Z
699
C3
no. 90-15

Algebraic Specification: Syntax, Semantics, Structure

Yellamraju V. Srinivas
Department of Information and Computer Science
University of California, Irvine, USA
srinivas@ics.uci.edu

Technical Report 90-15
19 June 1990

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Abstract

Algebraic specification is the technique of using algebras to model properties of a system and using axioms to characterize such algebras. Algebraic specification comprises two aspects: the underlying logic used in the axioms and algebras, and the use of a small, general set of operators to build specifications in a structured manner. We describe these two aspects using the unifying notion of institutions. An institution is an abstraction of a logical system, describing the vocabulary, the kinds of axioms, the kinds of algebras, and the relation between them. Using institutions, one can define general structuring operators which are independent of the underlying logic. In this paper, we survey the different kind of logics, syntax, semantics, and structuring operators that have been used in algebraic specification.

Contents

1 INTRODUCTION **1**
- 1.1 Data types and algebras . . . 1
- 1.2 Outline . . . 4
- 1.3 Background . . . 4
- 1.4 Acknowledgements . . . 4

2 LOGICS FOR ALGEBRAIC SPECIFICATION **5**
- 2.1 Equational logic . . . 5
- 2.2 First-order logic . . . 14
- 2.3 Inequations . . . 16
- 2.4 Conditional equations . . . 17
- 2.5 Quantifiers . . . 18
- 2.6 Predicate calculus . . . 19
- 2.7 Hidden functions . . . 22
- 2.8 Partial first-order logic . . . 23
- 2.9 Order-sorted logic . . . 26
- 2.10 Other logics . . . 31
- 2.11 Concluding remarks on logics . . . 31

3 SEMANTICS **32**
- 3.1 Initial semantics . . . 32
- 3.2 Final semantics . . . 36
- 3.3 Loose semantics . . . 39
  - 3.3.1 Existence of initial and terminal models . . . 42
- 3.4 Behavioural semantics . . . 44
- 3.5 Operational semantics: Deduction and rewriting . . . 46

4 STRUCTURE **48**
- 4.1 Institutions . . . 48
  - 4.1.1 Why all this abstractness? . . . 53
- 4.2 Institution-independent operations . . . 53
  - 4.2.1 An example: Parity automaton . . . 61
  - 4.2.2 Remarks on structuring operations . . . 66
- 4.3 Parameterization . . . 66
- 4.4 Modules . . . 79
- 4.5 Constraints . . . 83
- 4.6 Vertical structuring and algebraic implementation . . . 88
- 4.7 Concluding remarks on structure . . . 93

5 LANGUAGES **94**
- 5.1 ACT-ONE . . . 94
- 5.2 ASL . . . 96
- 5.3 CIP-L . . . 98
- 5.4 Clear . . . 99
- 5.5 OBJ . . . 100

5.6 Larch 101
5.7 Pluss 103

6 SUMMARY AND CONCLUDING REMARKS **105**
6.1 Evolution of algebraic specification 105
6.2 Some limitations 106
6.3 Specifying-in-the-large 106
6.4 Future directions 106

7 BIBLIOGRAPHIC NOTES **108**

8 REFERENCES **114**

List of Figures

1 Kinds of signatures and axioms 6
2 Models of a specification 33
3 Pieces of an institution 50
4 General operations for building specifications 55
5 Summary of language features 95

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*languages*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics, syntax*; D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*specification techniques*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*model theory*

General Terms: Languages, Theory

Additional Key Words and Phrases: Abstract data types, signatures, algebras, algebraic specification, institutions, abstract model theory, structuring operations, categories, functors, limits, colimits, freeness, adjoints

1 INTRODUCTION

Algebraic specification has been around for about fifteen years, and there have been many papers, theses, and books on particular aspects. However, there has not been a survey or overview which is accessible to the non-specialist. In this paper, we attempt to bring together the various strands of research into a coherent framework, thus exposing their relationships and the fundamental principles of algebraic specification.

Algebraic specification is the technique of using algebras (collections of sets and operations on these sets) to model properties of a system and using axioms to characterize such algebras. Different techniques use different kinds of axioms—equations, Horn clauses, etc. Each kind of axiom requires an underlying logic to impart meaning to the axiom. The concept of an institution generalizes the notion of a "logical system" in which such axioms can be defined, enabling the study of algebraic specification independent of the logical system used. An institution abstractly characterizes a logic by describing the syntax of specifications, the kinds of algebras, the kinds of axioms, and the relationships between these.

Algebraic specification is also concerned with the task of building specifications in a modular way from other, smaller specifications. Each algebraic specification language has its own set of "building" operations. Using institutions, we can abstract away from specific languages and identify certain basic structuring mechanisms, which can be studied in an institution independent way. We will describe a few simple operations for building specifications and also see their manifestations in several algebraic specification languages.

1.1 Data types and algebras

Algebraic specification begins with the (largely unstated) assumption that everything of interest is a type (at least in the artificial reality [Simon 70] that is the subject matter of computer science).[1] Following this assumption are two other assumptions: that types are algebras and that these algebras can be characterized axiomatically.

The simplest view of types is that a type is a *set* or collection of values. However, this is generally not sufficient to completely characterize the behaviour of the type; a type exhibits different sets of properties depending on the way it *used.* For example, consider a set of values arranged linearly as a list, with operations for accessing, inserting, deleting, or replacing elements in the list. Depending on the operations allowed, this same set of values behaves differently:

- if all accesses, insertions and deletions are restricted to one end of the list, then we call it a *stack*;
- if all insertions are restricted to one end and deletions and accesses to the other end, then we call it a *queue*;
- if only accesses and replacements are permitted, but their position is unrestricted, we call it an *array.*

[1]There has been no serious attempt to study the limits of expressiveness of algebraic specification techniques, except of course the specifiability of computable algebras.

To quote Knuth [Knuth 68, p. 234]:

> ...we must decide in each case how much structure to represent ..., and how accessible to make each piece of information. To make this decision, we need to know what operations are to be performed on the data. ... *we therefore consider not only the data structure but also the class of operations to be done on the data;* the design of computer representations depends on the desired function of the data as well as on its intrinsic properties. Such an emphasis on "function" as well as "form" is basic to design problems in general. [author's italics]

It appears that it would be fruitful to view data types as algebras, collections of sets together with operations on them. Once we have decided that data types are algebras, we need to find a way to describe algebras. One way is to follow the approach of universal algebra [Cohn 81; Grätzer 79; Birkhoff and Lipson 70], wherein classes of similar algebras are abstractly defined using formal names for the components of an algebra and axioms to be satisfied by these. An advantage of such an axiomatic method is that we can abstractly characterize the properties of a data type independent of its representation.

Pioneering work along these lines was done by Goguen et al., Guttag, Liskov, and Zilles [Goguen et al. 78; Guttag 75; Liskov and Zilles 75; Zilles 79]. After this start, a lot of research has been done in the past fifteen years on the algebraic specification of data types: different kinds of algebras, different kinds of axioms, different notions of "abstract" and "representation independent", the expressive power of different axiom systems, the problem of building large specifications, the problem of "executing" specifications, etc. In this paper, we survey the important aspects of the research in algebraic specification.

An example: The ring of integers

We will informally touch upon the issues involved in specifying an algebra, using the example algebra of integers. The integers form a ring with the usual operations of addition and multiplication.

DEFINITION 1.1: *Ring.* A ring is an algebraic structure with the following components

A non-empty set R, with two distinguished elements called zero, 0, and unit, 1

Two binary operations on R, called addition, $+$, and multiplication, $\times$

subject to the following axioms

1. The structure $\langle R, +, 0\rangle$ is an abelian group, i.e.,
   (a) (Associativity) $\forall a, b, c \in R \cdot (a + b) + c = a + (b + c)$
   (b) (Commutativity) $\forall a, b \in R \cdot a + b = b + a$
   (c) (Identity) $\forall a \in R \cdot a + 0 = a = 0 + a$
   (d) (Inverse) $\forall a \in R \cdot \exists(-a) \in R \cdot a + (-a) = 0 = (-a) + a$
2. The structure $\langle R, \times, 1\rangle$ is a monoid,[2] i.e.,
   (a) (Associativity) $\forall a, b, c \in R \cdot (a \times b) \times c = a \times (b \times c)$
   (b) (Identity) $\forall a \in R \cdot a \times 0 = a = 0 \times a$
3. Multiplication is distributive over addition, i.e.,
   (a) (Left) $\forall a, b, c \in R \cdot a \times (b + c) = (a \times b) + (a \times c)$
   (b) (Right) $\forall a, b, c \in R \cdot (a + b) \times c = (a \times c) + (b \times c)$

□

Thus an algebra is specified in two parts: (1) a *signature* containing the names of some sets and symbols standing for operations on these sets, and (2) axioms to be satisfied by these operations. Axioms can take a variety of forms. Usually they are universally quantified equations such as the axioms for associativity and commutativity above. In general, they can be formulas in first-order logic. For example, the axiom for inverse above uses existential quantification.

The integers, along with addition and multiplication, satisfy the ring axioms outlined above. They are said to be a *model* of the specification. Are they the only model? The answer is no. There are a number of other models: rational numbers, complex numbers, the integers modulo n, for any integer n, polynomials, etc. However, the integers do have a distinguished property, *initiality*, among all the models. In subsequent sections of the paper, we will use category theory to define a very general notion of signature, axiom, model, and satisfaction, and describe their manifestations in various logical systems.

[2] We follow [Mac Lane and Birkhoff 67] in requiring a multiplicative identity for a ring. Other authors assume that $\langle R, \times\rangle$ forms a semigroup, in which case, a ring as defined above would be called a ring with identity.

1.2 Outline

The paper is organized around the concepts of institutions (abstract logical systems) and institution-independent structuring operators for building large algebraic specifications. Institutions are introduced in section 4.1. Before that, we give several examples of logics in which specifications can be written, section 2, starting with equational logic and proceeding towards more powerful and expressive systems. In section 3, we describe the different kinds of semantics one can adopt for specifications. Section 4 deals with the structure of specifications, which can be studied independent of the logic or institution used. In section 4.2, we introduce the core structuring operations for algebraic specification. Sections 4.4 through 4.6 consider other techniques for building specifications, including parameterized specifications and modules. In section 5, we discuss the specific structuring operations which are present in several specification languages. We conclude in section 6 with a summary and some discussion.

Readers wishing to obtain an overview of algebraic specification before pursuing details may do so by choosing one or two subsections from each major section of this paper: for example, equational logic from section 2, initial semantics from section 3, informal remarks about institutions in sections 4.1, a few operations for building algebraic specifications in section 4.2, and one of the languages in section 5, say Pluss. Also, figures 1–5 concisely display the concepts covered in this survey.

1.3 Background

Although algebraic specification can be treated in a purely algebraic framework, ideas from category theory considerably simplify the theory. An example is the generalization afforded by the concept of institutions, which is formulated in terms of category theory. We do not describe category theory in this paper; the relevant concepts can be found in a companion paper [Srinivas 90]. We have organized the paper in such a way that the use of more advanced aspects of category theory is localized.

For a detailed description of category theory, we refer the reader to the literature; the notes in section 7 provide some pointers.

1.4 Acknowledgements

It is a pleasure to express my gratitude to my advisor, Peter Freeman, for constant encouragement and support, to David Rector for answering many questions about category theory and algebra, and to Ira Baxter for many fruitful discussions on algebraic specification. This work was supported in part by the National Science Foundation under the CER grant CCR-8521398.

2 LOGICS FOR ALGEBRAIC SPECIFICATION

The syntax of specifications comprises two aspects: signatures, which provide names for the components of an algebra, and axioms, which describe properties to be satisfied by an algebra. The form of specifications is dictated by the form of the algebras which are being described. Algebras are collections of sets with operations on them. There are three kinds of operations: total functions, partial functions, and relations. In addition, we can impose certain relationships between the carrier sets of the algebra, e.g., that one carrier set is a subset of another. Order-sorted algebras and higher-order algebras result from imposing such relationships. With each kind of algebra, the axioms used can vary from equations, conditional equations, and Horn clauses, to sentences in first-order logic and even higher-order logic.

Figure 1 summarizes various kinds of logic and provides references to representative articles and books using them. Each of these logical systems can be abstractly described as an institution (see section 4.1). In this section, we will describe equational logic, first-order logic, predicate calculus, partial first-order logic, and order-sorted logic.

A note about definitions. We give below several different definitions of concepts such as signature, sentence, and model, because they vary with the logic under consideration. Each of these definitions applies only to the logic being described in a particular subsection. Subsequently, section 3 onwards, we will use concepts such as signature, sentence, and model, in a sense which is independent of the logic or institution.

2.1 Equational logic

Equational logic is the simplest and most commonly used logic for algebraic specification. The operations are total functions and the axioms are equations. Equational logic is borrowed from universal algebra [Cohn 81; Grätzer 79]; however, in algebraic specification, the many-sorted version, first introduced in [Birkhoff and Lipson 70]), is used.

DEFINITION 2.1: *Signature.* A signature $\Sigma = \langle S, \Omega \rangle$, consists of a set S of *sorts* and a set Ω of *operation symbols.* Associated with each operation symbol is a sequence of sorts called its *rank.* For example, $f: s_1, s_2, \ldots, s_n \rightarrow s$ indicates that f is the name of an n-ary function, taking arguments of sorts $s_1, s_2, \ldots, s_n$ and producing a result of sort s. A nullary operation symbol, $c: \rightarrow s$, is called a *constant* of sort s. □

NOTATION. Specification names and sort names are typeset in SMALL CAPITALS, and operation names in Roman. To improve readability, we use mixfix notation for some of the operations. This is indicated by underscores in the signature of the operation. For example, " $\langle _, _ \rangle: \alpha, \beta \rightarrow$ PAIR " indicates that $\langle _, _ \rangle$ is a binary operation which expects an element of sort α in its first (left) slot and an element of sort β in its second (right) slot. Comments are enclosed in /* ... */.

Signatures	Axioms		
	Equations	Conditional Equations	Formulas with quantifiers
Total Functions	Equational Logic [Goguen et al. 78] [Ehrig and Mahr 85]	Conditional Equational Logic [Thatcher et al. 82]	
Partial Functions	Partial Equational Logic [Reichel 87]		Partial First-order Logic [Wirsing et al. 83] [Broy and Wirsing 82]
Relations		Horn-Clause Logic [Padawitz 88]	Predicate Calculus [Turski and Maibaum 87] [Bidoit 89]
Order-Sorted		Order-Sorted Logic [Goguen and Meseguer 88] [Goguen 89]	
Higher-order	Higher-order Equational Logic [Parsaye-Ghomi 82]		Higher-order Logic [Möller 87, Broy 87]
Continuous Functions	Equational Logic [Levy and Maibaum 82] [Goguen et al. 77]	Conditional Equational Logic [Möller 85] [Tarlecki and Wirsing 86]	

Figure 1: Kinds of signatures and axioms

EXAMPLE 2.2. Here is the signature for a ring. The signature has two binary operations, addition (+) and multiplication (×), together with two constants (0 and 1) which are their identities. We also introduce a unary operation, −, which assigns to each element its additive inverse. This is so that the axiom for inverse can be expressed as an equation (instead of using existential quantification as in example 1.1).

```
signature SIG-RING =
sorts R
operations
  _ + _ : R, R → R      /* addition */
      0 :       → R      /* additive identity */
      − :    R → R      /* additive inverse */
  _ × _ : R, R → R      /* multiplication */
      1 :       → R      /* multiplicative identity */
end
```

□

DEFINITION 2.3: *Terms.* Let $X = \{ X_s \mid s \in S \}$ be a set of *variables* indexed by S, i.e., each variable is associated with a sort. The set of terms generated by the signature Σ using the variables X,[3] denoted by $T_\Sigma(X)$, is defined as the indexed family

$$T_\Sigma(X) = \{ T_{\Sigma,s}(X) \mid s \in S \}$$

where $T_{\Sigma,s}(X)$, the set of terms of sort s, is defined inductively as follows:

1. if x is a variable of sort s, then $x \in T_{\Sigma,s}(X)$;
2. if c is a constant symbol of sort s, then $c \in T_{\Sigma,s}(X)$;
3. if f is an operation with rank $s_1, s_2, \ldots, s_n \to s$, and $t_1, t_2, \ldots, t_n$ are terms in $T_{\Sigma,s_1}(X)$, $T_{\Sigma,s_2}(X)$, $\ldots$, $T_{\Sigma,s_n}(X)$ respectively, then $f\underline{(}t_1, t_2, \ldots, t_n\underline{)} \in T_{\Sigma,s}(X)$.

□

When $X = \emptyset$, the set of terms $T_\Sigma(X)$ is called the set of *ground* terms and denoted by T_Σ.[4] Note that the parentheses in $f\underline{(}t_1, t_2, \ldots, t_n\underline{)}$ do not indicate evaluation; they are just formal symbols used to construct the term. When no confusion is likely, we omit the underlines, as is usual in writing the axioms of a specification.

EXAMPLE 2.4. Consider the signature SIG-RING of example 2.2 and variables a, b, c of sort R. Then $0, 0 + 1, 1 \times \underline{(}-1 + 0\underline{)}, a \times b, \underline{(}-a \times b\underline{)} + c$ are all terms, the first three being ground terms. □

[3] We assume that the variables are disjoint from the operation symbols in the signature.

[4] Thus we have $T_\Sigma(X) = T_{\Sigma(X)}$, where $\Sigma(X)$ is the signature Σ augmented by constant operations corresponding to the variables in X.

DEFINITION 2.5: *Equation.* Given a signature Σ, a Σ-equation is defined to be a triple $\langle X, l, r\rangle$, where X is a set of (sorted) variables and l and r are terms of the same sort s, i.e., $l, r \in T_{\Sigma,s}(X)$. □

NOTATION. As is usual in the algebraic specification literature, we write an equation $\langle X, l, r\rangle$ as $l = r$, omitting the variables. The variable declarations may be either left implicit, or all variables declared together at the beginning of a set of equations. This avoids clutter and normally does not entail any difficulties. However, as we see in section 3.5, the completeness of equational logic depends on rigorously writing down equations as triples. Also note that the "=" symbol in an equation is a syntactic symbol to separate the terms on its left and its right; it is not an equality predicate.

DEFINITION 2.6: *Specification.* A specification is a pair $\langle \Sigma, \mathcal{E}\rangle$ of a signature Σ and a set $\mathcal{E}$ of Σ-equations. □

Assumption. Unless otherwise stated, a specification is assumed to consist of *finite* sets of sorts, operations, and axioms.

EXAMPLE 2.7. Here is the complete specification of a ring. In subsequent specification, we will omit variable declarations in view of readability. We follow the convention that variables are typeset in *italics*.

spec RING =
sorts R
operations

$_ + _ : \mathrm{R}, \mathrm{R} \rightarrow \mathrm{R}$ /* addition */
$0 : \quad \rightarrow \mathrm{R}$ /* additive identity */
$- : \mathrm{R} \rightarrow \mathrm{R}$ /* additive inverse */
$_ \times _ : \mathrm{R}, \mathrm{R} \rightarrow \mathrm{R}$ /* multiplication */
$1 : \quad \rightarrow \mathrm{R}$ /* multiplicative identity */

axioms

/* $\langle \mathrm{R}, +, -, 0 \rangle$ forms an abelian group */
$\forall a, b, c \in \mathrm{R} \cdot a + (b + c) = (a + b) + c$
$\forall a, b \in \mathrm{R} \cdot a + b = b + a$
$\forall a \in \mathrm{R} \cdot a + 0 = a$
$\forall a \in \mathrm{R} \cdot a + (-a) = 0$
/* $\langle \mathrm{R}, \times, 1 \rangle$ forms a monoid */
$\forall a, b, c \in \mathrm{R} \cdot a \times (b \times c) = (a \times b) \times c$
$\forall a \in \mathrm{R} \cdot a \times 1 = a$
$\forall a \in \mathrm{R} \cdot 1 \times a = a$
/* distributivity of $\times$ over + */
$\forall a, b, c \in \mathrm{R} \cdot a \times (b + c) = (a \times b) + (a \times c)$
$\forall a, b, c \in \mathrm{R} \cdot (a + b) \times c = (a \times c) + (b \times c)$

end

□

DEFINITION 2.8: *Algebra.* Given a signature $\Sigma = \langle S, \Omega \rangle$, a Σ-algebra $A = \langle A_S, F_A \rangle$ consists of two families:

1. a collection of sets, called the *carriers* of the algebra, $A_S = \{ A_s \mid s \in S \}$; and
2. a collection of (total) functions, $F_A = \{ f_A \mid f \in \Omega \}$, such that, if the rank of f is $s_1, s_2, \ldots, s_n \rightarrow s$, then f_A is a function from $A_{s_1} \times A_{s_2} \times \cdots \times A_{s_n}$ to A_s. The symbol $\times$ indicates the cartesian product of sets here.

□

If $c\colon \rightarrow s$ is a constant in the signature Σ, then condition (2) above implies that there is a constant element c_A in the carrier A_s.

When no confusion is likely, we denote the carriers of an algebra A simply by A, rather than A_S.

DEFINITION 2.9: *Evaluation of terms.* Given a signature $\Sigma = \langle S, \Omega \rangle$, a set X of variables indexed by S, and a Σ-algebra A, an *assignment* of values in A to the variables in X is an indexed collection of functions, $\alpha: X \rightarrow A = \{\alpha_s: X_s \rightarrow A_s \mid s \in S\}$. An assignment for variables can be extended to an assignment for terms $\bar{\alpha}: T_\Sigma(X) \rightarrow A = \{\bar{\alpha}_s: T_{\Sigma,s}(X) \rightarrow A_s \mid s \in S\}$ as follows:

1. if x is a variable of sort s, then $\bar{\alpha}_s(x) = \alpha_s(x)$;
2. if c is a constant symbol of sort s, then $\bar{\alpha}_s(c) = c_A$;
3. if f is an operation with rank $s_1, s_2, \ldots, s_n \rightarrow s$, and $t_1, t_2, \ldots, t_n$ are terms in $T_{\Sigma,s_1}(X)$, $T_{\Sigma,s_2}(X)$, ..., $T_{\Sigma,s_n}(X)$ respectively, then $\bar{\alpha}_s(f\underline{(}t_1, t_2, \ldots, t_n\underline{)}) = f_A(\bar{\alpha}_{s_1}(t_1), \bar{\alpha}_{s_2}(t_2), \ldots, \bar{\alpha}_{s_n}(t_n))$.

□

Note that in case (3) above, while the parentheses in $f\underline{(}\ldots\underline{)}$ are formal symbols indicating term construction, those in $f_A(\ldots)$ indicate evaluation of the function.

