C** then $F_A(f)\colon F_O(A) \to F_O(B)$ is an arrow in **D**

such that:

- For any **C**-object A,

$$F_A(id_A) = ID_{F_O(A)}$$

- For any composable **C**-arrows f and g,

$$F_A(f;\ g) = F_A(f);\ F_A(g)$$

Given any two **C**-objects A and B, any functor $F\colon \mathbf{C} \to \mathbf{D}$ induces a mapping from the arrows between A and B to the arrows between $F(A)$ and $F(B)$; then

- A functor is said to be **full** if this mapping is surjective
- A functor is said to be **faithful** if this mapping is injective

Sub-Category

A category **C** is a sub-category of some other category **D**. if it set of objects and arrows are subsets of those from **D**. It is a **full sub-category** if the induced inclusion functor from **C** to **D** is full.

Natural Transformation

Given any two functors $F: \mathbf{C} \to \mathbf{D}$ and $G: \mathbf{C} \to \mathbf{D}$, a natural transformation $\eta: F \Rightarrow G$ consists of:

- For each **C**-object A, a **D**-arrow $\eta_A: F(A) \to G(A)$

such that:

- For any **C**-arrow $f: A \to B$,

$$\eta_A;\ G(f) = F(f);\ \eta_B$$

Diagram

A diagram in a category is any collection of objects and arrows from that category

CoCones

A cocone α in some category **C** consists of:

- some **C**-diagram α^D, called the *base* of the cone
- some **C**-object α^A, called the *apex* of the cone
- for each node I in the diagram, an arrow $\alpha_I: I \to A$

such that:

- For any edge $e: I \to J$ in α^D,

$$\alpha_I = e;\ \alpha_J$$

CoLimits

A colimit in some category **C** is any **C**-cocone α such that

- If β is any other **C**-cocone with the same diagram as α then there is a unique **C**-arrow $f: \alpha^A \to \beta^A$ such that for any node I in the base

$$\alpha_I ; f = \beta_I$$

Colimits for a diagram are unique up to isomorphism.

Initial Object

An object is initial in a category if it is the colimit of the empty diagram

Coproducts

The coproduct (or *sum*) of two objects is the colimit of the diagram containing those objects (with no edges)

Pushouts

For any two morphisms in a category $f: A \to B$ and $h: A \to C$, their pushout is the colimit of the corresponding diagram

Cones and Limits

Cones are dual to cocones

Limits are dual to colimits

Terminal Objects are dual to initial objects

Products are dual to sums

Pullbacks are dual to pushouts

Examples of Categories

Some of the standard categories mentioned in this thesis are:

- **Set** whose objects are sets and morphisms are (total) functions
- **Cat** whose objects are categories and whose morphisms are functors
- $\mathbf{Cat}^{\mathbf{op}}$ which is the dual of **Cat**; i.e. it has the same objects, but all the arrows are reversed
- **2** which has two objects and no arrows

Bibliography

[AL91] A. Asperti and G. Longo. *Categories, Types and Structures.* MIT Press, 1991.

[AM91] H. Alblas and B. Melichar, editors. *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science.* Springer Verlag, 1991.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[Avr91] Arnon Avron. Simple consequence relations. *Information and Computation*, 92:105–139, 1991.

[Bar74] K.J. Barwise. Axioms for abstract model theory. *Annals of Mathematical Logic*, 7:221–265, 1974.

[BEPP87] E.K. Blum, H. Ehrig, and F. Parisi-Presicce. Algebraic specification of modules and their basic interconnections. *Journal of Computer and System Sciences*, 34:293–339, 1987.

[BG80] R.M. Burstall and J.A. Goguen. The semantics of clear, a specification language. In G. Goos and J. Hartmanis, editors, *Advanced Course on Abstract Software Specification*, volume 86 of *Lecture Notes in Computer Science*, pages 292–332. Springer Verlag, 1980.

[BG81] R.M. Burstall and J.A. Goguen. An informal introduction to specifications using CLEAR. In R.S. Boyer and J.S. Moore, editors, *The Correctness problem in Computer Science*, pages 185–213. Academic Press, 1981.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint. *Algebraic Specification.* Addison-Wesley, 1989.

[BJ82] D. Bjorner and C.B. Jones. *Formal Specification and Software Development*, chapter 9, pages 271–320. Prentice Hall, 1982.

[BV87] C. Bierle and Angelika Voβ. Viewing implementations as an institution. In D.H.Pitt, A.Poigné, and D.E. Rydeheard, editors, *Category Theory and Computer Science*, volume 283 of *Lecture Notes in Computer Science*, pages 196–218. Springer Verlag, 1987.

[BW90] Michael Barr and Charles Wells. *Category Theory for Computing Science.* Prentice Hall, 1990.

[Cou90] P. Cousot. Methods and logics for proving programs. In van Leeuwen [vL90], chapter 15, pages 814–994.

[Cro93] Roy Crole. *Categories for Types.* Cambridge University Press, 1993.

[DF+93] G. Dowek, A. Felty, et al. The Coq proof assistant user's guide. Technical report, Project Formel, INRIA Rocquencourt, February 1993.

[DJ90] P. Deransart and M. Jourdan, editors. *Attribute Grammars and their Applications*, volume 461 of *Lecture Notes in Computer Science.* Springer Verlag, 1990.

[EM90] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints.* Springer Verlag, 1990.

[FS88] José Fiadeiro and Amilcar Sernadas. Structuring theories on consequence. In D. Sanella and A. Tarlecki, editors, *Recent Trends in Data Type Specification*, volume 332 of *Lecture Notes in Computer Science*, pages 44–72. Springer Verlag, 1988.