EXAMPLE 2.10. Consider the signature SIG-RING of example 2.2, variables a, b, c of sort R, and the standard algebra of integers. Let α be the assignment $\{a \rightarrow -1, b \rightarrow 4, c \rightarrow 5\}$. Then $\bar{\alpha}(\underline{(}-a + b\underline{)} \times c) = 25$. □

DEFINITION 2.11: *Satisfaction.* A Σ-algebra A is said to *satisfy* an equation $\langle X, t_1, t_2 \rangle$ if for all assignments $\alpha: X \rightarrow A$ of values to variables, $\bar{\alpha}(t_1) = \bar{\alpha}(t_2)$. □

NOTATION. We write $A \models e$ to indicate that an equation e is satisfied by an algebra A,

DEFINITION 2.12: *Model.* Given a specification $\langle \Sigma, \mathcal{E} \rangle$, a model of the specification is a Σ-algebra which satisfies *each* of the equations in the set $\mathcal{E}$. □

EXAMPLE 2.13. The standard algebra of integers, with carrier $Z = \{\cdots, -1, -2, 0, 1, 2, \cdots\}$ along with addition and multiplication, forms a model of the specification RING shown above. The set of integers modulo 2, $Z_2 = \{0, 1\}$, is also a model with the operations as defined below:

$+_2$	0	1
0	0	1
1	1	0

$\times_2$	0	1
0	0	0
1	0	1

$-_2$	
0	0
1	1

□

DEFINITION 2.14: *Homomorphism.* Given a signature $\Sigma = \langle S, \Omega \rangle$ and two Σ-algebras A and B, a Σ-homomorphism $h: A \to B$ is a family of functions $\{ h_s: A_s \to B_s \mid s \in S \}$ between the carriers which are compatible with the operations, i.e., for all operation symbols $f: s_1, s_2, \ldots, s_n \to s$, and for all $a_1 \in A_{s_1}, a_2 \in A_{s_2}, \ldots, a_n \in A_{s_n}$,

$$h_s(f_A(a_1, a_2, \ldots, a_n)) = f_B(h_{s_1}(a_1), h_{s_2}(a_2), \ldots, h_{s_n}(a_n)).$$

□

EXAMPLE 2.15. Consider again the standard integer algebra, $\langle Z, +, 0, -, \times, 1 \rangle$, and the integers modulo 2, $\langle Z_2, +_2, 0, -_2, \times_2, 1 \rangle$. The function which assigns to each integer its remainder modulo 2 is a homomorphism $h: Z \to Z_2$. A proof that this function is compatible with the operations in the two algebras, and thus is a homomorphism, can be found in [Mac Lane and Birkhoff 67, Chapter II]. □

The purpose of a signature is to specify syntactic names for the sets and operations of an algebra so that axioms about the algebra may be constructed. As with any syntactic mechanism, we expect that the particular names chosen not be important, only that they be used consistently. Thus, for example, we could have used the following signature for specifying rings, rather than the one in definition 1.1.

```
signature SIG-RING-1 =
sorts ANY
operations
    plus : ANY, ANY → ANY        /* addition */
    zero :          → ANY        /* additive identity */
     inv :      ANY → ANY        /* additive inverse */
   times : ANY, ANY → ANY        /* multiplication */
     one :          → ANY        /* multiplicative identity */
end
```

To define the relation between the signatures SIG-RING and SIG-RING-1, we need the notion of a signature morphism.

DEFINITION 2.16: *Signature morphism.* Given two signatures $\Sigma = \langle S, \Omega \rangle$ and $\Sigma' = \langle S', \Omega' \rangle$, a signature morphism $\sigma: \Sigma \to \Sigma'$ is a pair of functions $\langle \sigma_S: S \to S', \sigma_\Omega: \Omega \to \Omega' \rangle$ such that the ranks of the operations are preserved, i.e.,

for all operation symbols $f: s_1, s_2, \ldots, s_n \to s$ in Ω,

the operation symbol $\sigma_\Omega(f): \sigma_S(s_1), \sigma_S(s_2), \ldots, \sigma_S(s_n) \to \sigma_S(s)$ is in Ω'.

□

NOTATION. We ambiguously write σ for the two parts, σ_S, σ_Ω, of a signature morphism. The context usually determines which is intended.

EXAMPLE 2.17. The signature morphism σ: SIG-RING $\rightarrow$ SIG-RING-1 is defined by the following map:

$$\{\mathrm{R} \mapsto \mathrm{ANY},\ + \mapsto \text{plus},\ 0 \mapsto \text{zero},\ - \mapsto \text{inv},\ \times \mapsto \text{times},\ 1 \mapsto \text{one}\}$$

□

A bijective signature morphism such as the one above is just a renaming operation. Non-bijective morphisms are somewhat more interesting. An into (non-surjective) morphism can be used to "hide"[5] some sorts and operations, those that are not in the image of the morphism. A many-one (non-injective) morphism can be used to "coalesce" some sorts and operations.

EXAMPLE 2.18. Let SIG-GROUP be the signature for groups:

signature SIG-GROUP =
sorts G
operations
$_ * _ : \mathrm{G}, \mathrm{G} \rightarrow \mathrm{G}$ /* addition */
$\epsilon : \ \rightarrow \mathrm{G}$ /* identity */
$(_)' : \mathrm{G} \rightarrow \mathrm{G}$ /* inverse */
end

We can define two signature morphisms between SIG-GROUP and SIG-RING as follows:

$\sigma_{\mathrm{G}\rightarrow\mathrm{R}}$: SIG-GROUP $\rightarrow$ SIG-RING	$\sigma_{\mathrm{R}\rightarrow\mathrm{G}}$: SIG-RING $\rightarrow$ SIG-GROUP
$\mathrm{G} \mapsto \mathrm{R}$	$\mathrm{R} \mapsto \mathrm{G}$
$* \mapsto +$	$+ \mapsto *$
$\epsilon \mapsto 0$	$0 \mapsto \epsilon$
$(_)' \mapsto -$	$- \mapsto (_)'$
	$\times \mapsto *$
	$1 \mapsto \epsilon$

To see the effect of these signature morphisms on algebras, see example 2.22 below. □

When a signature morphism is applied to a specification, the axioms in the specification should also be appropriately translated. We now define translation of equations.

[5]The term "forget" is also used in the literature to indicate sorts and operations which are not accessible.

DEFINITION 2.19: *Translated equation.* Given two signatures $\Sigma = \langle S, \Omega \rangle$ and $\Sigma' = \langle S', \Omega' \rangle$, a signature morphism $\sigma: \Sigma \to \Sigma'$, and a Σ-equation $\langle X, l, r \rangle$, the translated equation is given by $\langle X', \bar{\sigma}(l), \bar{\sigma}(r) \rangle$, where

1. $X' = \{ X'_{s'} \mid s' \in S' \}$ with $X'_{s'} = \bigcup \{ X_s \mid s \in S \text{ and } \sigma(s) = s' \}$;
2. if x is a variable, then $\bar{\sigma}(x) = x$;
3. if f is an n-ary operation symbol in Σ,
   then $\bar{\sigma}(f\underline{(}t_1, t_2, \ldots, t_n\underline{)}) = \sigma(f)\underline{(}\bar{\sigma}(t_1), \bar{\sigma}(t_2), \ldots, \bar{\sigma}(t_n)\underline{)}$.

□

EXAMPLE 2.20. Given the signature morphism σ: SIG-RING → SIG-RING-1 of example 2.17, the equation

$$\langle \{a, b, c\colon \mathrm{R}\}, a \times (b + c), (a \times b) + (a \times c) \rangle$$

would translate into

$$\langle \{a, b, c\colon \mathrm{ANY}\}, \mathrm{times}(a, \mathrm{plus}(b, c)), \mathrm{plus}(\mathrm{times}(a, b), \mathrm{times}(a, c)) \rangle$$

□

We now define the effect of a signature morphism on the models of the signatures involved. This is akin to translation of equations; however, models are translated in the opposite direction.

DEFINITION 2.21: *Reduct.* Given a signature morphism $\sigma: \Sigma \to \Sigma'$, and a Σ'-algebra A', the σ-reduct of A', denoted by $A'|_\sigma$, is the Σ-algebra $A = \langle A_S, F_A \rangle$ defined as follows (with $\Sigma = \langle S, \Omega \rangle$):

$$A_s = A'_{\sigma(s)}, \text{ for } s \in S, \text{ and } f_A = (\sigma(f))_{A'}, \text{ for } f \in \Omega.$$

Given a Σ'-homomorphism $h': A' \to B'$ between two Σ'-algebras A' and B', the σ-reduct of h' is a Σ-homomorphism $h: A'|_\sigma \to B'|_\sigma$, denoted by $h'|_\sigma$, and defined by the family of functions $h_s = h'_{\sigma(s)}$, for $s \in S$. □

EXAMPLE 2.22. Consider the signature morphisms $\sigma_{\mathrm{G}\to\mathrm{R}}$ and $\sigma_{\mathrm{R}\to\mathrm{G}}$ of example 2.18. The $\sigma_{\mathrm{G}\to\mathrm{R}}$-reduct of any SIG-RING-algebra is obtained by ignoring the extra operations $\times$, and 1. This corresponds to the fact that the additive part of every ring is a group. The $\sigma_{\mathrm{R}\to\mathrm{G}}$-reduct of any SIG-GROUP-algebra is obtained by duplicating the operations in a group to act both as addition and multiplication in a ring. □

Summary

Let us recapitulate the various components of equational logic which have been discussed in this section. The main entities of concern are algebras, collections of sets and functions. Signatures provide names for these sets and functions. Signatures also serve as the basis

for generating terms, which in turn are used to construct equations. There is a satisfaction relation between equations and algebras, derived from the fact that terms can be interpreted as values in the algebra and that equations can be interpreted as imposing an equality between values. Specifications are signatures together with equations. Associated with a signature is a collection of algebras. The equations in a specification pick out a subset of these algebra. Finally, the names provided by a signature can be changed, inducing a corresponding translation of terms, equations, and algebras.

This kind of structure—signatures, terms, axioms, algebras, satisfaction, and the corresponding translations—is common to all the logics we will consider in subsequent sections. The notion of institution generalizes and abstractly describes this structure of a logic (see section 4.1).

2.2 First-order logic

Equational logic is sufficient for describing structures such as groups and rings in mathematics. However, it is not expressive enough for data structures, e.g., queues, some of whose functions are conditionally or partially defined. We now consider a very general form of axioms, sentences in first-order logic with equality.[6] We only describe sentences and satisfaction. Signatures and algebras are the same as those of equational logic.

DEFINITION 2.23: *Free variables.* Given a signature Σ, and a set of sorted variables X, the set $T_\Sigma(X)$ of all Σ-terms with variables in X is defined as for equational logic (definition 2.3). The set of *free* variables occuring in a term t, denoted by $\mathcal{F}(t)$, is defined as follows:

1. if t is a variable, $\mathcal{F}(t) = \{t\}$;
2. if t is a constant symbol, then $\mathcal{F}(t) = \emptyset$;
3. if t is $f\underline{(}t_1, t_2, \ldots, t_n\underline{)}$, then $\mathcal{F}(t) = \mathcal{F}(t_1) \cup \mathcal{F}(t_2) \cup \cdots \cup \mathcal{F}(t_n)$.

□

[6] The traditional usage of the name "first-order logic" applies to the logic which contains relations as part of the signature. We follow usage in the algebraic specification literature in calling the logic described in this section "first-order logic", even though signatures have no relations. Traditional first-order logic is called "many-sorted predicate calculus", and is described in section 2.6.

DEFINITION 2.25: *Satisfaction.* Given a Σ-algebra A, a Σ-formula φ with free variables in X, and an assignment $\alpha: X \to A$, the satisfaction of the formula φ by the algebra A under the assignment α, written $A \models_\alpha \varphi$, is defined inductively as follows (here $t_1, t_2 \in T_\Sigma(X)$, φ and ψ are Σ-formulas with free variables in X, and x is a variable of sort s):

1. $A \models_\alpha t_1 = t_2$ iff $\bar{\alpha}(t_1) = \bar{\alpha}(t_2)$;[8]
2. $A \models_\alpha \neg\varphi$ iff it is not the case that $A \models_\alpha \varphi$;
3. $A \models_\alpha \varphi \vee \psi$ iff $A \models_\alpha \varphi$ or $A \models_\alpha \psi$ or both;
4. $A \models_\alpha \varphi \wedge \psi$ iff both $A \models_\alpha \varphi$ and $A \models_\alpha \psi$;
5. $A \models_\alpha \varphi \Rightarrow \psi$ iff either it is not the case that $A \models_\alpha \varphi$, or it is the case that both $A \models_\alpha \varphi$ and $A \models_\alpha \psi$;
6. $A \models_\alpha \forall x \cdot \varphi$ iff for every element v of the carrier A_s, $A \models_{\alpha[x \leftarrow v]} \varphi$;
7. $A \models_\alpha \exists x \cdot \varphi$ iff there is at least one element v of the carrier A_s such that $A \models_{\alpha[x \leftarrow v]} \varphi$.

The notation $\alpha[x \leftarrow v]$ used above denotes an assignment $\alpha[x \leftarrow v]: X \cup \{x\} \to A$ derived from α as follows:

$$\alpha[x \leftarrow v](y) = \begin{cases} v, & \text{if } y = x; \\ \alpha(y), & \text{otherwise.} \end{cases}$$

A Σ-algebra A is said to satisfy a Σ-formula φ (with free variables in X), written $A \models \varphi$, iff for all assignments $\alpha: X \to A$, we have $A \models_\alpha \varphi$. Since a sentence is a closed formula, this definition also applies to the satisfaction of a sentence by an algebra, for which it turns out that satisfaction is independent of the assignment. □

DEFINITION 2.26: *Specification.* A specification is a pair $\langle \Sigma, \Phi \rangle$ of a signature Σ and a set Φ of Σ-sentences. □

As before, an algebra A is said to satisfy a set of sentences Φ, denoted $A \models \Phi$, if $A \models \varphi$ for each sentence $\varphi \in \Phi$.

2.3 Inequations

In this and the next two subsections, we look at some specializations of first-order logic (they may also be regarded as extensions to equational logic) which are commonly used in algebraic specification.

To motivate inequations, consider the specification RING of definition 1.1. The trivial ring consisting of just the element 0, with addition, multiplication, and inverse defined in the only possible way ($0 + 0 = 0$, $0 \times 0 = 0$, $-0 = 0$), forms a model of this specification. If we want to exclude this as a model of RING, we have to include the axiom $0 \neq 1$ in the specification.

[8] The "=" symbol in the formula $t_1 = t_2$ is a syntactic entity; that in $\bar{\alpha}(t_1) = \bar{\alpha}(t_2)$ is semantic equality. Also see the previous footnote.

DEFINITION 2.27: *Inequation.* Given a signature Σ, an inequation is a sentence of the form

$$\{X\}\ l \neq r$$

where $\{X\}$ is a set of (sorted) variables and l and r are terms belonging to $T_\Sigma(X)$. □

An inequation of the form above is equivalent to the first-order logic sentence

$$\forall X \cdot \neg(l = r)$$

and the definition of satisfaction carries through.

Inequations are primarily used to exclude trivial models, and to distinguish elements when adopting final algebra semantics (cf. section 3.2).

2.4 Conditional equations

Another improvement we can make to equations as axioms is to add conditions to them, with the semantics that an equation is true only when its associated conditions are true.

DEFINITION 2.28: *Conditional equation.* A conditional equation is a sentence of the form

$$\{X\}\ c_1 \wedge c_2 \wedge \cdots \wedge c_n \Rightarrow e$$

where $\{X\}$ is a set of (sorted) variables, e is an equation, and each c_i, for $i = 1, \ldots, n$, is either an equation or an inequation. By an equation (or inequation) we mean a formula of the form $l = r$ (respectively, $l \neq r$) where l and r are terms belonging to $T_\Sigma(X)$. □

If all the conditions c_i, for $i = 1, \ldots, n$, are equations, the axiom is said to be a *positive* conditional equation. Again, conditional equations are instances of first-order logic formulas, and the definition of satisfaction carries through.

NOTATION. We sometimes use the alternative syntax

$$\{X\}\ e\ \textbf{if}\ c_1 \wedge c_2 \wedge \cdots \wedge c_n$$

for the conditional equation above.

Expressive power

Conditional equations are strictly more powerful than equations, in the sense that there are data types which can be specified by a finite number of conditional equations, but for which there is no finite axiomatization using just equations. Conditional equations are needed when a function is not "uniformly" defined on the entire domain.

In [Thatcher et al. 82], there is a simple (albeit contrived) example and a rigorous proof to show that positive conditional equations are more expressive than equations. Below, we show a less contrived example which requires (negative) conditional equations. This data type supports associative storage and retrieval.

```
spec ASSOC =
        /* list with associative storage and retrieval */
sorts ASSOC, INDEX, VALUE
operations
       ⊥ :                               → VALUE     /* "undefined" or error value */
   empty :                               → ASSOC
     _[_] :                ASSOC, INDEX → VALUE     /* retrieve value stored at an index */
  _[_←_] : ASSOC, INDEX, VALUE → ASSOC     /* assign a value at an index */
axioms
  empty[i] = ⊥
        /* what is stored can be retrieved */
  a[i←x][i] = x
  a[i←x][j] = a[j] if i ≠ j
        /* later assignments override; independent assignments commute */
  a[i←x][i←y] = a[i←y]
  a[i←x][j←y] = a[j←y][i←x] if i ≠ j
end
```

Disjunctive conditional equations

In the conditional equations we have considered above, the antecedent is a conjunction of conditions. A more general form of these equations would also allow disjunctions of conditions. However, disjunctive conditional equations are subsumed by the conjunctive form. The definition of satisfaction of a set of equations by an algebra treats the set as a conjunction of equations. In other words, if $\mathcal{E} = \{e_1, e_2, \ldots, e_n\}$ is a set of Σ-equations, and A is a Σ-algebra, then the following are equivalent:

$$A \models \mathcal{E},$$
$$A \models e_1 \wedge e_2 \wedge \cdots \wedge e_n.$$

Using this and the properties of the "$\Rightarrow$" connective, we see that a disjunctive conditional equation

$$c_1 \vee c_2 \vee \cdots \vee c_n \Rightarrow e$$

is equivalent to the following set of equations:

$$c_1 \Rightarrow e,$$
$$c_2 \Rightarrow e,$$
$$\ddots$$
$$c_n \Rightarrow e.$$

2.5 Quantifiers

We first observe that equations are in reality universally quantified sentences, i.e., an equation $\langle X, l, r\rangle$ is equivalent to the first-order logic sentence

$$\forall X \cdot l = r.$$

We can also add existential quantifiers to equations or conditional equations. Existential quantifiers are useful for specifying properties without committing to a particular representation or algorithm.

Expressive power

Here is an example specification which is typical of situations where the existential quantifier is particularly useful. The specification describes the predicate "divides" on natural numbers, using the inverse operation, multiplication. Without the existential quantifier, an explicit algorithm for division would be needed.

spec NAT-DIV =
based on BOOL
sorts NAT
operations

$$
\begin{aligned}
0 :& \quad\quad\quad\quad \rightarrow \text{NAT}\\
\text{succ} :& \quad \text{NAT} \rightarrow \text{NAT}\\
_+_ :& \ \text{NAT},\text{NAT} \rightarrow \text{NAT}\\
\times :& \ \text{NAT},\text{NAT} \rightarrow \text{NAT}\\
_\ \text{divides}\ _ :& \ \text{NAT},\text{NAT} \rightarrow \text{BOOL}
\end{aligned}
$$

axioms

/* axioms for addition */

$m + 0 = m$

$m + \text{succ}(n) = \text{succ}(m + n)$

/* axioms for multiplication */

$m \times 0 = 0$

$m \times \text{succ}(n) = m + (m \times n)$

/* division is the inverse of multiplication */

m divides n **if** $\exists p \cdot m \times p = n$

end

Other examples where existential quantifiers are useful are: (1) the definition of the Kleene closure $r*$ of a regular expression r: given a string s, $s \in r* \Leftrightarrow \exists i \cdot s \in r^i$, where i is an integer; (2) the definition of reachability in a graph: a node y is reachable from a node x if there exists a path from x to y; (3) the computation of prime numbers using the sieve of Eratosthenes [Broy et al. 79].

2.6 Predicate calculus

In the previous sections, we considered different kinds of axioms. In this and the next few sections, we look at other kinds of operations. The operations used until now were total functions. Another kind of operation is a relation or predicate, commonly used in logic. First-order logic augmented with predicates is called many-sorted first-order predicate calculus. The definitions of signatures, formulas, satisfaction, etc., are all modified to handle predicates.

```
spec ASSOC =
        /* list with associative storage and retrieval */
sorts ASSOC, INDEX, VALUE
operations
      ⊥ :                           → VALUE     /* "undefined" or error value */
   empty :                          → ASSOC
    _[_] :            ASSOC, INDEX → VALUE     /* retrieve value stored at an index */
  _[_←_] : ASSOC, INDEX, VALUE → ASSOC     /* assign a value at an index */
axioms
  empty[i] = ⊥
        /* what is stored can be retrieved */
  a[i←x][i] = x
  a[i←x][j] = a[j] if i ≠ j
        /* later assignments override; independent assignments commute */
  a[i←x][i←y] = a[i←y]
  a[i←x][j←y] = a[j←y][i←x] if i ≠ j
end
```

Disjunctive conditional equations

In the conditional equations we have considered above, the antecedent is a conjunction of conditions. A more general form of these equations would also allow disjunctions of conditions. However, disjunctive conditional equations are subsumed by the conjunctive form. The definition of satisfaction of a set of equations by an algebra treats the set as a conjunction of equations. In other words, if $\mathcal{E} = \{e_1, e_2, \ldots, e_n\}$ is a set of Σ-equations, and A is a Σ-algebra, then the following are equivalent:

$$A \models \mathcal{E},$$
$$A \models e_1 \wedge e_2 \wedge \cdots \wedge e_n.$$

Using this and the properties of the "$\Rightarrow$" connective, we see that a disjunctive conditional equation

$$c_1 \vee c_2 \vee \cdots \vee c_n \Rightarrow e$$

is equivalent to the following set of equations:

$$c_1 \Rightarrow e,$$
$$c_2 \Rightarrow e,$$
$$\ddots$$
$$c_n \Rightarrow e.$$

2.5 Quantifiers

We first observe that equations are in reality universally quantified sentences, i.e., an equation $\langle X, l, r\rangle$ is equivalent to the first-order logic sentence

$$\forall X \cdot l = r.$$

DEFINITION 2.37: *Reduct.* Given a signature morphism $\sigma: \Sigma \rightarrow \Sigma'$, and a Σ'-structure A', the σ-reduct of A', denoted by $A'|_\sigma$, is the Σ-structure $A = \langle A_S, F_A, R_A \rangle$ defined as follows (with $\Sigma = \langle S, F, R \rangle$):

$$A_s = A'_{\sigma(s)}, \text{ for } s \in S; \quad f_A = (\sigma(f))_{A'}, \text{ for } f \in F; \quad \text{and } r_A = (\sigma(r))_{A'}, \text{ for } r \in R.$$

Given a Σ'-homomorphism $h': A' \rightarrow B'$ between two Σ'-structures A' and B', the σ-reduct of h' is a Σ-homomorphism $h: A'|_\sigma \rightarrow B'|_\sigma$, denoted by $h'|_\sigma$, and defined by the family of functions $h_s = h'_{\sigma(s)}$, for $s \in S$. □

Horn clauses

A restriction of many-sorted first-order predicate calculus, called Horn clause logic is popular because of its efficient rewriting techniques [Padawitz 88; Makowsky 87].

A Horn clause is a formula of the form

$$\gamma \Leftarrow \gamma_1, \gamma_2, \ldots, \gamma_n$$

where γ, γ_i, for $i = 1, \ldots, n$, are all atomic formulas. When $n = 0$, the clause is simply written as γ and is called a fact. The free variables in a Horn clause are implicitly universally quantified (over the entire formula). When there are no predicates involved, it is easy to see that Horn clauses correspond to the conditional equations of section 2.4.

2.7 Hidden functions

Sometimes, to define an algebra unambiguously, we have to introduce auxiliary operations which are not present in the signature of the algebra. Such operations are called *hidden* functions, because they are not present in the algebra. The simplest example where they are necessary is that of defining multiplication on natural numbers. We cannot do this without first defining addition (if only equations are allowed as axioms). Here is another example, a deterministic finite state automaton. The hidden function which computes the extension of the transition function to strings is necessary to define the language accepted by the automaton.

spec DFA =
based on
BOOL,
STRING-OF-Σ /* the string data type; see example 4.38 */
sorts
Q, /* the set of states */
Σ /* the input alphabet */
operations

q_0 :	$\rightarrow Q$	/* the initial state */
is-final :	$Q \rightarrow$ BOOL	/* predicate for final states */
δ :	$Q, \Sigma \rightarrow Q$	/* transition function */
$\hat{\delta}$:	Q, STRING-OF-$\Sigma \rightarrow Q$ **[hidden]**	/* extension of δ to strings */
accept :	STRING-OF-$\Sigma \rightarrow$ BOOL	/* the language accepted */

axioms
accept(x) = is-final($\hat{\delta}(q_0, x)$)

$\hat{\delta}(q, \epsilon) = q$
$\hat{\delta}(q, a\omega) = \hat{\delta}(\delta(q, a), \omega)$
end

It has been proved that equational specifications with hidden functions are strictly more powerful than those without (see [Thatcher et al. 82]). It is also the case that hidden functions are required to describe all computable algebras [Bergstra and Tucker 83].

We will not redefine signatures to include hidden functions; the same effect can be obtained (uniformly, for any kind of signature) by specification-building operations which hide some parts of a specification, cf. definition 4.14.

2.8 Partial first-order logic

In many algebraic structures, functions are not defined over the entire domain, e.g., accessing an empty array or list, or dividing a number by zero. In the axiom systems we have defined until now, we have assumed total functions. As a result, algebraic structures such as fields cannot be defined.[10] How do we accommodate such structures into the framework of algebraic specification?

One approach would be to make every function total by adding error elements to domains. Unfortunately, adding error elements, and naively extending the definitions of functions leads to inconsistent specifications. Goguen et al. [Goguen et al. 78] convincingly demonstrate that this can lead to very complicated specifications.

An alternative is to forgo the convenience of total functions and use partial functions and partial algebras. We now define partial first-order logic, a logic which is similar to first-order logic (section 2.2) except that the functions used are partial. We consider another approach, order-sorted logic, in the next section.

Signatures and terms for partial first-order logic are defined as for (total) first-order logic. The key feature of partial first-order logic is the presence of a *definedness predicate* D_s for each sort s.

[10] Using just equations as axioms.

DEFINITION 2.38: *Formulas.* We add the following rule for building an atomic formula, to the rules already given for (total) first-order logic (definition 2.24):

If t is a term of sort s, then $D_s(t)$ is an atomic formula.

□

NOTATION. Usually the sort subscript in D_s will be omitted, as it can be inferred from the context.

DEFINITION 2.39: *Partial algebra.* Given a signature $\Sigma = \langle S, \Omega \rangle$, a partial Σ-algebra $A = \langle A_S, F_A \rangle$ consists of two families:

1. a collection of sets, called the *carriers* of the algebra, $A_S = \{ A_s \mid s \in S \}$; and
2. a collection of *partial* functions, $F_A = \{ f_A \mid f \in \Omega \}$, such that, if the rank of f is $s_1, s_2, \ldots, s_n \rightarrow s$, then f_A is a partial function from $A_{s_1} \times A_{s_2} \times \cdots \times A_{s_n}$ to A_s.

□

DEFINITION 2.40: *Homomorphism.* Given two partial Σ-algebras A and B, a total[11] Σ-homomorphism $h: A \rightarrow B$ is a family of total functions $\{ h_s: A_s \rightarrow B_s \mid s \in S \}$ between the carriers which are compatible with the operations, i.e., for all operation symbols $f: s_1, s_2, \ldots, s_n \rightarrow s$, and for all $a_1 \in A_{s_1}, a_2 \in A_{s_2}, \ldots, a_n \in A_{s_n}$,

1. $f_A(a_1, a_2, \ldots, a_n)$ is defined $\Rightarrow f_B(h_{s_1}(a_1), h_{s_2}(a_2), \ldots, h_{s_n}(a_n))$ is defined, and
2. $h_s(f_A(a_1, a_2, \ldots, a_n)) = f_B(h_{s_1}(a_1), h_{s_2}(a_2), \ldots, h_{s_n}(a_n))$.

□

DEFINITION 2.41: *Evaluation of terms.* Given a partial Σ-algebra A, and a set X of sorted variables, a total assignment $\alpha: X \rightarrow A$ can be extended to a partial assignment for terms $\bar{\alpha}: T_\Sigma(X) \rightarrow A$ as follows:

1. if x is a variable, then $\bar{\alpha}(x) = \alpha(x)$;
2. if c is a constant symbol, then $\bar{\alpha}(c) = c_A$;
3. if f is an n-ary operation symbol, then $\bar{\alpha}(f(t_1, t_2, \ldots, t_n)) = f_A(\bar{\alpha}(t_1), \bar{\alpha}(t_2), \ldots, \bar{\alpha}(t_n))$, provided each of $\bar{\alpha}(t_1), \bar{\alpha}(t_2), \ldots, \bar{\alpha}(t_n)$ is defined, and f_A is defined; otherwise $\bar{\alpha}$ is undefined for this term.

□

[11]There are several variations of this definition. See, for example, [Broy and Wirsing 82; Wirsing and Broy 82].

DEFINITION 2.42: *Satisfaction.* The satisfaction of a Σ-formula φ (with free variables in X) by a partial Σ-algebra A under the assignment $\alpha: X \to A$ is defined as for first-order logic (definition 2.25), except for the following rules for atomic formulas:

1. $A \models_\alpha D(t)$ iff $\bar{\alpha}(t)$ is defined;
2. $A \models_\alpha t_1 = t_2$ iff $\bar{\alpha}(t_1)$ and $\bar{\alpha}(t_2)$ are both undefined, or they are both defined and equal.[12]

□

Note that for quantifiers, since the quantification ranges over elements of the carrier, only defined terms may be substituted for the quantified variables. As usual, a partial algebra satisfies a sentence when it satisfies it for all assignments.

EXAMPLE 2.43. We give below a specification for queues in which some operations are partial. Compare this with a specification using order-sorted logic, example 2.53.

```
spec QUEUE =
sorts QUEUE, DATA
operations      /* partial functions */
  emptyq :                    → QUEUE
     putq : DATA, QUEUE → QUEUE
     getq :           QUEUE → DATA
     popq :           QUEUE → QUEUE
axioms
  ¬D(popq(emptyq))
  ¬D(getq(emptyq))

  getq(putq(x, q)) = if q = emptyq then x else getq(q)
  popq(putq(x, q)) = if q = emptyq then q else popq(q)
end
```

□

Equational Partiality

It is frequently the case that partial functions are defined on subsets of their domains which can be equationally characterized. Such equations, called domain conditions, can be attached to the signature as shown below [Reichel 87]. Equational partiality is a restricted form of partiality, and results in a principle of induction for equationally partial algebras.

It is evident that domain conditions are equivalent to using conditional equations in the axioms section of a specification. The difference is methodological: domain conditions clearly separate axioms characterizing the domain of definition from axioms characterizing the rest of the behaviour of a function.

[12] This interpretation of "=" is called strong equality. For variations, see [Broy and Wirsing 82; Wirsing and Broy 82].

EXAMPLE 2.44. Here is an example of a bounded-queue which illustrates the use of conditions in the signature to indicate the domain of definition of partial operations.

```
spec BOUNDED-QUEUE =
based on NAT
sorts QUEUE, DATA
operations
  emptyq :                                               NAT → QUEUE
    putq : DATA, (q: QUEUE iff length(q) < bound(q)) → QUEUE
    getq :             (q: QUEUE iff length(q) > 0) → DATA
    popq :             (q: QUEUE iff length(q) > 0) → QUEUE
  length :                                            QUEUE → NAT
   bound :                                            QUEUE → NAT
axioms
  getq(putq(x, q)) = if q = emptyq(bound(q)) then x else getq(q)
  popq(putq(x, q)) = if q = emptyq(bound(q)) then q else popq(q)

  length(emptyq(n)) = 0
  length(putq(x, q)) = length(q) + 1

  bound(emptyq(n)) = n
  bound(putq(x, q)) = bound(q)
end
```

□

2.9 Order-sorted logic

Another approach to handling partiality (and errors) is to define a partial function as a total function on a subset of the domain. Order-sorted logic is a formalism based upon this philosophy. We describe order-sorted logic below by concentrating on the differences between order-sorted logic and equational logic (cf. section 2.1). We refer to the two logics by using the prefixes "order-sorted" and "many-sorted", respectively.

DEFINITION 2.45: *Signature.* An order-sorted signature Σ is a triple $\langle S, \leq, \Omega\rangle$ where $\langle S, \Omega\rangle$ is a many-sorted signature (definition 2.1) and $\langle S, \leq\rangle$ is a partial order on the set of sorts S.

We denote by $\Sigma_{\omega,s}$ the set of operation symbols in Ω with arity ω and sort s. The ordering on S is extended to strings of equal length in S^* by $s_1 \dots s_n \leq s'_1 \dots s'_n$ iff $s_i \leq s'_i$ for $i = 1, \dots, n$. The ordering on pairs $\langle\omega, s\rangle \in S^* \times S$ is defined by $\langle\omega, s\rangle \leq \langle\omega', s'\rangle$ iff $\omega \leq \omega'$ and $s \leq s'$.

The operations in Ω are required to satisfy the *monotonicity condition*:

$$\text{if } f \in (\Sigma_{\omega_1,s_1} \cap \Sigma_{\omega_2,s_2}) \text{ and } \omega_1 \leq \omega_2, \text{ then } s_1 \leq s_2.$$

□

If $s_1 \leq s_2$ in the partial order on the sorts, then s_1 is said to be *subsort* of s_2. Order-sorted signatures allow overloading of function names. Overloading of function names on subsorts is called *subsort* polymorphism, e.g., addition (+) on natural numbers, integers, and rationals. Overloading of function names on unrelated sorts is called *ad hoc* polymorphism, e.g., addition (+) on integers and logical-or (+) on booleans. Both kinds of overloading can be used if an order-sorted signature is *regular*.

DEFINITION 2.46: *Regular signature.* An order-sorted signature Σ is said to be regular iff given $\omega_0 \leq \omega_1$ in S^* and $f \in \Sigma_{\omega_1,s_1}$, there is a least rank $\langle \omega, s\rangle \in S^* \times S$ with $\omega_0 \leq \omega$ such that $f \in \Sigma_{\omega,s}$. □

A stronger condition than regularity, *coherence*, is required on signatures to ensure that satisfaction is closed under isomorphism. Coherence means that for any two sorts which are "related", there is a common supersort.

A connected component in a partially ordered set is a subset all of whose elements are comparable; in other words, connected components are equivalence classes generated by the symmetric, transitive closure of the ordering relation $\leq$.

DEFINITION 2.47: *Filtered.* A partially ordered set $\langle S, \leq\rangle$ is filtered if every pair of elements $s, t \in S$ has an upperbound, i.e., an element $u \in S$ such that $s \leq u$ and $t \leq u$. $\langle S, \leq\rangle$ is said to be *locally* filtered if each of its connected components is filtered. □

DEFINITION 2.48: *Coherent signature.* An order-sorted signature is coherent if it is regular and locally filtered. □

DEFINITION 2.49: *Algebra.* Given an order-sorted signature $\Sigma = \langle S, \leq, \Omega\rangle$, an order-sorted Σ-algebra A is a many-sorted Σ-algebra (definition 2.8) which respects the subsort relation, i.e.,

1. if $s \leq s'$ in S, then $A_s \subseteq A_{s'}$;
2. if $f \in (\Sigma_{\omega_1,s_1} \cap \Sigma_{\omega_2,s_2})$ with $\langle \omega_1, s_1\rangle \leq \langle \omega_2, s_2\rangle$, then $f_A\colon A_{\omega_2} \to A_{s_2}$ equals $f_A\colon A_{\omega_1} \to A_{s_1}$ when restricted to the subset A_{ω_1}. (If $\omega = s_1 \ldots s_n$, we write A_ω for $A_{s_1} \times \cdots \times A_{s_n}$.)

□

Condition (1) requires that the subsort relation be represented as a subset relation between carrier sets. Condition (2) requires that overloaded operators on subsorts agree as functions on the corresponding subsets.

DEFINITION 2.50: *Homomorphism.* Given two order-sorted Σ-algebras A and B, an order-sorted homomorphism $h\colon A \to B$ is a many-sorted homomorphism (definition 2.14) which respects the subsort relation, i.e.,

$$\text{if } s \leq s' \text{ and } a \in A_s, \text{ then } h_s(a) = h_{s'}(a).$$

□

DEFINITION 2.51: *Terms.* Given a set of sorted variables X and an order-sorted signature $\Sigma = \langle S, \leq, \Omega \rangle$, the set of terms $T_\Sigma(X)$ is defined as for many-sorted algebra (definition 2.3), augmented by the following rule:

$$\text{if } s \leq s' \text{ in } S \text{ and } t \in T_{\Sigma,s}(X), \text{ then } t \in T_{\Sigma,s'}(X), \text{ i.e.,} T_{\Sigma,s}(X) \subseteq T_{\Sigma,s'}(X).$$

□

This extra rule has the effect that the same term can belong to more than one sort. However, regularity of signatures implies that we can define the *least sort* of a term $t \in T_\Sigma(X)$, denoted by $\text{LS}(t)$.

In the case of many-sorted equational logic, we defined equations to be pairs of terms of the same sort. For order-sorted logic, this is too restrictive. It would be meaningful (and necessary) to allow one of the sorts to be a subsort of the other. In general, the two sorts should be comparable in the partial order given on the sorts.

DEFINITION 2.52: *Equations and satisfaction.* Given a coherent order-sorted signature $\Sigma = \langle S, \leq, \Omega \rangle$, an order-sorted Σ-equation is a triple $\langle X, l, r \rangle$, where X is a sorted set of variables, and l and r are terms in $T_\Sigma(X)$ with their least sorts, $\text{LS}(l)$ and $\text{LS}(r)$, in the same connected component of $\langle S, \leq \rangle$.

Assignments of values to variables and the extension of such assignments to terms is defined as for many-sorted equational logic (definition 2.9). For any Σ-equation $\langle X, l, r \rangle$, the assumption of coherence gives a common supersort s for the two sides l and r of the equation. An order-sorted Σ-algebra A satisfies the equation $\langle X, l, r \rangle$ iff for all assignments $\alpha: X \to A$, $\bar{\alpha}_s(l) = \bar{\alpha}_s(r)$. □

EXAMPLE 2.53. We will show how order-sorted logic elegantly solves the problem of partially defined functions. In the example below, the sort for queues has a subsort for non-empty queues. Compare with example 2.43 which uses partial functions.

spec QUEUE =
sorts DATA, NON-EMPTY-QUEUE, QUEUE
subsorts NON-EMPTY-QUEUE ≤ QUEUE
operations
emptyq : → QUEUE
putq : DATA, QUEUE → NON-EMPTY-QUEUE
getq : NON-EMPTY-QUEUE → DATA
popq : NON-EMPTY-QUEUE → QUEUE
axioms
getq(putq(x, q)) = **if** q = emptyq **then** x **else** getq(q)
popq(putq(x, q)) = **if** q = emptyq **then** q **else** popq(q)
end

□

One of the interesting features of order-sorted logic is retracts. Let s be a subsort of t. A *coercion* from s to t is a function which converts the sort of any term x from s to t. A *retract* is a function which converts the sort of a term x from t to s (provided x satisfies the

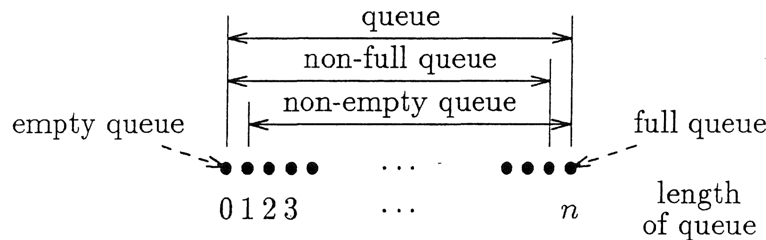
EXAMPLE 2.54. Here is an example which requires sort constraints. We use a slightly different syntax from that in [Goguen and Winkler 88]. Observe that the subsort NON-FULL-QUEUE cannot be specified using generators.

spec BOUNDED-QUEUE =
based on NAT
sorts DATA, QUEUE, NON-EMPTY-QUEUE, NON-FULL-QUEUE
subsorts NON-EMPTY-QUEUE $\leq$ QUEUE, NON-FULL-QUEUE $\leq$ QUEUE
operations

emptyq :	NAT $\rightarrow$ QUEUE	
emptyq :	NAT $\rightarrow$ NON-FULL-QUEUE	**if** length(q) $<$ bound(q)
putq :	DATA, NON-FULL-QUEUE $\rightarrow$ NON-EMPTY-QUEUE	
putq :	DATA, NON-FULL-QUEUE $\rightarrow$ NON-FULL-QUEUE	**if** length(q) $<$ bound(q)
getq :	NON-EMPTY-QUEUE $\rightarrow$ DATA	
popq :	NON-EMPTY-QUEUE $\rightarrow$ NON-FULL-QUEUE	
length :	QUEUE $\rightarrow$ NAT	
bound :	QUEUE $\rightarrow$ NAT	

axioms
/* axioms omitted */
end

The relationship between the various sorts is shown in the picture below (for a queue with bound n).



□

2.10 Other logics

Another system of axioms considered in the literature is higher-order logic, in which some sorts can be function spaces, i.e., functions become first-class objects [Parsaye-Ghomi 82; Möller 87; Möller 86; Broy 87; Maibaum and Lucena 80; Poigné 86]. This incorporates into algebraic specification the rich variety of programming clichés which are used in functional programming. In addition to other kinds of axioms, there are other kinds of algebras, e.g., algebras in which the carriers are (Scott-) domains and the operations are continuous functions. Such algebras allow the description of infinite objects such as streams and lazy lists [Goguen et al. 77; Levy and Maibaum 82; Möller 85; Tarlecki and Wirsing 86; Möller et al. 88].

2.11 Concluding remarks on logics

We have seen a variety of logics which can serve as a substrate for writing specifications. Are there criteria for choosing between these logics? An obvious criterion is expressive power. It has been shown [Bergstra and Tucker 83; Bergstra et al. 81; Meseguer and Goguen 84] that equational logic with hidden functions (and initial semantics, section 3.1) is sufficient for describing all computable algebras, i.e., algebras whose carriers are recursive sets and whose functions are recursive. If we adopt computability as a measure of expressiveness, then it would seem that we do not need logics more powerful than equational logic.

However, for the specification of software, we need another kind of expressiveness. A specification should *abstractly* characterize a type, program, function, or computation. "Abstract" here can mean independent of the representation of the entity being described. "Abstract" can also mean providing only the *essential* properties of an entity. Although equational logic is sufficient to describe all computable algebras, equations sometimes force the choice of a particular representation or algorithm. Here is where more powerful logics come to the rescue. Consider the specification of the notion of "prime number". Specifying this with equations requires an algorithm to determine primeness (see, for example, [Goguen and Winkler 88, page 52]). Using quantifiers allows a more direct and abstract expression of primeness: a prime is a natural number with no non-trivial factors.

$$\text{prime}(n) \;\textbf{if}\; \forall m \in \text{NAT} \cdot m = 1 \vee m = n \vee \neg(m \text{ divides } n),$$

$$\text{where, } m \text{ divides } n \;\textbf{if}\; \exists p \in \text{NAT} \cdot n = m \times p.$$

We believe that no one logic will be expressive enough to describe all problems of practical interest; thus, it is necessary to investigate several logics, and even consider writing a specification in several different logics at once (see section 4.1.1). Also, apart from the technical power of a logic, the understandability of a specification is affected by the logic in which it is written.

Another point to note is that most of the logics we have described in this section are variations of traditional logic and model theory. An exception is order-sorted logic, which was motivated by the need to handle partial functions and errors in typical data structures used in computer science. Order-sorted logic is a non-trivial departure from traditional universal algebra and model theory. Something like the machinery of order-sorted logic is required to adequately describe even the quintessential toy example of computer science, stacks [Goguen and Meseguer 87a].

3 SEMANTICS

Abstracting from the logics considered in the last section, we see that a specification is a signature together with a set of axioms. Associated with each signature is a collection of algebras, or *models*. The axioms in a specification serve to pick out some of these models via the satisfaction relation between axioms and models. This sub-collection of models can be considered to be the denotation or semantics of a specification. Since the primary purpose of a specification is to characterize a class of algebras, we can choose to impose further (implicit) constraints on the collection of models associated with a specification. There are several techniques to impose constraints: we may require the models to satisfy the axioms in some "canonical" fashion (initial and final semantics), we may require the models to satisfy the "behaviour" specified by the axioms (behavioural semantics), we may require certain parts of a specification to interpreted specially (hierarchy and data constraints). Each of these techniques is an attempt to describe certain classes of algebras as simply as possible.

There are four major schools of thought on which models should be regarded as the semantics of a specification:

- initial semantics
- final semantics
- loose semantics
- behavioural semantics

The semantics of a specification is defined in terms of the category of models associated with the specification. For a specification $\langle\Sigma, \Phi\rangle$, this category is formed with all the Σ-algebras as objects and all the Σ-homomorphisms as morphisms. Loose semantics regards the entire category as the semantics. Initial semantics picks out the initial objects in the category of models. An initial object in a category is an object from which there is exactly one morphism to every object. Final semantics picks out the final objects in a specified sub-category of the model category. A final object in a category is an object to which there is exactly one morphism from every object. Behavioural semantics allows models (with the same signature) which have the same observable properties (or behaviour) as some model in the model category. The different kinds of semantics are depicted in figure 2.

There are also combinations of semantics: initial behaviour semantics, i.e., the initial model in the category of models and behaviour preserving morphisms (see example 4.3); and stratified loose semantics, a generalization of initial and loose semantics (see definition 4.44).

We will look at each kind of semantics in detail in the following sections. We will describe properties of the concerned models, and investigate the effect of various kinds of axioms on the existence of particular kinds of models. At the end of this section, we will also describe a kind of operational semantics for specifications: rewriting using the axioms. Other kinds of constraints on models will be discussed in section 4.5.

3.1 Initial semantics

We first consider an example to introduce the concept of initial model. The specification is that of strings generated from an alphabet consisting of the letters "a" and "b".

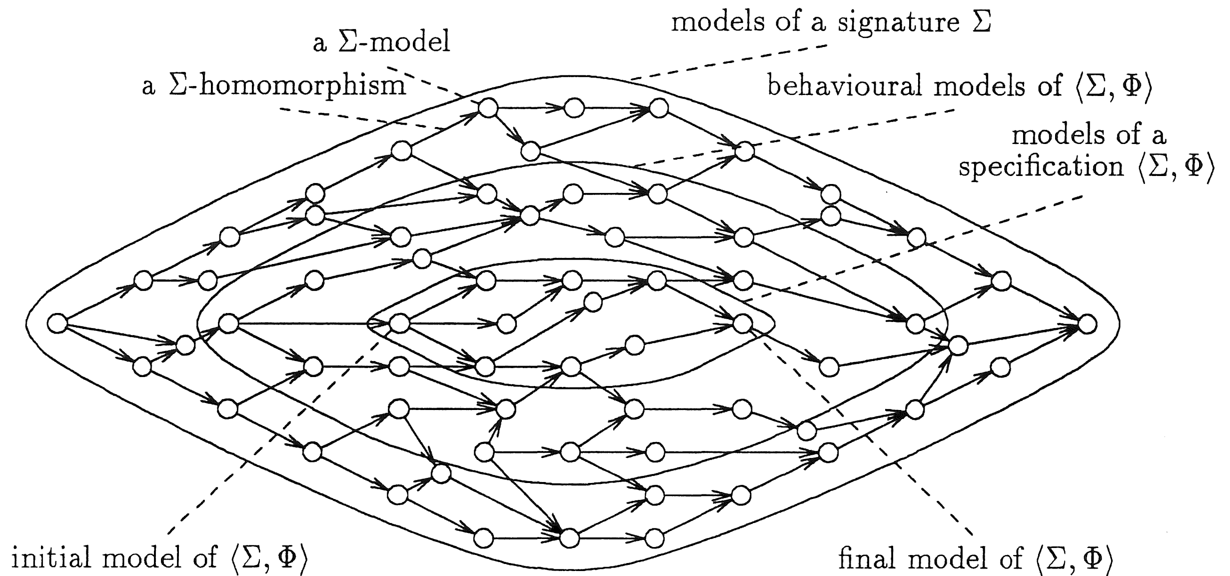


Figure 2: Models of a specification

spec STRING =
sorts STRING
operations

a :	→ ALPHABET	/* constant letters */
b :	→ ALPHABET	
' ' :	→ STRING	/* the empty string */
'_' :	ALPHABET → STRING	/* convert letter to string */
_ · _ :	STRING, STRING → STRING	/* concatenate */

axioms

$x \cdot (y \cdot z) = (x \cdot y) \cdot z$ /* concatenation is associative */
$x \cdot$ ' ' $= x$ /* ' ' is an identity */
' ' $\cdot\, x = x$

end

Consider a model of this specification with the carrier for the sort STRING being the set $\{\omega_1, \omega_2, \omega_3, \ldots\}$. Given two strings ω_i and ω_j, the concatenation operation produces another string $\omega_k = \omega_i \cdot \omega_j$. The term "$\omega_i \cdot \omega_j$" can be considered to be a "name" of the string ω_k. Of course, a string can have many names, because it can be constructed in many ways. Setting this aside for the moment, the set of all such names looks like a potential carrier set. Is it possible to define appropriate operations on it to make it into a model? The answer is "yes", as shown by the construction below.