[GB85] J.A. Goguen and R.M. Burstall. A study in the foundations of programming methodology : Specifications, institutions, charters and parchments. In D.H. Pitt, S. Abramsky, A. Poigné, and D.E. Rydeheard, ed-

itors, *Category Theory and Computer Programming*, volume 240 of *Lecture Notes in Computer Science*, pages 313–333. Springer Verlag, 1985.

[GB92] J.A. Goguen and R.M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the A.C.M.*, 39(1):95–146, January 1992.

[GJ90] D. Grune and C.J.H. Jacobs. *Parsing Techniques: a practical guide.* Ellis Horwood, 1990.

[Gog89] Joseph Goguen. A categorical manifesto. Technical Monograph PRG-72, Oxford University Computing Laboratory, March 1989.

[GTW78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R.T. Yeh, editor, *Current Trends in Programming Methodology*, volume IV, chapter 5, pages 80–149. Prentice Hall, 1978.

[Hen90] Matthew Hennessy. *The Semantics of Programming Languages.* Wiley, 1990.

[HST89a] R .Harper, D. Sanella, and A. Tarlecki. Logic representation in LF. In D.H. Pitt, D.E. Rydeheard, P. Dybjer, A.M. Pitts, and A. Poigné, editors, *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 250–272. Springer Verlag, 1989.

[HST89b] R. Harper, D. Sannella, and A. Tarlecki. Structure and representation in LF. In *Fourth Annual Symposium on Logic in Computer Science* [IEE89], pages 226–237.

[IEE89] IEEE. *Logic in Computer Science.* IEEE Computer Society Press, jun 1989.

[Kli93] P. Klint. The ASF+SDF meta-environment user's guide. Technical report, CWI, Amsterdam, 1993.

[Kos91] K. Koster. Affix grammars for programming languages. In Alblas and Melichar [AM91], pages 358–373.

[LMB92] J. Levine, T. Mason, and D. Brown. *Lex and Yacc.* O'Reilly, 1992.

[LS86] J. Lambek and P.J. Scott. *Introduction to Higher Order Categorical Logic.* Cambridge Studies in Advanced mathematics; 7. Cambridge University Press, 1986.

[MA86] E.G. Manes and M.A. Arbib. *Algebraic Approaches to Program Semantics.* Springer Verlag, 1986.

[Mac71] Saunders MacLane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics.* Springer Verlag, 1971.

[May85] Brian Mayoh. Gallaries and institutions. Technical Report DAIMI PB-191, Aarhus University, Computer Science Department, March 1985.

[Mes89] José Meseguer. General logics. In H.D. Ebbinghaus et al., editors, *Logic Colloquium 1987*, Studies in Logic and the Foundations of Mathematics, 129, pages 275–329. Elsevier Science Publishers B.V. (North-Holland), 1989.

[Mos89] Peter Mosses. Unified algebras and institutions. In *Fourth Annual Symposium on Logic in Computer Science* [IEE89], pages 304–312.

[Pag81] F.G. Pagan. *Formal Specification of Programming languages.* Prentice Hall, 1981.

[Pau90] L.C. Paulson. A formulation of the simple theory of types (for Isabelle). In P. Martin-Löf and G. Mints, editors, *COLOG-88 (Intl. Conf. on Computer Logic)*, volume 417 of *Lecture Notes in Computer Science*, pages 246–274. Springer Verlag, 1990.

[Pie91] B.C. Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, 1991.

[Poi89] Axel Poigné. Foundations are rich institutions, but institutions are poor foundations. In H. Ehrig, H. Herllich, H.J. Kerowski, and G. Pleuss, editors, *Categorical Methods in Computer Science*, volume 393 of *Lecture Notes in Computer Science*, pages 82–101. Springer Verlag, 1989.

[PP92] T. Pittman and J. Peters. *The Art of Compiler Design.* Prentice Hall, 1992.

[RB88] D.E. Rydeheard and R.M. Burstall. *Computational Category Theory.* Prentice Hall, 1988.

[Reg90] Gianna Reggio. Entities: an institution for dynamic systems. In H. Ehrig, K.P. Jantke, F.Orejas, and H.Reichel, editors, *Recent Trends in Data Type Specification*, volume 534 of *Lecture Notes in Computer Science*, pages 246–265. Springer Verlag, 1990.

[RR92] R.Burstall and R.Diaconescu. Hiding and behaviour: an institutional approach. Technical report, Programming Research Group, Oxford, 1992.

[Sch86] David A. Schmidt. *Denotational Semantics: a methodology for language development.* Allyn and Bacon, 1986.

[Sco74] Dana Scott. Completeness and axiomatizability in many-valued logic. In *Proceedings of the Tarski Symposium*, pages 411–435, Providence, R.I., 1974. A.M.S.

[Spi88] J.M. Spivey. *Understanding Z: A specification language and its formal semantics.* Cambridge University Press, 1988.

[Spi89] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1989.

[ST84] Donald Sanella and Andrzej Tarlecki. Building specifications in an arbitrary institution. In G. Khan, D.B. MacQueen, and G. Plotkin, editors, *Symposium on Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 337–356. Springer Verlag, 1984.

[Ste92] Petros Stefaneas. The modal charter. Technical Report PRG-TR-29-92, Oxford University Computing Lab., 1992.

[Ten91] R.D. Tennent. *Semantics of Programming Languages.* Prentice Hall, 1991.

[vL90] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science.* Elsevier, 1990.

[Wat91] David A. Watt. *Programming Language Syntax and Semantics.* Prentice Hall, 1991.