We describe the construction of the term algebra for the equational institution. Analogous constructions are possible for other institutions. The set of ground terms generated from a signature Σ is denoted, as usual, by T_Σ (see definition 2.3).

DEFINITION 3.1: *Ground term algebra.* The ground term algebra $\mathcal{T}_\Sigma$ generated by a signature $\Sigma = \langle S, \Omega \rangle$ consists of

1. the family of carrier sets $\{ T_{\Sigma,s} \mid s \in S \}$, and
2. the family of functions $\{ f_T : T_{\Sigma,s_1} \times T_{\Sigma,s_2} \times \cdots \times T_{\Sigma,s_n} \rightarrow T_{\Sigma,s} \mid f \in \Omega$ with rank $s_1 \times s_2 \times \cdots \times s_n \rightarrow s \}$ defined by $f_T(t_1, t_2, \ldots, t_n) = f\underline{(}t_1, t_2, \ldots, t_n\underline{)}$.

□

Observe the use of the syntactic symbols $\underline{(}\ldots\underline{)}$. The effect of applying a function to a set of terms is to generate the syntactic string corresponding to the application.

The ground term algebra is an algebra generated just from the signature. Now we account for the equations in a specification. An equation $l = r$ says that l and r can be considered to be two different ways of producing the same value, i.e., two "names" for the same value. We can achieve this in our term algebra by coalescing the terms representing the two names.

DEFINITION 3.2: *Congruence generated by equations.* Let $\Sigma = \langle S, \Omega \rangle$ be a signature, $\langle \Sigma, \mathcal{E} \rangle$ be an equational specification, $l, r, q \in T_{\Sigma,s}$ be terms of sort s, $f: s_1, \ldots, s', \ldots, s_n \rightarrow s$ be a function symbol, and $\tau_1, \tau_2 \in T_{\Sigma,s'}$ be terms of sort s'. The congruence $\equiv_\mathcal{E}$ on terms generated by a set of equations $\mathcal{E}$ is a family of equivalence relations $\{ \equiv_{\mathcal{E},s} \mid s \in S \}$ defined by

1. if $l = r$ is an equation in $\mathcal{E}$, then $l \equiv_{\mathcal{E},s} r$;
2. (reflexive) $l \equiv_{\mathcal{E},s} l$;
3. (symmetric) $l \equiv_{\mathcal{E},s} r \Rightarrow r \equiv_{\mathcal{E},s} l$;
4. (transitive) $l \equiv_{\mathcal{E},s} r \wedge r \equiv_{\mathcal{E},s} q \Rightarrow l \equiv_{\mathcal{E},s} q$;
5. if $\tau_1 \equiv_{\mathcal{E},s'} \tau_2$, then $f_T(t_1, \ldots, \tau_1, \ldots, t_n) \equiv_{\mathcal{E},s} f_T(t_1, \ldots, \tau_2, \ldots, t_n)$.

□

Thus a congruence is a family of equivalence relations on the carrier sets such that the equivalence relations are compatible with the functions in the algebra. Imposing this congruence on the ground term algebra produces an algebra which respects the equations.

DEFINITION 3.3: *Quotient term algebra.* The quotient term algebra $\mathcal{T}_\Sigma$ generated by a specification $\langle \Sigma, \mathcal{E} \rangle$ (with $\Sigma = \langle S, \Omega \rangle$) consists of

1. the carrier sets $\{ \mathcal{T}_{\Sigma,s} / \equiv_{\mathcal{E},s} \mid s \in S \}$; and
2. the functions $\{ f_T \mid f \in \Omega \}$ defined by $f_T([t_1], [t_2], \ldots, [t_n]) = [f_T(t_1, t_2, \ldots, t_n)]$,

where the notation $X/\equiv$ indicates the quotient of the set X by the equivalence relation $\equiv$, and $[x]$ indicates the equivalence class of the element x. □

The quotient term algebra is an initial algebra in the category of models of an equational specification. A proof that this algebra is initial in the category of models can be found

in [Ehrig and Mahr 85] or [Goguen et al. 78]. The unique homomorphism from $\mathcal{T}_\Sigma$ to any algebra A required by the definition of initiality is the "evaluation" map: it maps each term in the term algebra to the value obtained by evaluating it (see definition 2.9) in the algebra A.

DEFINITION 3.4: *Initial semantics.* Given a specification $\langle\Sigma, \Phi\rangle$, its initial semantics is the collection of all initial objects in the category of Σ-algebras satisfying the axioms Φ and Σ-homomorphisms. □

Remarks

Initial semantics is characterized by the following two slogans, invented by Burstall and Goguen [Burstall and Goguen 82]:

- **no junk:** every element in the carriers of an initial algebra can be represented by some term;
- **no confusion:** two terms denote the same element iff they can be proved equal via the equations.

The "no junk" condition says that the initial algebra is minimal in the sense that only those elements which are required by the specification (those which have names) are present. For example, if we take the initial algebra for the specification STRING above and add an extra element "c" in the carrier for ALPHABET, we still have a STRING-algebra (after modifying the interpretation of the function '_' so that 'c' $\mapsto$ ' '). However, this algebra is not initial because there is no term which corresponds to the element "c". The "no junk" condition provides a second-order structural induction principle for the initial algebra (see definition 3.14, and also [Meseguer and Goguen 84]).

The "no confusion" condition says that we cannot have two different "names" for the same element in a carrier (unless required by the equations). For example, if we use a one element carrier $\{0\}$ for the sort ALPHABET, then both "a" and "b" will map onto the element 0. Although we can obtain an algebra using this carrier, it is not an initial algebra. Again, the "no confusion" condition provides a second-order proof rule which states that two elements of a carrier are different if they are not provably equal via the axioms.

Initiality is quite a strong restriction on the class of models of a specification. Consider the following specification for natural numbers.

spec NAT =
sorts NAT
operations
  $0 : \quad\quad \to \text{NAT}$
  $\text{succ} : \text{NAT} \to \text{NAT}$
end

The initial model of this specification is the standard model of natural numbers. Since there are no equations, no two terms denote the same element. Thus, we have an infinite number of distinct elements $\{\,0, \text{succ}(0), \text{succ}(\text{succ}(0)), \text{succ}(\text{succ}(\text{succ}(0))), \ldots\}$ in the carrier

for NAT. The "no junk" condition ensures that these are the only elements in the carrier. On the other hand, loose semantics allows all models, e.g., NAT mod n for any n, and models including arbitrary junk in the carrier for NAT. An appreciation of the power of initiality can be gained by considering that the standard model of natural numbers needs the Peano induction rule (a second-order axiom) to uniquely characterize it.

Existence of initial models

The most general axiom system for which initial models always exist is Horn clause logic with equality [Mahr and Makowsky 84]. For conditions under which initial models exist when partial functions and more general axioms are allowed, see section 3.3.1. Meseguer [Meseguer 89] shows that any logic which is a categorical logic,[13] initial models exist. Categorical logics include higher-order equational logic [Parsaye-Ghomi 82], higher-order intuitionistic logic [Lambek and Scott 86], and polymorphic lambda-calculus [Meseguer 88].

EXAMPLE 3.5: *Boolean functions.* As an example of initial semantics, and for future reference, we define a two-valued Boolean algebra (initial semantics forces the model to be two-valued).

spec BOOL =
sorts BOOL
operations

$$\begin{array}{rr} \mathrm{T}: & \to \mathrm{BOOL} \\ \mathrm{F}: & \to \mathrm{BOOL} \\ _\vee_ : \mathrm{BOOL}, \mathrm{BOOL} & \to \mathrm{BOOL} \\ _\wedge_ : \mathrm{BOOL}, \mathrm{BOOL} & \to \mathrm{BOOL} \\ \neg_ : \quad \mathrm{BOOL} & \to \mathrm{BOOL} \end{array}$$

axioms

$$\begin{array}{ll} \neg\mathrm{T} = \mathrm{F} & \\ \neg\mathrm{F} = \mathrm{T} & \\ b \vee b' = b' \vee b & b \wedge b' = b' \wedge b \\ \mathrm{T} \vee b = \mathrm{T} & \mathrm{T} \wedge b = b \\ \mathrm{F} \vee b = b & \mathrm{F} \wedge b = \mathrm{F} \end{array}$$

end

□

3.2 Final semantics

Sometimes initial models contain too much information; they are not abstract enough. Consider the STRING specification again, this time with an added operation which determines whether a given letter occurs in a string. To reinforce that the intended model is different, we show alongside a renamed, but isomorphic, specification.

[13] This phrase has a technical meaning: a theory is a category with some properties P, and models are P-preserving functors into some model category (see [Kock and Reyes 77]).

spec STRING =
based on BOOL
sorts STRING
operations

$$
\begin{array}{rrl}
\mathrm{a}: & & \rightarrow \mathrm{ALPHABET} \\
\mathrm{b}: & & \rightarrow \mathrm{ALPHABET} \\
\text{`'}: & & \rightarrow \mathrm{STRING} \\
\text{`_'}: & \mathrm{ALPHABET} & \rightarrow \mathrm{STRING} \\
\cdot: & \mathrm{STRING}, \mathrm{STRING} & \rightarrow \mathrm{STRING} \\
_\in_: & \mathrm{ALPHABET}, \mathrm{STRING} & \rightarrow \mathrm{BOOL}
\end{array}
$$

axioms

$x \cdot (y \cdot z) = (x \cdot y) \cdot z$
$x \cdot \text{`'} = x$
$\text{`'} \cdot x = x$
$\alpha \in \text{`'} = \mathrm{F}$
$\alpha \in \text{`}\beta\text{'} =$ **if** $\alpha = \beta$ **then** T **else** F
$\alpha \in (x \cdot y) = \alpha \in x \vee \alpha \in y$

end

spec SET =
based on BOOL
sorts SET
operations

$$
\begin{array}{rrl}
\mathrm{a}: & & \rightarrow \mathrm{ALPHABET} \\
\mathrm{b}: & & \rightarrow \mathrm{ALPHABET} \\
\{\,\}: & & \rightarrow \mathrm{SET} \\
\{_\}: & \mathrm{ALPHABET} & \rightarrow \mathrm{SET} \\
\cup: & \mathrm{SET}, \mathrm{SET} & \rightarrow \mathrm{SET} \\
_\in_: & \mathrm{ALPHABET}, \mathrm{SET} & \rightarrow \mathrm{BOOL}
\end{array}
$$

axioms

$x \cup (y \cup z) = (x \cup y) \cup z$
$x \cup \{\,\} = x$
$\{\,\} \cup x = x$
$\alpha \in \{\,\} = \mathrm{F}$
$\alpha \in \{\beta\} =$ **if** $\alpha = \beta$ **then** T **else** F
$\alpha \in (x \cup y) = \alpha \in x \vee \alpha \in y$

end

From the perspective of the operator "$\in$", the carrier SET can have only four distinct elements: $\{\,\}$, $\{\mathrm{a}\}$, $\{\mathrm{b}\}$, and $\{\mathrm{a}, \mathrm{b}\}$. We see that the initial algebra contains an infinite number of distinct elements for this carrier. The initial algebra contains too much information because each element of a carrier is essentially a record of how it was constructed.

The motivation for abstract data types in computer science is to abstractly characterize a data type without worrying about the specific details of its implementation—also known as representation independence, encapsulation, or information hiding. The abstractness is achieved by specifying certain operations for manipulating the data structure and describing these operations by axioms. Properties which cannot be determined using these operations are incidental. This leads to the following principle for constructing an algebra from a specification (attributed to Burstall in [Wirsing and Broy 82]):

- **no apartheid:** two elements of a carrier are distinct iff they are forced to be different by the axioms.

Unfortunately, this principle is too strong. We can identify *all* the elements of every carrier, resulting in a so-called trivial algebra: every carrier is a singleton set, and every function is defined in the only possible way on the singleton carriers. We are thus led to the following slogan [Wirsing and Broy 82]:

- **no collapse:** trivial algebras are not to be considered as models.

In the category of models of a specification, the "no apartheid" condition corresponds to picking the terminal (or final) object, i.e., an object to which there is a unique morphism from every object in the category. However, with the trivial algebra excluded, a final algebra may not exist. If it does exist, it is unique upto isomorphism, and is considered to be the denotation of a specification when adopting final algebra semantics.

In initial algebra semantics, we start with the term algebra and identify elements of a carrier as indicated by the axioms. In final algebra semantics, we start with a trivial algebra formed by identifying all the elements in the term algebra. We then separate those elements which are deemed to be different by the axioms. Equations are the usual axioms which are used to identify elements. Dually, the natural way to differentiate elements is via inequations. However, inequations soon become unwieldy and hard to reason with.

Fortunately, there is a cleaner and simpler method for adopting final semantics [Guttag 75; Guttag and Horning 78; Kamin 83; Wand 79; Hornung and Raulefs 80]. Data types are specified by extending a set of *primitive types* (with given equality predicates on them). New sorts and operations—called the *types of interest*—are added to the primitive specification. Two elements e_1 and e_2 of the carrier for a new sort are considered distinct if there is an operation (or a derived operation) which maps them to two distinct elements of the carrier for a primitive sort. Thus, the primitive sorts implicitly provide an inequality relation required to obtain a non-trivial final model.

EXAMPLE 3.6. In the SET specification above, the two sorts BOOL (with $\mathrm{T} \neq \mathrm{F}$) and ALPHABET (with $\mathrm{a} \neq \mathrm{b}$) can be considered to be the primitive sorts. The sort SET along with all the set-operations forms the extended specification. The elements $\{\mathrm{a}\}$ and $\{\mathrm{b}\}$ in the carrier for SET are distinct because we can use $\in$ to distinguish them: $\mathrm{a} \in \{\mathrm{a}\} = \mathrm{T}$ while $\mathrm{a} \in \{\mathrm{b}\} = \mathrm{F}$. Using this scheme, the carrier for SET has four distinct elements: $\{\,\}$, $\{\mathrm{a}\}$, $\{\mathrm{b}\}$, and $\{\mathrm{a}, \mathrm{b}\}$. □

We now formally describe specifications which use terminal algebra semantics [Hornung and Raulefs 80]. We use the prefix "t-" to denote that the intended model is the terminal algebra. We will assume a single primitive sort BOOL with two constants T and F with $\mathrm{T} \neq \mathrm{F}$.

DEFINITION 3.7: *t-specification.* A t-specification is a pair $\langle \Sigma, \Phi \rangle$ consisting of a signature Σ and a set of axioms Φ. The signature contains a distinguished sort called BOOL along with two constants T and F of sort BOOL. □

Operation symbols with output sorts other than BOOL are called *constructors.* Operation symbols with output sort BOOL are called *observers.* To adopt final semantics, we require that the elements of BOOL not be identified (no confusion) and that all terms of sort BOOL be provably equal to T or F (no junk).

In the definition below, we use the symbol $\models$ to indicate logical consequence (see definition 4.6).

DEFINITION 3.8: *Consistent, complete.* A t-specification $\langle \Sigma, \Phi \rangle$ is said to be *consistent* if $\Phi \not\models \mathrm{T} = \mathrm{F}$. It is said to be *complete* if for every ground term t of sort BOOL, either $\Phi \models t = \mathrm{T}$ or $\Phi \models t = \mathrm{F}$. □

DEFINITION 3.9: *Final semantics.* A *t-model* of a t-specification $\langle \Sigma, \Phi \rangle$ is a $\langle \Sigma, \Phi \rangle$-model (i.e., a Σ-algebra satisfying Φ) which interprets the sort BOOL as a two element carrier $\{t, f\}$ with $t \neq f$. Such models form a sub-category of the category of models of $\langle \Sigma, \Phi \rangle$.

The final semantics of the specification is the collection of all terminal objects in the sub-category of t-models defined above. □

We now give an explicit construction of the terminal model for specifications using equational logic [Hornung and Raulefs 80]. The idea is to impose a congruence on the ground term algebra which identifies terms which behave the same with respect to BOOL. To define "behave", we need the notion of a context.

DEFINITION 3.10: *Context.* Given a signature $\Sigma = \langle S, \Omega \rangle$ and a sorted set of variables X, an $\langle s, s' \rangle$-*context* is a term $t \in T_{\Sigma(X),s'}$ of sort s' with exactly one free variable of sort s. The set of all such $\langle s, s' \rangle$-contexts will be denoted by $C_\Sigma(s, s')$. □

DEFINITION 3.11: *Terminal congruence.* Let $\Sigma = \langle S, \Omega \rangle$ be a signature, $\mathcal{E}$ be a set of equations, $\langle \Sigma, \mathcal{E} \rangle$ be a t-specification, and $\equiv_\mathcal{E}$ be the (initial) congruence generated by $\mathcal{E}$ (see definition 3.2). The terminal congruence generated by $\mathcal{E}$, denoted by $\sim_\mathcal{E}$, is a family $\{ \sim_{\mathcal{E},s} \mid s \in S \}$ of equivalence relations defined for any two terms p and q of sort s by $p \sim_{\mathcal{E},s} q$ if and only if for all contexts $t \in C_\Sigma(s, \text{BOOL})$ of sort BOOL with a single free variable x_s of sort s, we have $t[x_s/p] \equiv_{\mathcal{E},\text{BOOL}} t[x_s/q]$, where $t[x/y]$ indicates the substitution of x by y in t. □

The terminal congruence $\sim_\mathcal{E}$ coarsens the initial congruence $\equiv_\mathcal{E}$, in the sense that each equivalence class of $\equiv_{\mathcal{E},s}$ is a subset of some equivalence class of $\sim_{\mathcal{E},s}$. The terminal algebra is obtained as the quotient $\mathcal{T}_\Sigma/\sim_\mathcal{E}$ of the ground term algebra (under the additional assumptions that the specification is consistent and complete). For proofs, see [Hornung and Raulefs 80].

In our discussion of final semantics above, we used a single sort BOOL to distinguish terms. We can also use more than one distinguishing sort, as in example 3.6 above (see also [Hornung and Raulefs 81; Guttag 75]). Usually, the distinguishing sorts are constrained to be interpreted initially. Thus the approach of final semantics can be characterized as picking out the final object in the category of consistent and sufficiently complete extensions of some initial model [Kamin 83; Wand 79].

Existence of terminal models

As indicated by the construction above, terminal models exist for equational specifications which are consistent and (sufficiently) complete. This continues to hold even if the axioms are conditional equations [Hornung and Raulefs 80; Wand 79; Kamin 83], inequalities [Hornung and Raulefs 81], or positive formulas (i.e., universally and existentially quantified disjunctions and conjunctions of equations) [Broy et al. 79]. For partial functions and more general axioms, see section 3.3.1.

3.3 Loose semantics

Initial and final semantics are characteristic of the computer science approach to formal systems. Inspired by the theory of programming language semantics, this philosophy dictates that there be a precise and *unique* meaning for every formal construct. On the other hand, if one adopts the approach of mathematics, all models are created equal. Non-standard models are common, and standard models need not be initial. This is a particularly fruitful approach for software engineering, where a specification defines the *necessary* features of a software system. Designating certain models as the intended semantics would be a premature design decision.

DEFINITION 3.12: *Loose semantics.* Given a specification $\langle \Sigma, \Phi \rangle$, its loose semantics is the collection of all Σ-algebras which satisfy the axioms Φ. □

Consider again the specification of natural numbers:

spec NAT =
sorts NAT
operations

$$\begin{aligned} 0 : & \quad \rightarrow \text{NAT} \\ \text{succ} : & \quad \text{NAT} \rightarrow \text{NAT} \\ _+_ : & \quad \text{NAT}, \text{NAT} \rightarrow \text{NAT} \end{aligned}$$

axioms

$m + 0 = m$

$m + \text{succ}(n) = \text{succ}(m + n)$

end

The initial model is the standard model of natural numbers, $\mathcal{N} = \{0, 1, 2, \ldots\}$. However, there are many other non-isomorphic models of this specification. The natural numbers modulo k, for any number k, are also a model. If addition modulo k (written $+_k$) is defined as

$$m +_k n = (m + n) \bmod k$$

where $+$ is the standard addition on $\mathcal{N}$, it is easy to see that $\langle \{0, 1, \ldots, k-1\}, +_k \rangle$ satisfies the axioms in the specification NAT.

Consider another example, the specification STRING / SET of section 3.2. Sets form the final model, while strings (or sequences) form the initial model. There are a number of variants in between: multisets or bags form a model if $\alpha \in x$ is interpreted as "α occurs at least once in the bag x".

Junk, reachability, and induction

When adopting loose semantics, one can add arbitrary junk elements to any carrier of a model, while still satisfying the specification. For example, the carrier $\mathcal{N}_\perp = \{\perp, 0, 1, 2, \ldots\}$ along with functions

$$\begin{aligned} \text{succ} : & \quad \mathcal{N}_\perp \rightarrow \mathcal{N}_\perp \\ _+_ : & \quad \mathcal{N}_\perp, \mathcal{N}_\perp \rightarrow \mathcal{N}_\perp \end{aligned}$$

satisfying the usual axioms for addition and defined to be strict on $\perp$, also forms a model of the specification NAT. However, there are no terms in the specification which can represent $\perp$. To avoid such a problem (the consequence of which is that we cannot reason about such "junk" elements in the carrier) we can restrict the models to be reachable.

DEFINITION 3.13: *Reachability.* A model is said to be *reachable* if every element of every carrier is the interpretation of some ground term of the specification.

Let $\Sigma = \langle S, \Omega \rangle$ be a signature and A be a Σ-algebra. The interpretation t_A of a ground term $t \in T_\Sigma$ in the algebra A is defined inductively as (cf. evaluation, definition 2.9):

1. if t is a constant $c: \to s$, then $t_A = c_A$;
2. if t is the term $f(t_1, \ldots, t_n)$ and the interpretations of $t_1, \ldots, t_n$ are $t_{1A}, \ldots, t_{nA}$ respectively, then $t_A = f_A(t_{1A}, \ldots, t_{nA})$ (i.e., apply the interpretation of f to the interpretations of the arguments).

□

An equivalent definition is that an algebra is reachable if the unique homomorphism[14] from the term algebra is surjective. Thus reachability is equivalent to the "no junk" condition of initial semantics. Reachable models are also called *(finitely) term generated* (reachability is formulated as a "principle of generation" in [Bauer and Wössner 82]). The reachability condition immediately provides a second-order proof rule:

DEFINITION 3.14: *Structural induction on terms.* For any finitely term generated model M, and any formula $\varphi(x)$ with exactly one free variable x of sort s, to prove $M \models \forall x \in s \cdot \varphi(x)$, it suffices to show that

1. for all operations $f: s_1, s_2, \ldots, s_n \to s$ of sort s, with $s_i \neq s$ for each i,
   $M \models \forall x_1, x_2, \ldots, x_n \cdot \varphi(f(x_1, x_2, \ldots, x_n))$; and
2. for all operations $f: s_1, s_2, \ldots, s_n \to s$ of sort s, with $s_i = s$ for some i,
   $M \models \forall x_1, x_2, \ldots, x_n \cdot (\psi_1 \wedge \psi_2 \wedge \cdots \wedge \psi_n \Rightarrow \varphi(f(x_1, x_2, \ldots, x_n)))$,

   where $x_1, x_2, \ldots, x_n$ are variables not occuring in φ and for each i,

$$\psi_i = \begin{cases} \varphi(x_i), & \text{if } s_i = s; \\ \text{true}, & \text{otherwise.} \end{cases}$$

□

Structural induction is an instance of a more general principle, Noetherian induction, which works on any well-founded set, i.e., a partially ordered set containing no infinitely descending chains [Burstall 69; Cohn 81]. For induction on terms, the partial order is the sub-term relation. Thus the induction principle says that if a property is true of all subterms of a term, and is preserved by term constructors, then it is true of all terms.

[14]Provided by the initiality of the term algebra.

EXAMPLE 3.15. As an example of the use of this principle, let us prove that the number of nodes in a binary tree is $(2n+1)$ for some n. Here is a specification for binary trees, and an enrichment with a function which returns a pair of numbers: the number of non-leaf nodes and the number of leaf nodes.

```
spec BIN-TREE =
sorts TREE
operations
   leaf :              → TREE
  node : TREE, TREE → TREE
end
```

```
spec BIN-TREE-1 =
based on
  BIN-TREE,
  NAT-PAIR      /* pairs of natural numbers */
operations
  count : TREE → NAT-PAIR
axioms
  count(leaf) = ⟨0,1⟩      /* non-leaf nodes, leaf nodes */
  count(node(l,r)) = ⟨1 + l_n + r_n, l_l + r_l⟩
       where⟨l_n, l_l⟩ = count(l), ⟨r_n, r_l⟩ = count(r)
end
```

For reachable models of BIN-TREE-1, we will prove the following theorem:

THEOREM. $\forall t \in \text{TREE}\ \exists n \in \text{NAT} \cdot \text{count}(t) = \langle n, n+1\rangle$.

PROOF. We will use structural induction on terms of sorts TREE.

Basis: By definition, $\text{count}(\text{leaf}) = \langle 0, 1\rangle$, which gives $n = 0$.

Induction step: Assume that for some $l, r \in \text{TREE}$, we have $\text{count}(l) = \langle x, x+1\rangle$ and $\text{count}(r) = \langle y, y+1\rangle$. Then, by definition, $\text{count}(\text{node}(l,r)) = \langle 1+x+y, (x+1)+(y+1)\rangle$, whereby we get $n = x + y + 1$.

□

Despite the negative connotation of the word "junk", junk is sometimes useful. Unreachable elements are common in implementations of one algebraic specification by another. Consider an implementation of strings of natural numbers in which strings are represented as arrays with a special token marking the end of the string. This token is usually not a natural number and is thus unreachable in the string specification. Thus junk is a matter of perspective: elements which are unreachable using one signature may be reachable using another.

3.3.1 Existence of initial and terminal models

When adopting loose semantics, it is useful to look at the existence of initial and terminal models. These models impose the finest and coarsest congruences, respectively, on the term algebra and thus bound the range of models.

Broy, Wirsing, and others analyze the lattice structure of term-generated models of a specification [Wirsing and Broy 82; Broy et al. 84; Wirsing and Broy 80; Wirsing et al. 83].

This is based on the observation that there is a unique surjective homomorphism from the term algebra to every term-generated model. This homomorphism induces a congruence on the term algebra. Thus, term-generated algebras[15] can be treated as quotients of the term algebra. It can be shown that the class of all such congruences forms a lattice. In case this lattice is complete, initial and terminal models exist, and are given by the finest and coarsest congruences, respectively.

Wirsing et al. [Wirsing et al. 83] give several conditions for the existence of initial and terminal models, in the context of specifications which use partial functions and axioms from first-order logic. Here we discuss sufficient conditions which impose restrictions on the kinds of axioms used. We first need some definitions.

DEFINITION 3.16: *Universal-existential formula.* Given a signature Σ, a Σ-formula is called universal-existential if it is (first-order equivalent to a Σ-formula) of the form

$$\forall x_1 \in s_1 \ldots \forall x_n \in s_n \exists y_1 \in s'_1 \ldots \exists y_n \in s'_m \cdot \bigwedge_{i=1}^{k} \left(f_1^i \wedge \cdots \wedge f_l^i \Rightarrow g_1^i \vee \cdots \vee g_h^i\right)$$

where all the $f_j^i, j = 1, \ldots, l$ and $g_k^i, k = 1, \ldots, h$ are atomic Σ-formulas (equalities or definedness predicates). □

DEFINITION 3.17: *Maximal formula.* For a specification $\langle \Sigma, \Phi \rangle$ a Σ-formula φ with at most the free variables $x_1, \ldots, x_n$ of sorts $s_1, \ldots, s_n$ is called maximal if for all ground terms $t_1, \ldots, t_n \in T_\Sigma$ of sorts $s_1, \ldots, s_n$ the validity of the sentence (obtained by substituting the variables x_i by the terms t_i in φ)

$$D(t_1) \wedge \cdots \wedge D(t_n) \Rightarrow \varphi[x_1/t_1, \ldots, x_n/t_n]$$

is model-independent. □

In particular, the definedness predicate is maximal if the definedness of terms is independent of the model.

DEFINITION 3.18: *Uniform existential quantifier.* Given a specification $\langle \Sigma, \Phi \rangle$, an existential quantifier for y in a closed prenex Σ-formula $\forall x_1 \in s_1 \ldots \forall x_n \in s_n \exists y \in s' \cdot \varphi$ is called uniform if for all ground terms $t_1, \ldots, t_n \in T_\Sigma$ of sorts $s_1, \ldots, s_n$ there exists a term t' of sorts s' such that

$$D(t_1) \wedge \cdots \wedge D(t_n) \Rightarrow \exists y \in s' \cdot \varphi[x_1/t_1, \ldots, x_n/t_n]$$

is valid iff

$$D(t_1) \wedge \cdots \wedge D(t_n) \Rightarrow D(t') \wedge \varphi[x_1/t_1, \ldots, x_n/t_n, y/t']$$

is valid. □

Uniformity of an existential quantifier roughly means that it can be interpreted by the same term in every model, and can thus be replaced by a Skolem function. The definition can be extended to more than one existential quantifier.

[15] Actually, isomorphism classes of term-generated algebras.

With these definitions, we can give sufficient conditions for the existence of initial and terminal models. Initial models are sensitive to existential quantifiers and disjunctions. Terminal models are sensitive to negations and inequalities.

Initial models exist if the definedness predicate is maximal, all axioms are (closed) universal-existential formulas, all equations (occuring in the axioms) are maximal, and all existential quantifiers are uniform [Wirsing et al. 83].

Terminal models exist if the specification is satisfiable (i.e., has at least one model), the definedness predicate is maximal, all axioms are (closed) universal-existential formulas, and for all inequalities $\neg f_j^i$ of the form $u_j^i \neq v_j^i$ the formula $u_j^i = z$ is maximal (where z is a new variable) [Wirsing et al. 83].

3.4 Behavioural semantics

For many specifications, we are interested in only a part of a model, the part that realizes the *behaviour* described in the specification. Other parts of the model are incidental or auxiliary, although they may be necessary to realize the required behaviour (cf. hidden functions, section 2.7). Restricting attention to certain behaviours is routine in automata theory. For example, Turing machines with one head and one tape, one head and two tapes, or two heads and two tapes, are all equivalent when we restrict our attention to acceptance or rejection of a string in a language. However, they differ when we consider their time or space complexity. Another situation is when we assume that the semantics of a program is its input/output behaviour, the internal states of the computation being irrelevant. This leads us to the notion of *observable* behaviour (a notion first proposed in [Giarratana et al. 76]).

In a behavioural specification, we designate certain parts of the specification as "interesting" or "visible", the other parts being "auxiliary" or "hidden". We then expand the class of models to include those models whose observable behaviour is the same. This technique of providing a specific instance with the intention of describing all models which are abstractly the same as the instance, is called "abstract model specification" by Liskov and Berzins [Liskov and Berzins 79]. This technique is useful when it is easier to describe an abstract model than axiomatically characterizing the intended class of models. It might even be the case that it is impossible to axiomatically characterize the intended class of models within the axioms system being employed.

What, then, is an observation on a specification? In general, an observation is not just an input-output relation but the truth-value of some formula [Sannella and Tarlecki 87]. We give below a general definition of observational equivalence (taken from [Sannella and Tarlecki 87]) which subsumes the different kinds of behavioural equivalence used in the literature. Although this definition is independent of the logic used in the specification, it will be helpful to keep in mind some concrete logic, such as first-order logic. We first assume that the observations are ground formulas, or sentences.

DEFINITION 3.19: *Observational equivalence.* Given a signature Σ and a set Φ of Σ-sentences, two Σ-algebras A and B are said to be observationally equivalent with respect to Φ, written $A \equiv_\Phi B$, if for any sentence $\varphi \in \Phi$, $A \models \varphi$ iff $B \models \varphi$. □

The definition above captures the intuitive notion that when the two algebras are tested with formulas from Φ, they produce the same truth-values.

Observational equivalence using ground formulas is not powerful enough, because the carriers of an algebra may contain unreachable elements ("junk" elements which are not denotable by any term). To handle such algebras, observations have to be formulas containing free variables. We now define observational equivalence which uses formulas which may not be closed.

DEFINITION 3.20: *Observationally reducible.* Given a signature Σ, a set X of variables of sorts in Σ, a set $\Phi(X)$ of Σ-formulas with free variables in X, and two Σ-algebras A and B, the algebra A is said to be observationally reducible to the algebra B with respect to $\Phi(X)$, written $A \leq_{\Phi(X)} B$, if for any valuation $v_A: X \rightarrow |A|$ there exists a valuation $v_B: X \rightarrow |B|$ such that for all formulas $\varphi \in \Phi$, $A \models_{v_A} \varphi$ iff $B \models_{v_B} \varphi$. □

DEFINITION 3.21: *Observational equivalence.* Given a signature Σ, a set X of variables of sorts in Σ, and a set $\Phi(X)$ of Σ-formulas with free variables in X, two Σ-algebras A and B are said to be observationally equivalent with respect to $\Phi(X)$, written $A \equiv_{\Phi(X)} B$, if $A \leq_{\Phi(X)} B$ and $B \leq_{\Phi(X)} A$. □

When the set of variables X is empty, this definition reduces to the definition given for ground formulas.

We can use the notion of observational equivalence to define behavioural semantics for specifications.

DEFINITION 3.22: *Behavioural semantics.* Given a Σ-specification SP, and a set $\Phi(X)$ of Σ-formulas with free variables in X, the behavioural semantics of SP with respect to the "observations" $\Phi(X)$ is the collection of Σ-algebras which are observationally equivalent (wrt $\Phi(X)$) to a model of SP. □

Observe that the category of models used in final algebra semantics is a special case of behavioural semantics; a terminal algebra is a "minimal" realization of some given behaviour [Goguen 73; Goguen and Meseguer 82]. Behaviour for final semantics is usually defined as the truth-values of equations of the form $t_1 = t_2$ where t_1 and t_2 are terms having output sorts in the set of distinguishing (or primitive) sorts. Using this connection, Reichel [Reichel 81] proposes behavioural equivalence as a unifying theme for initial and final semantics. Observational equivalence is also related to the notion of full abstractness from programming language semantics: the semantics of a language is said to be fully abstract if any two entities which cannot be distinguished in any context in the language have the same denotation [Milner 77].

EXAMPLE 3.23. Consider the specification for sets used in section 3.2. We can "observe" the models of this specification using the equation

$$\alpha \in x = b \qquad \alpha \in \text{ALPHABET}, x \in \text{SET}, b \in \text{BOOL}$$

thus implying that membership in a set is the only behaviour in which we are interested. Let Σ_{SET} denote the signature of the SET specification. Consider a Σ_{SET}-algebra which represents sets as heaps, with the heaps being rebalanced after every union operation. In this model, the union operation need not be associative.

Thus, we see that observational equivalence *expands* the class of models to include even those which may not satisfy the axioms in the specification. □

3.5 Operational semantics: Deduction and rewriting

One of the uses of algebraic specifications is to reason about the properties of the system being modelled, or to verify the correctness of an implementation (which might also be an algebraic specification). Such formal reasoning requires a deductive calculus, which is usually provided by the underlying logic.

One-sorted equational logic has very simple rules of deduction: three rules derived from the fact that "=" is an equivalence relation, and one rule stating that "equals can be substituted for equals". We list below the many-sorted versions of these rules.

We assume a signature $\Sigma = \langle S, \Omega \rangle$, sets X and Y of S-sorted variables, Σ-terms t_1, t_2 and t_3 with free variables in X, and Σ-terms u_1 and u_2 with free variables in Y. We denote by $t[x/u]$ the substitution of the variable x in the term t by the term u, provided x and u are of the same sort.

Reflexivity: $$\frac{}{\forall X \cdot t = t}$$

Symmetry: $$\frac{\forall X \cdot t_1 = t_2}{\forall X \cdot t_2 = t_1}$$

Transitivity: $$\frac{\forall X \cdot t_1 = t_2, \forall X \cdot t_2 = t_3}{\forall X \cdot t_1 = t_3}$$

Substitution: $$\frac{\forall X \cdot t_1 = t_2, \forall Y \cdot u_1 = u_2}{\forall X - \{x\} \cup Y \cdot t_1[x/u_1] = t_2[x/u_2]}$$

The naive generalization of one-sorted deduction rules to many-sorted equational logic (and the common practice of declaring variables at the beginning of axiom sets) is not sound, as shown in [Goguen and Meseguer 84]. The variables appearing in an equation have to be explicitly declared in each equation separately[16] and two new deduction rules are needed.

We say that a sort is void in a signature Σ if there are no operations with that sort as the codomain.

[16]This is only necessary if there are sorts whose carriers can be empty.

$$\text{Abstraction:} \qquad \frac{\forall X \cdot t_1 = t_2, y \notin X}{\forall X \cup \{y\} \cdot t_1 = t_2}$$

$$\text{Concretion:} \qquad \frac{\forall X \cdot t_1 = t_2, x \in X_s, \text{ sort } s \text{ non-void}, x \notin t_1, x \notin t_2}{\forall X - \{x\} \cdot t_1 = t_2}$$

The deduction rules for first-order logic can be found in any standard book on logic, e.g., [Enderton 72]. Deduction rules for higher-order logic are given in [Broy 87]. These include the α- and β-conversion rules of the λ-calculus. Depending on the kinds of models, we might additionally have rules for extensionality, fixpoints, and continuity [Möller 86; Möller 85]. For deduction in order-sorted logic, see [Goguen and Meseguer 88].

If a deductive calculus is sound and complete, then we can use it to "execute" the axioms in an algebraic specification, i.e., given a term, we can reduce it another term (hopefully in normal form) using the deduction rules. This technique can be used to test the behaviour of an algebraic specification, to detect certain simple errors, and to ensure that the specification describes the intended model(s).[17] OBJ3 [Goguen and Winkler 88] and ASSPEGIQUE [Bidoit et al. 85] are just two of the many systems which execute algebraic specifications. There is an extensive amount of literature on rewriting techniques, e.g., [Huet and Oppen 80].[18]

[17]The latter assumes that the deduction rules preserve denotations, a key assumption of a logical programming language [Goguen 87b].

[18]See also the workshops on rewriting, Springer LNCS, vols. 202, 256, 355, and 308.

4 STRUCTURE

We have defined specifications to be triples consisting of sorts, operations (and possibly relations), and axioms. As the specifications grow more complex, there may be a large number of sorts, operations, and axioms. Large specifications necessitate structuring mechanisms using which we can build specifications out of smaller, "mind-sized" chunks. We have already used a structuring mechanism in the examples in the previous sections: the "**based on**" construct imports a previously defined specification. Another advantage of building specifications in a structured manner is that we can reason about the properties of specifications more easily: if there are inference rules corresponding to specification building operations,[19] then proofs can also be structured in the same way as the specifications.

It is a commonly held belief in the algebraic specification literature that, independent of the logic used in specifications, the basic structuring mechanisms used in building specifications are essentially the same. We quote from [Goguen and Burstall 85b, page 314]:

> We claim that putting together small specifications to describe complex models is the essence of a specification language; the rest has to do with the particular brand of syntactic sugar and underlying logic that are used.

Thus it is useful to abstractly characterize the components and properties of a logical system which can be used in algebraic specification. Generalizing the different logics discussed in section 2, we can identify the following components: signatures, axioms, specifications, models, satisfaction, and morphisms of signatures, specifications, axioms, and models. The abstract notion of a logical system is called an *institution*, a concept which was first presented by Goguen and Burstall [Goguen and Burstall 83], deriving from their work on the semantics of the specification language Clear. Subsequently, much work has been done in an institution-independent setting, including the structured building of specifications, implementation of one specification by another, specification modules, and observational equivalence of specifications.

The rest of this section is organized as follows. In section 4.1, we define institutions and provide examples. Section 4.2 describes a small set of operations for building specifications in a structured manner. Section 4.3 is devoted to parameterized specifications, and section 4.4 to modules. Second-order constraints useful for building specifications in a hierarchical manner are discussed in section 4.5. Implementation of specifications is discussed in section 4.6.

4.1 Institutions

An institution is an abstract logical system for specifying algebras. Such a logical system consists of two parts: syntax and semantics. The syntax is specified in terms of signatures (which define the basic vocabulary) and the sentences that can be generated over a given signature. The semantics is specified in terms of models associated with a signature and a satisfaction relation between models and sentences. Thus sentences can be used as axioms to constrain the class of models associated with a signature. All these entities are described

[19] Such rules are given in [Sannella and Tarlecki 88a] for the specification building operations described in section 4.2.

using category theory which requires that not only the objects be specified (signatures, sentences, models) but that the morphisms also be specified (renaming of signatures, translation of sentences, homomorphisms of models). These data are subject to the single axiom that the satisfaction relation be preserved under change of signatures, capturing the intuitive notion that truth or validity is independent of the syntax. Figure 3 informally shows the various pieces of an institution, that of equational logic (see section 2.1 for details).

DEFINITION 4.1: *Institution.* An institution I is a 4-tuple $\langle \mathbf{Sig}_I, \mathbf{Sen}_I, \mathbf{Mod}_I, \models \rangle$ consisting of

1. a category $\mathbf{Sig}_I$ of signatures and signature morphisms,
2. a functor $\mathbf{Sen}_I : \mathbf{Sig}_I \to \mathbf{Set}$ (where $\mathbf{Set}$ is the category of sets and functions) which assigns to each signature Σ the set of Σ-sentences, and to each signature morphism $\sigma : \Sigma \to \Sigma'$, the function which translates Σ-sentences to Σ'-sentences,
3. a functor $\mathbf{Mod}_I : \mathbf{Sig}_I \to \mathbf{Cat}^{op}$ (where $\mathbf{Cat}$ is the category of all[20] categories and functors between them) which assigns to each signature Σ the category of Σ-models, and to each signature morphism $\sigma : \Sigma \to \Sigma'$, the functor which translates Σ'-models to Σ-models (note the change in direction), and
4. a satisfaction relation $\models_{I,\Sigma} \subseteq |\mathbf{Mod}_I(\Sigma)| \times \mathbf{Sen}_I(\Sigma)$ between models and sentences for each signature Σ,[21]

subject to the condition that satisfaction be preserved under change of signature:

Satisfaction Condition. For any signature morphism $\sigma : \Sigma \to \Sigma'$ in $\mathbf{Sig}_I$, for any Σ-sentence $\varphi \in \mathbf{Sen}_I(\Sigma)$, and for any Σ'-model $M' \in |\mathbf{Mod}_I(\Sigma')|$,

$$M' \models_{I,\Sigma'} \mathbf{Sen}_I(\sigma)(\varphi) \Leftrightarrow \mathbf{Mod}_I(\sigma)(M') \models_{I,\Sigma} \varphi.$$

□

NOTATION. The subscript specifying the name of the institution in words like $\mathbf{Sig}_I$ will be dropped if the institution is evident from the context. Signatures will be usually denoted by Σ, Σ', Σ_1, etc., and signature morphisms by σ, σ', σ_1, etc. Elements of the set $\mathbf{Sen}(\Sigma)$ will be called Σ-sentences, and the function $\mathbf{Sen}(\sigma) : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}(\Sigma')$ translating Σ-sentences into Σ'-sentences will be ambiguously denoted by σ itself. Objects in the category of models $\mathbf{Mod}(\Sigma)$ will be called Σ-models and morphisms Σ-homomorphisms. The functor $\mathbf{Mod}(\sigma) : \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$ will be called the σ-reduct functor and denoted by $_|_\sigma$. The subscripts for the satisfaction relation will usually be dropped. As an illustration of these conventions, the satisfaction condition is concisely written as

[20]The category of all categories leads to foundational difficulties [Mac Lane 71, section 1.6] similar to Russell's paradox. We can avoid this by considering only those categories which are small with respect to some universe.

[21]The notation $|\mathcal{C}|$ denotes the collection of objects of the category $\mathcal{C}$.

Signature STR	
sorts	
ALPHA, STR	
operations	
‘ ’ :	$\rightarrow$ STR
‘_’ :	ALPHA $\rightarrow$ STR
$_\cdot_$:	STR, STR $\rightarrow$ STR

Signature Morphism σ: STR $\rightarrow$ SET		
ALPHA	$\mapsto$	NAT
STR	$\mapsto$	SET
‘ ’	$\mapsto$	$\{\,\}$
‘_’	$\mapsto$	$\{_\}$
$_\cdot_$	$\mapsto$	$_\cup_$

Signature SET	
sorts	
BOOL, NAT, SET	
operations	
T, F :	$\rightarrow$ BOOL
$\{\,\}$:	$\rightarrow$ SET
$\{_\}$:	NAT $\rightarrow$ SET
$_\cup_$:	SET, SET $\rightarrow$ SET
$_\in_$:	NAT, SET $\rightarrow$ BOOL

STR-Sentences
$x\cdot(y\cdot z)=(x\cdot y)\cdot z$
$x\cdot$‘ ’$=x$
$x\cdot y=y\cdot x$

Sentence Translation
$x\cdot(y\cdot z)=(x\cdot y)\cdot z$
$\mapsto x\cup(y\cup z)=(x\cup y)\cup z$
$x\cdot$‘ ’$=x$
$\mapsto x\cup\{\,\}=x$
$x\cdot y=y\cdot x$
$\mapsto x\cup y=y\cup x$

SET-Sentences
$x\cup(y\cup z)=(x\cup y)\cup z$
$x\cup\{\,\}=x$
$x\cup y=y\cup x$
$\alpha\in\{\,\}=\mathrm{F}$
$\alpha\in(x\cup y)=\alpha\in x\vee\alpha\in y$

STR-Model, $\mathcal{A}$	
A	ASCII characters
$A*$	ASCII strings
‘ ’	empty string
‘_’	singleton string
$_\cdot_$	concatenate

STR-model, $\mathcal{N}\|_\sigma$ (induced by σ)	
N	natural numbers
$\mathcal{P}(N)$	powerset of N
$\{\,\}$	empty set
$\{_\}$	singleton set
$_\cup_$	union

SET-Model, $\mathcal{N}$	
$\{t,f\}$	truth-values
N	natural numbers
$\mathcal{P}(N)$	powerset of N
$\{\,\}$	empty set
$\{_\}$	singleton set
$_\cup_$	union
$_\in_$	membership

STR-Satisfaction		
$\mathcal{A}$	$\models$	$x\cdot(y\cdot z)=(x\cdot y)\cdot z$
$\mathcal{A}$	$\models$	$x\cdot$‘ ’$=x$
$\mathcal{A}$	$\not\models$	$x\cdot y=y\cdot x$
$\mathcal{A}$	$\not\models$	$x\cdot x=x$

STR-Satisfaction (induced by σ)		
$\mathcal{N}\|_\sigma$	$\models$	$x\cdot y=y\cdot x$
$\mathcal{N}\|_\sigma$	$\models$	$x\cdot x=x$

SET-Satisfaction		
$\mathcal{N}$	$\models$	$x\cup y=y\cup x$
$\mathcal{N}$	$\models$	$a\in\{a\}=\mathrm{T}$
$\mathcal{N}$	$\models$	$x\cup x=x$
$\mathcal{N}$	$\not\models$	$\mathrm{T}=\mathrm{F}$

Figure 3: Pieces of an institution

$$M' \models \sigma(\varphi) \Leftrightarrow M'|_\sigma \models \varphi.$$

EXAMPLE 4.2: *Equational logic.* Equational logic, as described in section 2.1 is an institution, denoted by EQ. $\mathbf{Sig}_{\mathrm{EQ}}$ is the category of signatures (definition 2.1) and signature morphisms (definition 2.16). The sentences in this institution are equations (definition 2.5). Thus the functor $\mathbf{Sen}_{\mathrm{EQ}}$ assigns to each signature Σ the set of Σ-equations, and to each signature morphism σ the translation function σ' (definition 2.19). The functor $\mathbf{Mod}_{\mathrm{EQ}}$ assigns to each signature Σ the category of Σ-algebras (definition 2.8) and Σ-homomorphisms (definition 2.14). $\mathbf{Mod}_{\mathrm{EQ}}$ also assigns to each signature morphism σ the σ-reduct functor (definition 2.21). The satisfaction relation between Σ-algebras and Σ-equations is given in definition 2.11. A proof of the satisfaction condition for equational logic can be found in [Goguen and Burstall 85a]. □

Similar to equational logic, the other logics described in section 2 form institutions: first-order logic, conditional equational logic, many-sorted predicate calculus (with and without equality), Horn-clause logic, partial first-order logic, and order-sorted logic.

EXAMPLE 4.3: *Behavioural specifications.* Nivela and Orejas [Nivela and Orejas 87] propose an institution to describe behavioural semantics of specifications. This institution is interesting because it does not arise directly from a standard logic. It is similar to the institution of conditional equational logic, except that

1. the sorts in a signature are divided into observable and non-observable sorts;
2. an algebra satisfies an equation if it satisfies all the observable consequences of the equation; and
3. morphisms between algebras are homomorphisms which preserve behaviour.

The result of these changes is that isomorphic models (in the category of models and behaviour preserving homomorphisms) are behaviourally equivalent. Initial models provide a combination of initial semantics and behavioural semantics. □

Specifications, theories, free variables

We now show a few examples of general constructions using the apparatus of institutions.

DEFINITION 4.4: *Specification.* A specification SP is a pair $\langle \Sigma, \Phi \rangle$ consisting of a signature Σ and a collection Φ of Σ-sentences (i.e., $\Phi \subseteq \mathbf{Sen}(\Sigma)$). A model of a specification $SP = \langle \Sigma, \Phi \rangle$ is a Σ-model M such that $M \models \varphi$ for each $\varphi \in \Phi$. The collection of all such models M will be denoted by $Mod[SP]$. The subcategory of $\mathbf{Mod}(\Sigma)$ induced by $Mod[SP]$ will also be denoted by $Mod[SP]$. □

DEFINITION 4.5: *Specification morphism.* A specification morphism from a specification $SP = \langle \Sigma, \Phi \rangle$ to a specification $SP' = \langle \Sigma', \Phi' \rangle$ is a signature morphism $\sigma: \Sigma \to \Sigma'$ such that for any model $M \in Mod[SP']$ we have $M|_\sigma \in Mod[SP]$. The specification morphism is also denoted by the same symbol, $\sigma: SP \to SP'$. □

DEFINITION 4.6: *Logical Consequence.* Given a signature Σ, a Σ-sentence φ is said to be a logical consequence of the Σ-sentences $\varphi_1, \ldots, \varphi_n$, written $\varphi_1, \ldots, \varphi_n \models \varphi$, if each Σ-model that satisfies the sentences $\varphi_1, \ldots, \varphi_n$ also satisfies φ. □

DEFINITION 4.7: *Closure, closed.* Given a signature Σ, the closure $\overline{\Phi}$ of a set of Σ-sentences Φ is the set of all Σ-sentences which are the logical consequence of Φ, i.e., $\overline{\Phi} = \{ \varphi \mid \Phi \models \varphi \}$. A set of Σ-sentences Φ is said to be closed if and only if $\Phi = \overline{\Phi}$. □

DEFINITION 4.8: *Theory, presentation.* A theory T is a pair $\langle \Sigma, \Phi \rangle$ consisting of a signature Σ and a closed set Φ of Σ-sentences. A specification $\langle \Sigma, E \rangle$ is said to be a presentation for a theory $\langle \Sigma, \Phi \rangle$ if $\overline{E} = \Phi$. A model of a theory is defined just as for specifications; the collection of all models of the theory T is denoted by $Mod[T]$. Theory morphisms are defined analogous to specification morphisms. □

There is an alternative, more intuitive, way of defining specification and theory morphisms using logical consequence. A specification morphism $\sigma: SP \rightarrow SP'$ is a signature morphism σ such that for each $\varphi \in \Phi$, the translated sentence $\sigma(\varphi)$ is a logical consequence of Φ' (for a theory morphism, the translated sentence *belongs* to the target theory). This follows from a simple application of the satisfaction condition.

We can also adopt a proof-theoretic viewpoint, and substitute deduction for logical consequence. A theory would then be defined as a pair $\langle \Sigma, \Phi \rangle$ where Φ is closed under deduction. This is equivalent to the definition given above if the rules of deduction are sound and complete. However, the definition of institutions above does not include a notion of proof. This can be easily added; see [Goguen and Burstall 85b; Meseguer 89].

Specifications and specification morphisms form a category called **Spec**. Theories and theory morphisms form a category called **Theory**. The abstract machinery of institutions allows us to prove that, if the category **Sig** of signatures is cocomplete, then so are the categories **Spec** and **Theory** (see [Goguen and Burstall 84a; Goguen and Burstall 84b] or [Rydeheard and Burstall 88, Chapter 9]). Thus, if we can "put together" signatures using colimits, then we can put together specifications and theories built using those signatures.

Institutions do not have the notion of a "free variable" in a sentence because they are directly based on the notion of satisfaction. However, free variables are sometimes necessary, e.g., the definition of observational equivalence (definition 3.21). We show here the technique used in [Sannella and Tarlecki 88a] to add free variables to an institution. Free variables can be thought of as extending a signature with extra constants, one for each variable.

DEFINITION 4.9: *Formula, valuation.* If $\theta: \Sigma \rightarrow \Sigma'$ is a signature morphism, then Σ'-sentences can be thought of as Σ-formulas with free variables in "$\Sigma' - \theta(\Sigma)$".[22] A valuation of the free variables into a Σ-model M can be thought of as a Σ'-model M' with $M'|_\theta = M$. □

This definition does not put enough constraints on what it means to be a free variable; but it is sufficient to define observational equivalence, as in [Sannella and Tarlecki 87] (cf. definition 3.21).

[22] This notation is only to aid the intuition. Images of signature morphisms and differences of signatures need not be defined in an arbitrary institution.

4.1.1 Why all this abstractness?

The abstract approach of institutions is necessary to get away from the idiosyncrasies of the syntax and the logic, and to focus upon the essential problem of building a large piece of formal text in a structured manner. It is because of such unifying notions that research in algebraic specification is much more coherent than research in, say, programming languages. The myriad programming languages with countless variations of syntax which are around only serve to obfuscate the underlying issue of building large programs in an intelligible way.

A second situation in which the need for an abstract characterization arises is when we want to use more than one logical system in a specification, e.g., with some parts of the specification in equational logic, some parts in temporal logic, and others in a logic supporting imperative features. In such a situation, it is essential that we abstract away from the individual logics to establish a common basis with which we can connect different logics. Institution morphisms (translations of signatures, sentences, and models across institutions) provide a way of doing this [Goguen and Burstall 83; Goguen and Burstall 85b; Meseguer 89; Sannella and Tarlecki 88b; Tarlecki 85a]. Specifying in multiple institutions is still an open problem because there are several technical difficulties involved. It should be noted that the abstract machinery of institutions allow us to precisely state the problem; thus institutions are a step in the right direction.

4.2 Institution-independent operations

As noted at the beginning of this section (section 4), the essential purpose of a specification language is to provide operations for building specifications in a structured manner. There is a surprising amount of commonality among the various algebraic specification languages that have been described in the literature (see section 5). In this section, we will describe some of these common methods of structuring specifications.

Assumptions

For an institution to be useful in building specifications, we assume that it provides some tools for "putting things together". Specifically, we will assume that the category **Sig** of signatures has all colimits and that the functor **Mod** preserves these colimits, i.e., it translates them into limits in **Cat**, the category of all categories (colimits go to limits because **Mod** is contravariant).

Levels of semantics

The semantics of operations for building specifications can be given at three levels: the presentation level (signatures and axioms), the theory level (sets of sentences closed under logical consequence), and the model level (the category of models for a specification). The most fruitful approach would be to describe the operations at all the three levels. Of course, we have to impose the condition that the semantics defined at one level is compatible with that defined at another level.

Theories are usually infinite and presentations are usually finite. Thus it is easier to deal with presentations. Goguen and Burstall [Goguen and Burstall 84a] show the soundness of performing operations on theories by using their counterparts at the presentation

level. Similarly, models are usually abstract and infinite; we can only deal with concrete representations of the models (using theories or presentations). Compatibility between presentation semantics and denotational semantics (model level) is embodied in the principle of compositionality:

DEFINITION 4.10: *Compositionality.* If f is an operation symbol of n arguments, and the function $\mathcal{M}$ (the "meaning" function) maps syntactic objects onto their denotations, then $\mathcal{M}$ is compositional if

$$\mathcal{M}(f(a_1,\ldots,a_n)) = \mathcal{M}(f)(\mathcal{M}(a_1),\ldots,\mathcal{M}(a_n)).$$

□

An elegant formulation of this principle if the idea that the syntax of a language is an initial algebra, and the semantics is a homomorphism of the initial algebra into an algebra of "meanings" [Goguen et al. 77].

In this section, we only describe the semantics of specification building operations at the model level (consistent with the model-theoretic bias of institutions). The design of syntactic and proof-theoretic counterparts of these operations is an open problem (however, see [Sannella and Tarlecki 85; Bidoit 89]).

Specification building operations

A basic set of specification building operations was proposed in the kernel language ASL [Sannella and Wirsing 83; Wirsing 86]. We present below Sannella and Tarlecki's generalization of these operations to an arbitrary institution [Sannella and Tarlecki 88a]. There are eight specification building operations which seem to form a useful kernel for algebraic specification languages (summarized in Figure 4):

1. build a specification from a signature and a collection of axioms;
2. form the union of a family of specifications;
3. translate a specification via a signature morphism;
4. hide some details of a specification while preserving its models;
5. constrain the models of a specification to be minimal;
6. close the class of models under isomorphism;
7. expand the class of models using observational equivalence; and
8. parameterize a specification.

These operations do not form a kernel in the technical sense that all specification building operations can be generated from the kernel. However, they are adequate for describing (with minimum machinery and assumptions) those structuring operations which are commonly found in algebraic specification languages.

We will now describe these operations in detail (except parameterization, which is treated in the next section). We will sometimes assume that signatures consist of sorts and operations (as in equational logic or first-order logic).

Operation	Sig	Models
$\langle\Sigma, \Phi\rangle$	Σ	$\{ M \mid M \in \lvert\mathbf{Mod}(\Sigma)\rvert \text{ and } M \models \Phi \}$
$\bigcup_{i\in I} SP_i$	Σ	$\bigcap_{i\in I} Mod[SP_i]$
translate SP **by** σ	Σ'	$\{ M' \mid M' \in \lvert\mathbf{Mod}(\Sigma')\rvert \text{ and } M'\rvert_\sigma \in Mod[SP] \}$
derive from SP' **by** σ	Σ	$\{ M'\rvert_\sigma \mid M' \in Mod[SP'] \}$
iso close SP	Σ	$\{ M \mid M \in \lvert\mathbf{Mod}(\Sigma)\rvert \text{ and } \exists N \in Mod[SP] \cdot M \cong N \}$
minimal SP **wrt** σ	Σ	$\{ M \mid M \text{ is } \sigma\text{-minimal in } Mod[SP] \}$
abstract SP **wrt** $\Phi(X)$	Σ	$\{ M \mid M \in \lvert\mathbf{Mod}(\Sigma)\rvert \text{ and } \exists N \in Mod[SP] \cdot M \equiv_{\Phi(X)} N \}$
$\lambda X{:}\,\Sigma_{\mathrm{par}} \cdot SP_{\mathrm{res}}$	Σ_{res}	$Mod[(\lambda X{:}\,\Sigma_{\mathrm{par}} \cdot SP_{\mathrm{res}})(SP)] = Mod[SP_{\mathrm{res}}(X/SP)]$

Figure 4: General operations for building specifications

NOTATION. For a category $\mathcal{C}$, the collection of objects in $\mathcal{C}$ will be denoted by $|\mathcal{C}|$. For a specification SP, the signature and the collection of models of SP will be denoted by $Sig[SP]$ and $Mod[SP]$, respectively.

DEFINITION 4.11: *Basic Specifications.* If $\Sigma \in |\mathbf{Sig}|$ is a signature and $\Phi \subseteq \mathbf{Sen}(\Sigma)$ is a set of Σ-sentences, then $\langle\Sigma, \Phi\rangle$ is a basic specification whose models are all Σ-models satisfying the sentences in Φ.

$$\begin{aligned} Sig[\langle\Sigma, \Phi\rangle] &= \Sigma \\ Mod[\langle\Sigma, \Phi\rangle] &= \{ M \mid M \in |\mathbf{Mod}(\Sigma)| \text{ and } M \models \Phi \} \end{aligned}$$

□

This is the basic tool for building small specifications. This base of small specifications can then be extended using the other building operations. Here is a basic specification for bags. Although this specification is called BAG, its semantics is loose and models include strings, sets, and other such structures. We use the notation **spec** _ = **sorts** _ **operations** _ **axioms** _ **end** rather than $\langle\Sigma, \Phi\rangle$.

spec BAG =
sorts BAG
operations

$$\begin{array}{rl} \{\}: & \to \text{BAG} \\ \{_\}: & \text{ANY} \to \text{BAG} \\ _\cup_: & \text{BAG}, \text{BAG} \to \text{BAG} \end{array}$$

axioms

$x \cup (y \cup z) = (x \cup y) \cup z$
$x \cup \{\} = x$
$\{\} \cup x = x$

end

DEFINITION 4.12: *Union.* If $\{ SP_i \mid i \in I \text{ and } Sig[SP_i] = \Sigma \}$ is a family of specifications with the same signature Σ, then $\bigcup_{i \in I} SP_i$ is the union of these specifications with models as those Σ-models which satisfy each of the specifications individually.

$$\begin{aligned} Sig[\bigcup_{i \in I} SP_i] &= \Sigma \\ Mod[\bigcup_{i \in I} SP_i] &= \bigcap_{i \in I} Mod[SP_i] \end{aligned}$$

□

This is the fundamental operation for combining specifications. Note the restriction that each of the specifications have the same signature. Higher-level operations that combine specifications with different signatures can be built using the translate operation (defined below) in conjunction with the union operation.

DEFINITION 4.13: *Translate.* If SP is a Σ-specification and $\sigma: \Sigma \to \Sigma'$ is a signature morphism, then **translate** SP **by** σ is a Σ'-specification whose models are extensions of SP-models.

$$\begin{aligned} Sig[\textbf{translate } SP \textbf{ by } \sigma] &= \Sigma' \\ Mod[\textbf{translate } SP \textbf{ by } \sigma] &= \{ M' \mid M' \in |\mathbf{Mod}(\Sigma')| \text{ and } M'|_\sigma \in Mod[SP] \} \end{aligned}$$

□

Translate is the basic renaming operation. Together with union, it can be used to define a "sum" operation. For example, if SP is a Σ-specification, SP' is a Σ'-specification, and $\iota: \Sigma \hookrightarrow \Sigma \cup \Sigma'$ and $\iota': \Sigma' \hookrightarrow \Sigma \cup \Sigma'$ are the obvious inclusions of Σ and Σ' into their union, then we can define

$$SP + SP' \stackrel{\text{def}}{=} (\textbf{translate } SP \textbf{ by } \iota) \cup (\textbf{translate } SP' \textbf{ by } \iota')$$

thus giving

$$
\begin{aligned}
Sig[SP+SP'] &= \Sigma \cup \Sigma' \\
Mod[SP+SP'] &= \{\, M \mid M \in |\mathbf{Mod}(\Sigma\cup\Sigma')|, \\
&\qquad M|_{\iota} \in Mod[SP], \text{ and } M|_{\iota'} \in Mod[SP'] \,\}
\end{aligned}
$$

To prevent name clashes and permit sharing, the sum operation could also be defined as a colimit (as in the language Clear [Burstall and Goguen 79], see also section 5.4). Using "+", we can define an operation to enrich a specification $SP = \langle\langle S,\Omega\rangle,\Phi\rangle$ by new sorts S', operations Ω', and axioms Φ':

$$\textbf{enrich } SP \textbf{ by sorts } S' \textbf{ operations } \Omega' \textbf{ axioms } \Phi' \stackrel{\text{def}}{=} SP + \langle\langle S\cup S', \Omega\cup\Omega'\rangle, \Phi'\rangle$$

As an example, we can enrich our BAG specification with a membership relation; but first, we have to combine the specification with that of BOOL (see example 3.5).

spec BAG-1 =
enrich BAG + BOOL **by**
operations
  $_ \in _$: ANY, BAG $\rightarrow$ BOOL
axioms
  $\alpha \in \{\} = \mathrm{F}$
  $\alpha \in \{\beta\} =$ **if** $\alpha = \beta$ **then** T **else** F
  $\alpha \in (x \cup y) = \alpha \in x \vee \alpha \in y$
end

Many specification languages have different kinds of enrich operations to control the protection of the specification which is enriched. We will illustrate the kinds of enrichments possible in the context of initial semantics for specifications. Let SP' be an enrichment of the specification SP. Let I and I' be the initial models of SP and SP' respectively. Let $I'|_{SP}$ be the SP-reduct of I'.

SP' is said to be a *consistent* extension of SP if no two distinct elements of any carrier in I are identified in $I'|_{SP}$ ("no new confusion"). SP' is said to be a *sufficiently complete* extension if the carriers of $I'|_{SP}$ contain no elements which are not present in the carriers of I ("no new junk"). SP' is a *conservative* extension (or persistent extension) if it is both consistent and sufficiently complete, i.e., the model I "persists" in I'.

For final semantics, the definitions are essentially the same (definition 3.8). For the case of loose semantics, see definition 4.43.

The enrichment above of BAG + BOOL to BAG-1 is persistent, for loose semantics, because the addition of the membership operation does not affect the rest of the model. However, if we enrich BAG-1 to BAG-2 as shown below, we only get a sufficiently complete extension.

spec BAG-2 =
enrich BAG-1 **by**
operations
  insert : ANY, BAG $\rightarrow$ BAG
axioms
  $\mathrm{insert}(\alpha, x) = \{\alpha\} \cup x$
  $\mathrm{insert}(\alpha, \mathrm{insert}(\alpha, x)) = \mathrm{insert}(\alpha, x)$
end

DEFINITION 4.14: *Derive.* If $\sigma: \Sigma \to \Sigma'$ is a signature morphism and SP' is a Σ'-specification, then **derive from** SP' **by** σ is a Σ-specification whose models are σ-reducts of SP'-models.

$$\begin{aligned} Sig[\textbf{derive from } SP' \textbf{ by } \sigma] &= \Sigma \\ Mod[\textbf{derive from } SP' \textbf{ by } \sigma] &= \{ M'|_\sigma \mid M' \in Mod[SP'] \} \end{aligned}$$

□

Derive is a kind of inverse operation to translate. It can be used to hide parts of a signature while preserving the class of models. Thus it is useful in algebraic implementations (see section 4.6).

In our BAG example, the enrichment with BOOL introduced extra operations like $\wedge$, $\vee$, and $\neg$. We can hide these operations by using "derive" with a signature morphism $Sig[\text{BAG-3}] \to Sig[\text{BAG-1}]$ defined below.

spec BAG-3 =
derive from BAG-1 **by**
BOOL $\mapsto$ BOOL
T $\mapsto$ T
F $\mapsto$ F
BAG $\mapsto$ BAG
$\{\} \mapsto \{\}$
$\{_\} \mapsto \{_\}$
$_ \cup _ \mapsto _ \cup _$
$_ \in _ \mapsto _ \in _$
end

DEFINITION 4.15: *Isomorphism class.* If SP is a Σ-specification, then **iso close** SP is the specification with models being the isomorphism class of the models of SP.

$$\begin{aligned} Sig[\textbf{iso close } SP] &= \Sigma \\ Mod[\textbf{iso close } SP] &= \{ M \mid M \in |\mathbf{Mod}(\Sigma)| \text{ and } \exists N \in Mod[SP] \cdot M \cong N \} \end{aligned}$$

□

This operation is necessary because it does not follow from the axioms for institutions that model classes are closed under isomorphism. It is typically needed after a "derive" operation.

DEFINITION 4.16: *Minimal.* If *SP* is a Σ-specification and $\sigma\colon\Gamma \to \Sigma$ is a signature morphism, then **minimal** *SP* **wrt** σ is a Σ-specification whose models are minimal extensions of their σ-reducts.

$$\begin{aligned} Sig[\textbf{minimal } SP \textbf{ wrt } \sigma] &= \Sigma \\ Mod[\textbf{minimal } SP \textbf{ wrt } \sigma] &= \{ M \mid M \text{ is } \sigma\text{-minimal in } Mod[SP] \} \end{aligned}$$

□

The notion σ-minimal used above needs some explanation. First, recall that an algebra is called minimal if it has no proper sub-algebras. In an arbitrary institution, sub-models are represented by monomorphisms,[23] i.e., A is a sub-model of B iff there is a monomorphism $m\colon A \rightarrowtail B$. Now, let $\sigma\colon\Gamma \to \Sigma$ be a signature morphism as above, and $K \subseteq |\mathbf{Mod}(\Sigma)|$ be a collection of Σ-models. We say that B is σ-minimal in K if $B \in K$ and B contains no proper sub-model with an isomorphic σ-reduct. Formally,

DEFINITION 4.17: *σ-minimal.* Given a model $B \in K \subseteq |\mathbf{Mod}(\Sigma)|$ and a signature morphism $\sigma\colon\Gamma \to \Sigma$, B is σ-minimal in K if for every monomorphism (sub-model) $m\colon A \rightarrowtail B$ in $\mathbf{Mod}(\Sigma)$ for which we have $m|_\sigma\colon A|_\sigma \cong B|_\sigma$ (i.e., the σ-reducts of A and B are isomorphic in $\mathbf{Mod}(\Gamma)$), the monomorphism m is an isomorphism, i.e., $A \cong B$ (in $\mathbf{Mod}(\Sigma)$). □

Roughly, "minimal" produces models which are minimal extensions (no junk) of their σ-reducts. The operation "minimal" can be used to define the standard model $\mathcal{N}$ of natural numbers. Let NAT = $\langle \Sigma_{\mathrm{NAT}}, \Phi_{\mathrm{NAT}} \rangle$ be the following specification:

spec NAT =
sorts
  NAT
operations
  $0 : \quad \to \mathrm{NAT}$
  $\mathrm{succ} : \mathrm{NAT} \to \mathrm{NAT}$
axioms
  $0 \neq \mathrm{succ}(x)$
  $(\mathrm{succ}(x) = \mathrm{succ}(y)) \Rightarrow x = y$ /* the successor function is injective */
end

If $\iota_{\mathrm{NAT}}\colon\emptyset \to \Sigma_{\mathrm{NAT}}$ is the inclusion of the empty signature into the signature of NAT, then the standard model is given by

$$\mathcal{N} = \textbf{minimal } \mathrm{NAT} \textbf{ wrt } \iota_{\mathrm{NAT}}.$$

The axioms for "succ" ensure that no two natural numbers are identified. The **minimal** operation ensures that there is no junk.

[23]Roughly speaking, monomorphisms are injective functions.

The "minimal" operation can also be used to define a "reachability" operation on models. A model is reachable if all the elements of the carriers can be represented by some ground term (see definition 3.13). There is also a weaker notion in which only some carriers are required to be reachable. If $\Sigma = \langle S, \Omega \rangle$ is a signature, and $R \subseteq S$ is a subset of the sorts, then a Σ-algebra A is said to be *reachable on* R, if every element of a carrier in R is the interpretation (under some valuation) of a term built using operations from Ω and free variables from the carriers $S - R$.

This notion of reachability can be defined using the **minimal** operation. For a Σ-specification SP and a subset of sorts $S \subseteq \text{sorts}(\Sigma)$, let **reachable** SP **on** S be a specification building operation defined by

$$\begin{aligned} \mathit{Sig}[\textbf{reachable } SP \textbf{ on } S] &= \Sigma \\ \mathit{Mod}[\textbf{reachable } SP \textbf{ on } S] &= \{ M \mid M \in \mathit{Mod}[SP] \text{ and } M \text{ is reachable on } S \} \end{aligned}$$

Then,

$$\textbf{reachable } SP \textbf{ on } S \stackrel{\text{def}}{=} \textbf{reachable } SP \textbf{ wrt } \iota$$

where $\iota: \langle \text{sorts}(\Sigma) - S, \emptyset \rangle \hookrightarrow \Sigma$ is the inclusion of the complement of S into Σ, and **reachable _ wrt _** is defined by

$$\textbf{reachable } SP \textbf{ wrt } \sigma \stackrel{\text{def}}{=} SP + \textbf{minimal } \langle \mathit{Sig}[SP], \emptyset \rangle \textbf{ wrt } \sigma.$$

Continuing our bag example, let

spec BAG-4 =
enrich BAG-3 **by**
axioms
  $T \neq F$
end

We can ensure that the interpretation of the sub-specification BOOL is initial, and that there is no junk in the carrier for BAG, by using

$$\textbf{spec } \text{BAG-5} = \textbf{reachable } \text{BAG-4} \textbf{ on } \{\text{BOOL}, \text{BAG}\}.$$

DEFINITION 4.18: *Abstract.* If SP is a Σ-specification, X is a set of variables[24] with sorts in Σ, $\Phi(X)$ is a set of Σ-formulas with free variables in X, then **abstract** SP **wrt** $\Phi(X)$ expands the class of models of SP to those Σ-models which are observationally equivalent (see definition 3.21) to some model of SP.

$$\begin{aligned} \mathit{Sig}[\textbf{abstract } SP \textbf{ wrt } \Phi(X)] &= \Sigma \\ \mathit{Mod}[\textbf{abstract } SP \textbf{ wrt } \Phi(X)] &= \{ M \mid M \in |\textbf{Mod}(\Sigma)| \text{ and} \\ &\qquad \exists N \in \mathit{Mod}[SP] \cdot M \equiv_{\Phi(X)} N \} \end{aligned}$$

□

In our BAG example, we can expand the class of models by saying that the relevant operator for distinguishing between bags is the membership relation, $\in$. This can be done by

abstract BAG-3 **wrt** $\{\alpha \in x = b\}$

where α is variable of sort ANY, x is a variable of sort BAG, and b is a variable of sort BOOL. Now we can have models in which the union operation, $\cup$, need not be associative. An example of such a model is a heap representation of bags, with a balancing step after every union operation.

4.2.1 An example: Parity automaton

We now build a small specification—that of a finite state automaton to recognize the parity of a bit-string—to illustrate the usage of the specification building operations described above, and some of the interactions between these operations.

Informal description of the intended models

The intent is to describe deterministic finite state automata whose input consists of strings built from the alphabet $\{0, 1\}$ (called bit-strings), and which accept strings whose parity is even. The parity of a bit-string is even if the number of 1's in the string is even. We will also show specifications for automata which do not have unreachable states, and minimal automata which realize the behaviour (of accepting even-parity bit-strings).

NOTATION. We will use the specification building operations **initial** *SP* and **final** *SP* to denote specifications whose models are the initial and final models of *SP*.

Formal specification of even-parity automata

We will start with a basic specification for boolean values which will be required later. The intent is to specify an algebra with one carrier containing exactly two (distinct) elements.

initial spec BOOL =
sorts BOOL
operations
  T : $\to$ BOOL
  F : $\to$ BOOL
end

Next, we define bits and bit-strings. The use of initial semantics for specifying bit-strings considerably simplifies the specification (see the specification BIT-STRING$'$ for another way of achieving the same effect).

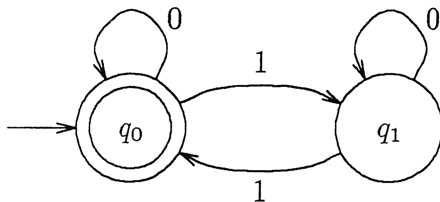
[24] Strictly speaking, institutions do not support free variables directly. Sannella and Tarlecki [Sannella and Tarlecki 88a] show how to introduce free variables into institutions (see also definition 4.9), and define observational equivalence and the **abstract** operation in an arbitrary institution. We prefer to give here the more intuitive definition which uses free variables, rather than the institution-independent definition.

spec BIT =
translate BOOL **by**
BOOL $\mapsto$ BIT
T $\mapsto$ 1
F $\mapsto$ 0
end

initial spec BIT-STRING =
enrich BIT
sorts BIT-STRING
operations
ϵ : $\rightarrow$ BIT-STRING /* the empty string */
_ _ : BIT, BIT-STRING $\rightarrow$ BIT-STRING /* adjoin a bit to a string */
end

Here are some examples of bit-strings: ϵ, 1ϵ, 101ϵ, 10110ϵ.

We are now ready to describe an automaton to recognize even parity. The automaton contains two states: one each for the even/odd parity of the input string seen so far. The state is not changed on an input of 0. The state is flipped on an input of 1. The automaton starts in a state of even parity, which is also the accepting state.

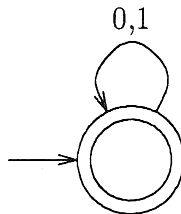


spec PARITY-AUTOMATON =
enrich BOOL, BIT
sorts Q
operations
init : $\rightarrow Q$ /* the initial state */
q_0, q_1 : $\rightarrow Q$ /* states in the automaton */
is-final : $Q \rightarrow$ BOOL /* predicate for final states */
$\delta : Q$, BIT $\rightarrow Q$ /* transition function */
axioms
init = q_0
is-final(q_0) = T is-final(q_1) = F
$\delta(q_0, 0) = q_0$ $\delta(q_0, 1) = q_1$
$\delta(q_1, 0) = q_1$ $\delta(q_1, 1) = q_0$
end

The specification PARITY-AUTOMATON is a transcription of the figure of the automaton above. However, since the semantics of PARITY-AUTOMATON is loose, it has an infinite number of models. All these models contain the two states and the transition structure shown in the figure above. In addition, they can contain an arbitrary number of unreachable states with arbitrary transitions. We can exclude models with unreachable states by using

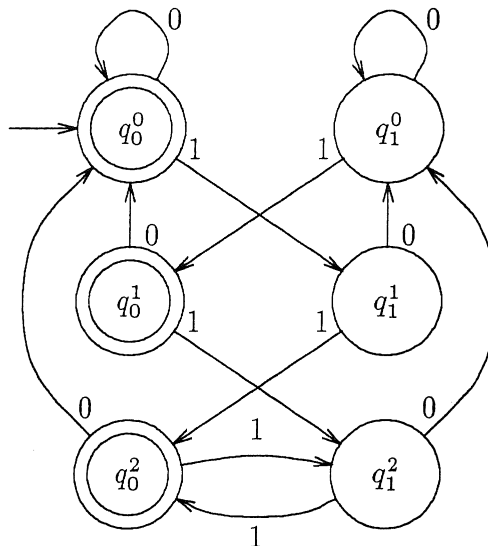
spec REACHABLE-PARITY-AUTOMATON =
  reachable PARITY-AUTOMATON **on** $\{Q\}$.

As a side effect of this operation, the specification becomes monomorphic, i.e., it has only one model up to isomorphism (the model depicted in the figure above). Observe that the trivial model consisting of just one state



is not a model of REACHABLE-PARITY-AUTOMATON because (1) the sub-specification BOOL is interpreted initially (giving $T \neq F$), (2) the semantics of **enrich** requires that the BOOL-reduct of any model of REACHABLE-PARITY-AUTOMATON be a model of the specification BOOL, and (3) the equations for "is-final" imply that if $T \neq F$ then $q_0 \neq q_1$.

There are many other (reachable) finite state automata which accept even-parity strings but which are not models of REACHABLE-PARITY-AUTOMATON. Here is one, which has additional states to detect consecutive pairs of 1's. In the state q_i^j, i indicates the parity of the string seen so far, and j indicates the number of consecutive 1's seen so far.



We can include such automata in the models of REACHABLE-PARITY-AUTOMATON if we specify that the behaviour of interest is the recognition of even-parity strings; the particular set of states or the transition function is not significant. To do this, we first have to extend the specification REACHABLE-PARITY-AUTOMATON to define the effect on strings. Note that in the specification below, although two copies of BIT are imported (one via REACHABLE-PARITY-AUTOMATON, and one via BIT-STRING), they are coalesced into one because of the way the **enrich** operation is defined.

spec REACHABLE-PARITY-AUTOMATON-1 =
enrich REACHABLE-PARITY-AUTOMATON, BIT-STRING
operations
$\hat{\delta}$: Q, BIT-STRING $\to Q$ /* extension of δ to strings */
evenp : BIT-STRING $\to$ BOOL /* effective predicate realized by automaton */
axioms
$\hat{\delta}(q, \epsilon) = q$
$\hat{\delta}(q, a\omega) = \hat{\delta}(\delta(q, a), \omega)$

$\text{evenp}(\omega) = \text{is-final}(\hat{\delta}(\text{init}, \omega))$
end

Next, we hide the two specific states, q_0 and q_1, which we have used to define parity in the specification PARITY-AUTOMATON.

spec REACHABLE-PARITY-AUTOMATON-2 =
derive from REACHABLE-PARITY-AUTOMATON-1 **by**
(Sig[REACHABLE-PARITY-AUTOMATON-1] $- \{q_0, q_1\}$)
$\hookrightarrow$ Sig[REACHABLE-PARITY-AUTOMATON-1]
end

Now, we can use the **abstract** specification-building operation to specify that the relevant part of the specification is the predicate for determining even parity.

spec ABSTRACT-PARITY-AUTOMATON =
abstract REACHABLE-PARITY-AUTOMATON-2 **wrt**
$\text{evenp}(\omega) = b$
end

It appears that with ABSTRACT-PARITY-AUTOMATON we have abstractly characterized a finite state automaton for accepting even-parity strings. However, there are several side effects of the **abstract** operation. The reachability of REACHABLE-PARITY-AUTOMATON-2 is not preserved. We can rectify this by using

spec REACHABLE-ABSTRACT-PARITY-AUTOMATON =
reachable ABSTRACT-PARITY-AUTOMATON **on** $\{Q\}$.

This specification is still not suitable because the initiality of the sub-specifications BOOL, BIT, and BIT-STRING, is not preserved. For the specifications BOOL and BIT, we can use an alternative scheme (rather than initiality) to restrict the models:

spec REACHABLE-ABSTRACT-PARITY-AUTOMATON′ =
reachable
abstract REACHABLE-PARITY-AUTOMATON-2 **wrt**
$\text{evenp}(\omega) = b$
$\text{T} \neq \text{F}$
$0 \neq 1$
end
on $\{Q$, BOOL, BIT$\}$
end

Restricting to the initial model of BIT-STRING is not as simple. One way is to allow higher-order formulas, e.g., initiality, as observations in the **abstract** operation; but this would mean redefining observational equivalence. A second choice is to use initial restrictions (see definition 4.40). A third choice is add enough axioms to the specification BIT-STRING to make it monomorphic, as shown below.

spec BIT-STRING$'$ =
enrich BIT, BIT-STRING
sorts NAT
operations

$0 : \quad \longrightarrow \text{NAT}$

$\text{succ} : \text{NAT} \longrightarrow \text{NAT}$

$\text{length} : \text{BIT-STRING} \longrightarrow \text{NAT}$

axioms

$0 \neq \text{succ}(x)$

$(\text{succ}(x) = \text{succ}(y)) \Rightarrow x = y$ /* the successor function is injective */

$\text{length}(\epsilon) = 0$

$\text{length}(a\omega) = \text{succ}(\text{length}(\omega))$

$\text{length}(x) \neq \text{length}(y) \Rightarrow x \neq y$

$\text{length}(ax) = \text{length}(by) \wedge a \neq b \Rightarrow ax \neq by$

$\text{length}(ax) = \text{length}(by) \wedge a = b \wedge x \neq y \Rightarrow ax \neq by$

end

We can ensure initial interpretation for bit-strings by substituting BIT-STRING$'$ for the specification BIT-STRING in the construction of REACHABLE-PARITY-AUTOMATON-2; let the specification so obtained be REACHABLE-PARITY-AUTOMATON-2$'$. The **abstract** operation also has to be modified by adding the axioms of BIT-STRING$'$ as observations.

spec REACHABLE-ABSTRACT-PARITY-AUTOMATON$''$ =
reachable
abstract REACHABLE-PARITY-AUTOMATON-2$'$ **wrt**

$\text{evenp}(\omega) = b$

$\text{T} \neq \text{F}$

$0 \neq 1$

$Axioms[\text{BIT-STRING}']$

end
on $\{Q, \text{BOOL}, \text{BIT}, \text{NAT}, \text{BIT-STRING}\}$
end

Finally, we note that the specification REACHABLE-ABSTRACT-PARITY-AUTOMATON$''$ does not have any initial model. This is so because the automaton can perform an arbitrary computation on the side as long as it recognizes even-parity strings. Thus there is no "largest" automaton which can serve as the initial model. However, we can retrieve the original two-state automaton with which we started as the terminal model of the specification REACHABLE-ABSTRACT-PARITY-AUTOMATON$''$:

spec MINIMAL-PARITY-AUTOMATON =
final REACHABLE-ABSTRACT-PARITY-AUTOMATON$''$.

4.2.2 Remarks on structuring operations

We can classify the building operations discussed in this section into several kinds:

1. atomic: build basic specifications;
2. incremental building: union, sum, various flavours of enrichment;
3. parametric building: lambda-expressions;
4. name management: translate, derive; and
5. model management: iso close, abstract, minimal.

This list does not contain operations for obtaining initial semantics or final semantics for specifications. Sannella and Tarlecki [Sannella and Tarlecki 88a] explain this absence as a matter of taste, and technically because initiality is not monotonic with respect to inclusion of model classes (all the building operations listed above are monotonic). Monotonicity is desirable if we want to build specifications recursively: monotonic operations guarantee the existence of fixpoints. However, we believe that initial and final semantics are powerful, simple, and elegant ways of specifying many data structures such as strings, trees, and sets. Thus, they should be included as basic specification building operations.

4.3 Parameterization

Most data types in computer science are insensitive to the type of the basic components out of which they are built. For example, we can have queues of integers, queues of communication packets (in a distributed program), queues of read/write requests (in a disk driver program), etc. All such variations can be uniformly described by considering "queue" to be a data type *constructor*, which when given an arbitrary type ANY, produces the type QUEUE-OF-ANY, the type of queues built out of basic components of type ANY. Observe the analogy with functions, e.g., the function $\sqrt{x}$, which takes any non-negative real number x and produces its square-root. Exploiting this analogy, Sannella and Tarlecki [Sannella and Tarlecki 88a] define a *parameterized specification* to be a lambda-expression, $\lambda X\colon \Sigma_{\mathrm{par}} \cdot SP_{\mathrm{res}}$, where SP_{res} is a specification building operation producing Σ_{res}-specifications from Σ_{par}-specifications. When this expression is applied to a concrete specification SP with signature Σ_{par}, the result is obtained by substituting the variable X by the concrete parameter SP in the specification building operation SP_{res}:

$$
\begin{aligned}
Sig[(\lambda X\colon \Sigma_{\mathrm{par}} \cdot SP_{\mathrm{res}})(SP)] &= \Sigma_{\mathrm{res}} \\
Mod[(\lambda X\colon \Sigma_{\mathrm{par}} \cdot SP_{\mathrm{res}})(SP)] &= Mod[SP_{\mathrm{res}}(X/SP)]
\end{aligned}
$$

This scheme for parameterization, although very general, is also quite weak. Most notions of parameterization in the literature have facilities for imposing additional conditions

on the parameter, e.g., attached to the parameter might be a specification SP_{par}. A concrete specification SP is an instance of the parameter only if it satisfies the specification SP_{par}, usually via a specification morphism $\sigma: SP_{\text{par}} \to SP$. The kernel language ASL [Wirsing 86] (see also section 5.2) incorporates this feature. An example where the parameter has additional conditions is that of sorting: to sort a collection of entities, there should be an ordering relation defined on the entities.

Another difficulty with the scheme above is that the signature of the resulting specification is constant (Σ_{res}). In general, we would like the signature of the resulting specification to vary depending on the signature of the actual parameter. The schemes for parameterization described below have this facility.

There have been an number of proposals for parameterized specifications in the literature which fit the general mould of the definition above. The specification building operations used for parameterization vary from shallow transformations on syntax to deep transformations involving models. These variations are described below.

Macro substitution

The simplest scheme for parameterizing a specification is to designate parts of the specification as "variable". The specification then becomes a macro which can be instantiated to obtain a specialized specification. Observe that this is a purely syntactic transformation.

EXAMPLE 4.19. As an example, we define a macro for queues. Compare this with the unparameterized case in example 2.43.

spec QUEUE =
  parameter
  sorts
    DATA   /* specifies that DATA is a parameter variable */
  end
sorts QUEUE
operations
  emptyq : $\to$ QUEUE
  putq : DATA, QUEUE $\to$ QUEUE   /* the variable DATA is used here */
  getq : QUEUE $\to$ DATA   /* ... and here */
  popq : QUEUE $\to$ QUEUE
axioms
  $\neg D(\text{getq}(\text{emptyq}))$
  $\neg D(\text{popq}(\text{emptyq}))$
  getq(putq(x, q)) = **if** q = emptyq **then** x **else** getq(q)
  popq(putq(x, q)) = **if** q = emptyq **then** emptyq **else** popq(q)
end

□

The specification above has the sort DATA as a parameter. The specification can be instantiated by substituting a concrete sort name for DATA and including the specification corresponding to that type. Variables, such as DATA above, are called *scheme variables* and specifications containing them are called specification schemes [Bauer et al. 85]. It should

be noted that a specification containing scheme variables does not have the semantics of a normal specification. Its semantics is only defined after the scheme variables contained in it are instantiated.

Pushouts

Sometimes it is necessary that the parameter in a specification satisfy some additional constraints. For example, if we are defining a (parameterized) priority queue, with the priority values obtained from a parameter type, then the parameter should provide an ordering relation on the priority values. Such constraints can be stated by adding operations and axioms to the parameter part. The instantiation of such additionally constrained parameters can be formalized using the notion of a pushout from category theory [Ehrig and Mahr 85; Ehrig et al. 81; Ehrig et al. 80c; Ehrich 82].

DEFINITION 4.20: *Sub-specification.* A specification S is said to be a sub-specification of a specification T if all the sorts, operations, and axioms of S are contained in T. □

DEFINITION 4.21: *Parameterized specification.* A parameterization specification is a pair of specifications $\langle P, SP\rangle$ consisting of a *formal parameter* specification P and a *target* specification SP, such that P is a sub-specification of SP. □

In other words, a parameterized specification is a normal specification with a distinguished parameter part. A parameterized specification $\langle P, SP\rangle$ can be represented by an inclusion arrow $i: P \hookrightarrow SP$ in the category **Spec**. More generally, this arrow need only be an injection, or a monomorphism, $m: P \rightarrowtail SP$.

Now, consider a parameter specification P. What does it mean for a specification P' to be an instance of the parameter P? First, there should be a sort and an operation in P' for each sort and operation (respectively) in P. Second, the specification P' should satisfy the axioms in the parameter P. These two conditions can be achieved by giving a specification morphism $v: P \rightarrow P'$. Such a specification morphism is called a *view* [Burstall and Goguen 77]. The effect of instantiating the parameter P by a specification P' can then be described by the following pushout in the category **Spec**.

$$\begin{array}{ccc} P & \xrightarrow{i} & SP \\ {\scriptstyle v}\downarrow & & \downarrow{\scriptstyle v'} \\ P' & \xrightarrow[i']{} & SP' \end{array}$$

The pushout determines a new specification SP' built from SP with the components of the parameter P substituted by those of P' (as specified by the view v, and appropriately renaming entities to avoid name clashes). The result SP' also contains all the axioms in the actual parameter P' which are not in P.

EXAMPLE 4.22. We illustrate the instantiation of a parameterized specification using the example of a priority queue. A priority queue is a specialization of an ordinary queue in which each element has a priority value attached to it. Priority values are ordered: this is imposed as a condition on the parameter. When an element is retrieved from the queue,

rather than returning the oldest element, the element with the highest priority is returned (when priorities are equal, the older element is returned).

```
parameter TOTAL-ORDER =
based on BOOL
sorts DATA
operations
  _ ≥ _ : DATA, DATA → BOOL
axioms
      /* the relation is total */
  D(x ≥ y)
      /* reflexive */
  x ≥ x
      /* anti-symmetric */
  (x ≥ y) ∧ (y ≥ x) ⇒ (x = y)
      /* transitive */
  (x ≥ y) ∧ (y ≥ z) ⇒ (x ≥ z)
end
```

```
spec PRIORITY-QUEUE =
parameter TOTAL-ORDER
sorts QUEUE
operations
  emptyq :                  → QUEUE
     putq : DATA, QUEUE → QUEUE
     getq :       QUEUE → DATA
     popq :       QUEUE → QUEUE
axioms
  ¬D(getq(emptyq))
  ¬D(popq(emptyq))
  getq(putq(x, q)) = if q = emptyq then x
     elseif m ≥ x then m else x
       where m = getq(q)
  popq(putq(x, q)) = if q = emptyq then emptyq
     elseif m ≥ x then putq(x, popq(q)) else q
       where m = getq(q)
end
```

We instantiate the parameterized specification PRIORITY-QUEUE using pairs of natural numbers. In every pair, the first element is the data and the second is the priority. The ordering on the pairs is the ordering on the second component of the pairs.

```
spec NAT-PAIR =
based on NAT
sorts NAT-PAIR
operations
     ⟨_, _⟩ : NAT, NAT → NAT-PAIR
       first : NAT-PAIR → NAT
    second : NAT-PAIR → NAT
axioms
  first(⟨a, b⟩) = a
  second(⟨a, b⟩) = b
  ⟨first(p), second(p)⟩ = p
end
```

```
spec NAT-PAIR-WITH-ORDER =
based on BOOL, NAT-PAIR
operations
  _ ≥ _ : NAT-PAIR, NAT-PAIR → BOOL
axioms
  ⟨a, b⟩ ≥ ⟨c, d⟩ ⇔ ∃n · b = d + n
end
```

Now we can match up the parameter TOTAL-ORDER used in PRIORITY-QUEUE with the specification NAT-PAIR-WITH-ORDER defined above to give the following view:

$$\begin{array}{ccc} \text{DATA} & \mapsto & \text{NAT-PAIR} \\ \geq & \mapsto & \geq \end{array}$$

Strictly speaking, this signature morphism should be accompanied by a proof that the axioms of TOTAL-ORDER are satisfied by NAT-PAIR-WITH-ORDER under this translation. We omit the proof and assume that the signature morphism is also a specification morphism.

The effect of the pushout used to instantiate the parameter

$$\begin{array}{ccc} \text{TOTAL-ORDER} & \longrightarrow & \text{PRIORITY-QUEUE} \\ \downarrow & & \downarrow \\ \text{NAT-PAIR-WITH-ORDER} & \longrightarrow & \text{PRIORITY-QUEUE-OF-NAT-PAIR} \end{array}$$

is the following specification of a priority queue which has pairs of natural numbers as elements.

spec PRIORITY-QUEUE-OF-NAT-PAIR =
based on NAT, BOOL
sorts QUEUE
operations

$\langle _, _ \rangle$:	NAT, NAT → NAT-PAIR	/* operations on pairs */
first :	NAT-PAIR → NAT	
second :	NAT-PAIR → NAT	
$_ \geq _$:	NAT-PAIR, NAT-PAIR → BOOL	/* the ordering relation */
emptyq :	→ QUEUE	/* operations on queues */
putq :	NAT-PAIR, QUEUE → QUEUE	
getq :	QUEUE → NAT-PAIR	
popq :	QUEUE → QUEUE	

axioms

/* axioms for pairs */

$\text{first}(\langle a, b\rangle) = a$

$\text{second}(\langle a, b\rangle) = b$

$\langle \text{first}(p), \text{second}(p)\rangle = p$

/* axioms for the ordering relation */

$\langle a, b\rangle \geq \langle c, d\rangle \Leftrightarrow \exists n \cdot b = d + n$

/* axioms for queues */

$\neg D(\text{getq}(\text{emptyq}))$

$\neg D(\text{popq}(\text{emptyq}))$

$\text{getq}(\text{putq}(x, q)) =$ **if** $q = \text{emptyq}$ **then** x **elseif** $m \geq x$ **then** m **else** x
where $m = \text{getq}(q)$

$\text{popq}(\text{putq}(x, q)) =$ **if** $q = \text{emptyq}$ **then** emptyq
elseif $m \geq x$ **then** $\text{putq}(x, \text{popq}(q))$ **else** q
where $m = \text{getq}(q)$

end

□

Colimits and sharing

The pushout approach to instantiating parameters breaks down when there are shared sub-specifications between the formal parameter, the target, and the concrete parameter. Let us

consider a modification of the previous example of parameterization: instead of a priority queue, a parameterized version of the bounded queue of example 2.44.

```
parameter TRIV =      spec BOUNDED-QUEUE =
sorts                 parameter TRIV
  DATA                based on NAT
end                   sorts QUEUE
                      operations
                        emptyq :          NAT → QUEUE
                          putq : DATA, QUEUE → QUEUE
                          getq :       QUEUE → DATA
                          popq :       QUEUE → QUEUE
                        length :       QUEUE → NAT
                         bound :       QUEUE → NAT
                      axioms
                            /* axioms omitted */
                      end
```

Observe that this specification imports the specification NAT. Let us instantiate the parameter with NAT-PAIR, using the view DATA $\mapsto$ NAT-PAIR; the intent is to obtain a bounded queue with elements as pairs of natural numbers. We have to complete the following pushout to obtain the result:

$$\begin{array}{ccc} \text{TRIV} & \longrightarrow & \text{BOUNDED-QUEUE} \\ \downarrow & & \\ \text{NAT-PAIR} & & \end{array}$$

Unfortunately, the result contains two (disjoint) copies of the specification NAT, one from the parameter, and one from the target, since a pushout automatically renames components to avoid name clashes. We can remove this anomaly by specifying that the two copies of NAT which are imported into BOUNDED-QUEUE and NAT-PAIR are in fact the same. This leads to the concept of a based specification.

Based theories were first introduced by Burstall and Goguen [Burstall and Goguen 79] to define the semantics of the language Clear. A based theory is a theory together with an environment containing all the theories on which it is based. An environment is a collection of theories and theory morphisms which indicate the sharing relationships between the various theories.

We will now use a generalization of pushouts, colimits, to describe the combination of theories and specifications which have shared sub-parts. We will use based specifications rather than based theories (the difference is that the collection of axioms in a theory is closed under logical consequence, definition 4.8). Working with specifications instead of theories is justified because colimits of specifications can be transferred to colimits of theories [Goguen and Burstall 84a].

DEFINITION 4.23: *Diagram.* A diagram in a category $\mathcal{C}$ is a collection of $\mathcal{C}$-objects and a collection of $\mathcal{C}$-arrows between these objects. □

(co-cone)

DEFINITION 4.24: *Cone.* Given a diagram D in a category $\mathcal{C}$ and a $\mathcal{C}$-object c, a cone[25] *from the base D to the vertex c* is a collection of $\mathcal{C}$-arrows $\{ f_i : d_i \to c \mid d_i \in D \}$, one for each object d_i in the diagram D, such that for any arrow $g : d_i \to d_j$ in D, the following triangle commutes

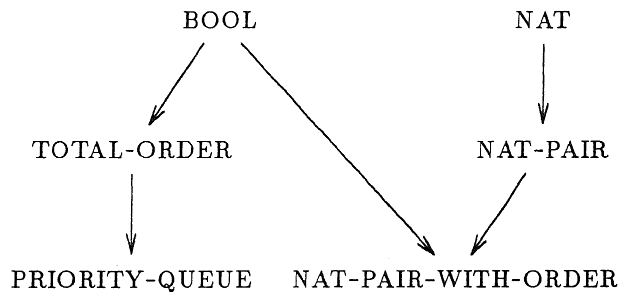
$$\begin{array}{ccc} d_i & \xrightarrow{g} & d_j \\ & f_i \searrow \; \swarrow f_j & \\ & c & \end{array}$$

i.e., we have $f_j \circ g = f_i$. □

DEFINITION 4.25: *Based specification.* A based specification is a cone in the category **Spec** of specifications and specification morphisms. □

Specifically, if $\langle B, S \rangle$ is a cone from the base B to the vertex T in the category **Spec**, then the specification S is said to be based upon the specifications B_i in the base B. The arrows in the base show the sharing relationships between the base specifications.

Here are examples of based specifications: the specifications of example 4.22 defining priority queues. The arrows indicate inclusion of specifications.



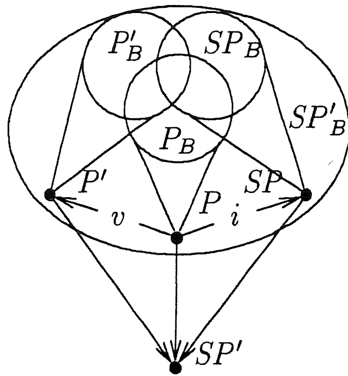
Using the concept of based specifications, we can account for the sharing of specifications. A parameterized specification is, as before, a morphism from the parameter specification to the target specification; this time, however, both are based specifications, and their bases may intersect, i.e., share certain sub-specifications. Similar considerations apply to the parameter specification and its view into the concrete parameter instance. All these specifications, together with the morphisms, form a diagram in the category **Spec**. The result of instantiating the parameter is obtained as the *colimit* of such a diagram, as defined below.

[25] We follow MacLane [Mac Lane 71] in the terminology here. Some authors use the name "co-cone".

DEFINITION 4.26: *Colimit.* A colimit for a diagram D in a category $\mathcal{C}$ is a $\mathcal{C}$-object c along with a cone $\{ f_i : d_i \to c \mid d_i \in D \}$ from D to c such that for any other cone $\{ f_i' : d_i \to c' \mid d_i \in D \}$ from D to a vertex c', there is a unique $\mathcal{C}$-arrow $f : c \to c'$ such that for every object d_i in D, the following triangle commutes

$$\begin{array}{ccc} & d_i & \\ f_i \swarrow & & \searrow f_i' \\ c & \underset{f}{\longrightarrow} & c' \end{array}$$

i.e., $f \circ f_i = f_i'$. □



P is the formal parameter with base P_B
SP is the target specification with base SP_B
$i : P \hookrightarrow SP$ is the inclusion of P into SP
P' is the actual parameter with base P'_B
$v : P \to P'$ is the view from P to P'
SP'_B is the base formed by
the based specifications P, SP, and P'
along with the morphisms i and v
SP' is the colimit object of SP'_B

Another, equivalent, approach is to consider parameterization in the category of based theories and based theory morphisms (as in [Burstall and Goguen 79]). The colimit is then just a pushout.

Free functors and initial semantics

Until now, we have considered parameterization at the syntactic level. How do we define the semantics of a parameterized specification and its instantiation? We will consider initial semantics here, and final and loose semantics in the next two sections.

Let us consider again the parameterized specification QUEUE of example 4.19. Taking the initial semantics of the target specification is clearly inadequate; the parameter sort DATA has no constructors and hence, the initial model is trivial. This will be true in general, because we cannot determine the constructors of a parameter sort until it is instantiated. Thus, the appropriate semantics for a parameterized specification is some procedure which translates *any* model of the parameter specification into an initial model of the (instantiated) target specification. This is formalized using the concept of a *free* functor. The (initial) semantics of parameterized specifications was first proposed in [Thatcher et al. 82, 1978 version], and later followed up in [Ehrig et al. 80c; Ehrig et al. 81]. The approach is described in detail in [Ehrig and Mahr 85, Chapters 7 and 8].

Let $\langle P, SP \rangle$ be a parameterized specification. There is an obvious "forgetful" functor U from the category of models of SP to the category of models of P which maps a model of SP into a model of P by just forgetting the extra sorts, operations and axioms in SP. If $i : P \to SP$ is the inclusion specification morphism, then the forgetful functor $U : Mod[SP] \to Mod[P]$ is just the reduct functor $_|_i$ present in the underlying institution.

First, let us consider a specific model of the parameter, say $M \in Mod[P]$. We would like to extend this to a model N of the target specification SP, in some "canonical" fashion. The concept of a canonical extension is captured in the following definition from category theory.

DEFINITION 4.27: *Universal arrow.* Given a functor $G: \mathcal{D} \to \mathcal{C}$ and a $\mathcal{C}$-object c, a universal arrow from c to G is a pair $\langle r, u\rangle$ consisting of a $\mathcal{D}$-object r and a $\mathcal{C}$-arrow $u: c \to G(r)$ such that for every pair $\langle d, f\rangle$ of a $\mathcal{D}$-object d and a $\mathcal{C}$-arrow $f: c \to G(d)$, there is a unique $\mathcal{D}$-arrow f' such that the following diagram commutes

$$\begin{array}{ccccc} & u & G(r) & & r \\ c & & \downarrow G(f') & & \downarrow f' \\ & f & G(d) & & d \end{array}$$

i.e., $G(f') \circ u = f$. □

In more plain terms, this definition says that the object $c \in |\mathcal{C}|$ is extended in a canonical fashion to an object $r \in |\mathcal{D}|$ (with respect to the functor G). The extension is canonical in the sense that every other extension (say $d \in |\mathcal{D}|$) can be uniquely retrieved from the canonical extension. The arrow $u: c \to G(r)$ provided by the definition above indicates the relationship between the object c and its "image" in the extended object r.

For the case of the parameterized specification $i: P \to SP$ above, we can extend a model M of P to a model N of SP by finding a universal arrow $\langle N, f: M \to N|_i\rangle$ from M to $U: Mod[SP] \to Mod[P]$.[26]

[26] N is also called the free construction over M with respect to U [Ehrig and Mahr 85].

EXAMPLE 4.28. We will show what the extended model looks like for the case of the parameterized specification QUEUE of example 4.19. Let us instantiate the parameterized specification with the concrete parameter NAT. Intuitively, we expect the instantiated specification to be the type of queues of natural numbers.

In the initial model of the specification NAT, the carrier NAT consists of

$$\{0, \text{succ}(0), \text{succ}(\text{succ}(0)), \text{succ}(\text{succ}(\text{succ}(0))), \ldots\}.$$

Let us denote this more conveniently by $\{0, 1, 2, 3, \ldots\}$. Let us also abbreviate the constructors for queues as follows:

$$\begin{array}{ll} | & \text{for emptyq, and} \\ x \triangleright q & \text{for putq}(x, q)\text{, with association to the right.} \end{array}$$

Instantiating the queue specification with the initial model of NAT, we obtain a carrier for QUEUE which consists of

$$\left\{ \begin{array}{l} |, 0 \triangleright |, 1 \triangleright |, 2 \triangleright |, \ldots \\ 0 \triangleright 0 \triangleright |, 0 \triangleright 1 \triangleright |, 3 \triangleright 4 \triangleright |, \ldots \\ 0 \triangleright 0 \triangleright 0 \triangleright |, 0 \triangleright 1 \triangleright 1 \triangleright |, 3 \triangleright 4 \triangleright 2 \triangleright |, \ldots \\ \ddots \end{array} \right\}$$

Observe that this is just like the construction of the initial algebra, except that the elements in the carrier DATA are provided by a parameter. The resultant algebra is called a *free algebra*. □

To provide the initial semantics for a parameterized specification, we require an assignment of a free algebra (i.e., a universal arrow) to each model M of the parameter. Moreover, this assignment should be compatible with translations of M.

DEFINITION 4.29: *Free functor.* A functor $F: \mathcal{C} \to \mathcal{D}$ is said to be free[27] with respect to a functor $G: \mathcal{D} \to \mathcal{C}$ if to each $\mathcal{C}$-object c there is a universal arrow $\eta_c: c \to G(F(c))$ such that the following diagram commutes for every $\mathcal{C}$-arrow $f: c \to c'$

$$\begin{array}{ccccc} c & \xrightarrow{\eta_c} & G(F(c)) & & F(c) \\ f \downarrow & & \downarrow G(F(f)) & & \downarrow F(f) \\ c' & \xrightarrow[\eta_{c'}]{} & G(F(c')) & & F(c') \end{array}$$

i.e., we have $G(F(f)) \circ \eta_c = \eta_{c'} \circ f$. □

[27]This definition is equivalent to providing an adjunction between the categories $\mathcal{C}$ and $\mathcal{D}$.

We can now define the semantics of a parameterized specification $\langle P, SP\rangle$ to be the (isomorphism class of the) free functor $F: P \rightarrow SP$ which is left adjoint to the forgetful functor $U: SP \rightarrow P$.

How does this semantics relate to the instantiation of parameters using pushouts? If $i: P \rightarrow SP$ is a parameterized specification, M is the initial model of the concrete parameter specification P', and N is the initial model of the instantiated specification SP', then the parameter instantiation described by the following diagram

$$\begin{array}{ccc} P & \xrightarrow{i} & SP \\ v\downarrow & & \downarrow v' \\ P' & \xrightarrow[i']{} & SP' \end{array}$$

is said to be correct if

1. (actual parameter protection) $N|_{i'} = M$, i.e., N is a conservative extension of M; and
2. (passing compatibility) $N|_{v'} = F(M|_v)$, i.e., the syntactic substitution of P' for P in SP is compatible with the semantics of $\langle P, SP\rangle$.

Persistency

The correctness of parameter instantiation using pushouts can be guaranteed if the free functor F which is the semantics of the parameterized specification is *persistent*, a notion which we now define. Informally, persistency means that an argument to the functor is preserved ("persists") in the result, i.e., (a) no new elements are added to the parameter sorts, and (b) no two elements of any parameter sort are identified. Persistency can be thought of as a generalization of the two slogans of initial semantics, no junk and no confusion (see also the discussion after definition 4.13).

DEFINITION 4.30: *Persistency.* Given a forgetful functor $G: \mathcal{D} \rightarrow \mathcal{C}$, a free functor $F: \mathcal{C} \rightarrow \mathcal{D}$ is said to be persistent if the universal arrows from which it arises $\eta_c: c \rightarrow G(F(c))$ are all isomorphisms. Moreover, if all the arrows η_c are identities, then F is said to be *strongly* persistent. □

For a more general definition not restricted to free functors, see [Ehrig and Mahr 85, p. 220]. An extensive discussion of sufficient conditions to ensure persistency can be found in [Padawitz 87].

We will informally show here how (strong) persistency implies the correctness of parameter instantiation using pushouts. For a formal proof in the equational institution with initial semantics, see [Ehrig and Mahr 85]. By the assumption that the model functor **Mod** translates colimits to limits, a pushout of signatures is translated into a pullback of model categories in **Cat**. An immediate corollary is the following lemma.

Amalgamation Lemma. Given a pushout in the category **Sig** of signatures,

$$\begin{array}{ccc} \Sigma & \xrightarrow{\sigma_1} & \Sigma_1 \\ \sigma_2 \downarrow & & \downarrow \sigma_2' \\ \Sigma_2 & \xrightarrow[\sigma_1']{} & \Sigma' \end{array}$$

for any Σ_1-algebra A_1 and Σ_2-algebra A_2 such that $A_1|_{\sigma_1} = A_2|_{\sigma_2}$, there exists a unique Σ'-algebra A' such that $A'|_{\sigma_1'} = A_2$ and $A'|_{\sigma_2'} = A_1$. The algebra A' is called the amalgamation of A_1 and A_2 along σ_1 and σ_2. □

Using this lemma, we can extend a strongly persistent free functor $F: P \to SP$ of the parameterized specification $\langle P, SP \rangle$ to a strongly persistent free functor $F': P' \to SP'$ (mapping a model M of P' to the amalgamation of $M|_{\sigma_2}$ and $F(M|_{\sigma_2})$). The strong persistency of F' means that the actual parameter is protected (condition 1 above). From the amalgamation lemma and the fact that left adjoint functors preserve colimits [Mac Lane 71, page 115], passing compatibility (condition 2 above) is guaranteed.

Final semantics

The free functor $F: P \to SP$ assigns to each parameter algebra A the free algebra $F(A)$. The freeness of $F(A)$ (see definitions 4.27 and 4.29) is just a restatement of the fact that the pair $\langle F(A), \eta_A: A \to U(F(A)) \rangle$ is initial in the category of all such pairs. Thus freeness is a generalization of initiality.

Dually, we can define a "co-free" functor $F^\circ: P \to SP$ which assigns to each parameter algebra A, a pair $\langle F^\circ(A), \xi_A: A \to U(F^\circ(A)) \rangle$ which is terminal in the category of all such pairs. Thus we have a generalization of final semantics, and a co-free functor can be considered to be the final algebra semantics for a parameterized specification.

This approach to parameterization is described in [Ganzinger 83] and [Hornung and Raulefs 81]. Just as in final semantics (definition 3.11), the algebra $F^\circ(A)$ can be constructed using a congruence which identifies all those elements which behave the same way in all contexts. Corresponding to the no junk and no confusion conditions, we have the notions of "consistent" (ξ_A is injective) and "sufficiently complete" (ξ_A is surjective). When both conditions are satisfied, the functor F° is said to be (terminal) persistent.

Note that a co-free functor is not quite a right adjoint (to the forgetful functor). Goguen and Meseguer [Goguen 73; Goguen and Meseguer 82] use a right adjoint functor which produces models which are "minimal" realizations of some behaviour.

Loose semantics

The simplest way to define loose semantics for parameterization is by using syntactic substitution. This is the approach followed in CIP-L (see the discussion of macro-substitution at the beginning of this section).

Another way is to follow the λ-calculus approach, as in [Sannella and Tarlecki 88a] and ASL [Wirsing 86]. A parameterized specification is a lambda-expression, $\lambda X: \Sigma_{\text{par}} \cdot SP_{\text{res}}$. The semantics is again defined to be the class of models of the result specification, obtained by

substitution. This is more than textual substitution because SP_{res} can use any specification-building operation.

An interesting generalization of the initial and loose approaches is *stratified* loose semantics, where the semantics of a parameterized specification $\langle P, SP \rangle$ is considered to be the class of all persistent functors from $Mod[P]$ to $Mod[SP]$ (see definition 4.44).

Parameter passing

Parameter passing is the instantiation of the formal parameter in a parameterized specification. The various techniques we have seen—macro substitution, pushouts, free functors—are classified by Wirsing and Broy [Wirsing and Broy 82] as shown below. The resultant signature is the same in all the cases, that obtained by a pushout. The set of axioms will, in general, differ. The axioms associated with the result are as follows (A is the argument specification, and R is the result specification obtained by a pushout, $Gen[R]$ is the class of (finitely) term-generated models of R, φ is any $Sig[R]$-formula):

call-by-specification: R;

call-by-theory: $\{ \varphi \mid \forall M \in Mod[R] \cdot M \models \varphi \}$;

call-by-type: $\{ \varphi \mid \forall M \in Gen[R] \cdot M \models \varphi \}$;

call-by-algebra: $\{ \varphi \mid \forall M \in Mod[R] \cdot M \models \varphi$ and φ is a closed formula $\}$.

Call-by-specification is the same as a pushout in the category **Spec** of specifications and specification morphisms. Call-by-theory is a pushout in the category **Theory** of theories and theory morphisms. Call-by-type differs from call-by-theory in that the result specification contains all those sentences which are provable by structural induction. Call-by-algebra is the same as the semantics obtained by a free functor.

When the concrete parameter is itself another parameterized specification, then the parameter passing mechanism has to be slightly modified. This case is called *parameterized* parameter passing (or general parameter passing) by Ehrig et al. [Ehrig et al. 81]. The result of instantiating a parameterized specification $i_1: P_1 \to SP_1$ by a parameterized specification $i_2: P_2 \to SP_2$ via a view $h: P_1 \to SP_2$ is another parameterized specification $i' \circ i_2: P_2 \to SP'$ obtained by a pushout as in the following diagram:

$$\begin{array}{ccccc} & & P_1 & \xrightarrow{i_1} & SP_1 \\ & & h\downarrow & & \downarrow h' \\ P_2 & \xrightarrow[i_2]{} & SP_2 & \xrightarrow[i']{} & SP' \end{array}$$

It can be shown by diagram chasing that the composition of parameterized specifications (as defined by the pushout above) is associative and compatible with parameter instantiation (see, for example, [Ehrig and Mahr 85, Chapters 7 and 8]).

4.4 Modules

We now discuss an algebraic version of the computer science notion of a module as an information-hiding structure. A module is different from a specification in that a module can include hidden components.[28] This approach was proposed by Ehrig et al. [Blum et al. 87; Ehrig and Mahr 90; Weber and Ehrig 86]. Their concept of a module specification integrates the notions of parameterization, implementation, and information hiding. Module interconnections put together specifications at a somewhat higher-level than the specification building operations of section 4.2.

DEFINITION 4.31: *Module specification.* A module specification consists of four specifications, PAR (parameter), IMP (import interface), EXP (export interface), BOD (body), and specification morphisms e, s, i, v, such that the following diagram commutes

$$\begin{array}{ccc} \text{PAR} & \xrightarrow{e} & \text{EXP} \\ i\downarrow & & \downarrow v \\ \text{IMP} & \xrightarrow[s]{} & \text{BOD} \end{array}$$

i.e., $v \circ e = s \circ i$. □

Intuitively, a module can be thought of as implementing the export types using the import types, and possibly parameterized by a parameter common to both import and export. The body can contain extra types which may be necessary for this implementation, but which are hidden. To define the semantics of modules, we need an auxiliary notion, the restriction of models along a specification morphism.

(For this definition, we will assume a concrete institution such as equational logic. An institution-independent version of restriction can be defined using the **reachable** specification-building operation; see the discussion after definition 4.16).

DEFINITION 4.32: *Restriction.* Given an SP-algebra A and a specification morphism $\sigma: SP' \to SP$, the restriction of A along σ, denoted by $\mathcal{R}_\sigma(A)$, is the intersection of all those SP-subalgebras[29] of A which have the same σ-reduct as A, i.e.,

$$\mathcal{R}_\sigma(A) = \bigcap\{\, B \in Mod[SP] \mid B \subseteq A \text{ and } B|_\sigma = A|_\sigma \,\}.$$

□

Restriction can be extended to a functor $\mathcal{R}_\sigma: Mod[SP] \to Mod[SP]$. The restriction operation is closely related to the **reachable** specification-building operation: $\mathcal{R}_\sigma(A)$ is that subalgebra of A which is reachable from (the carriers of) $A|_\sigma$ using constants and operations of A.

The restriction functor is used to control the visibility of exported components in a module: only those elements which are reachable from the parameter are visible. The semantics of a module is a functor which translates models of IMP into models of EXP.

[28]The notion of module described in this section is different from that proposed by Goguen and Meseguer [Goguen and Meseguer 82]. Their notion of a module is a specification with hidden sorts.

[29]A subalgebra is a collection of subsets of the carrier sets, such that the subsets are closed under the operations of the algebra.

DEFINITION 4.33: *Semantics of modules.* For a module as defined above, the unrestricted and restricted semantics $\mathcal{S}, \mathcal{S}^r : Mod[\text{IMP}] \to Mod[\text{EXP}]$ are given by

$$\mathcal{S} = _|_v \circ \mathcal{F}_s \quad \text{and} \quad \mathcal{S}^r = \mathcal{R}_e \circ \mathcal{S}$$

where

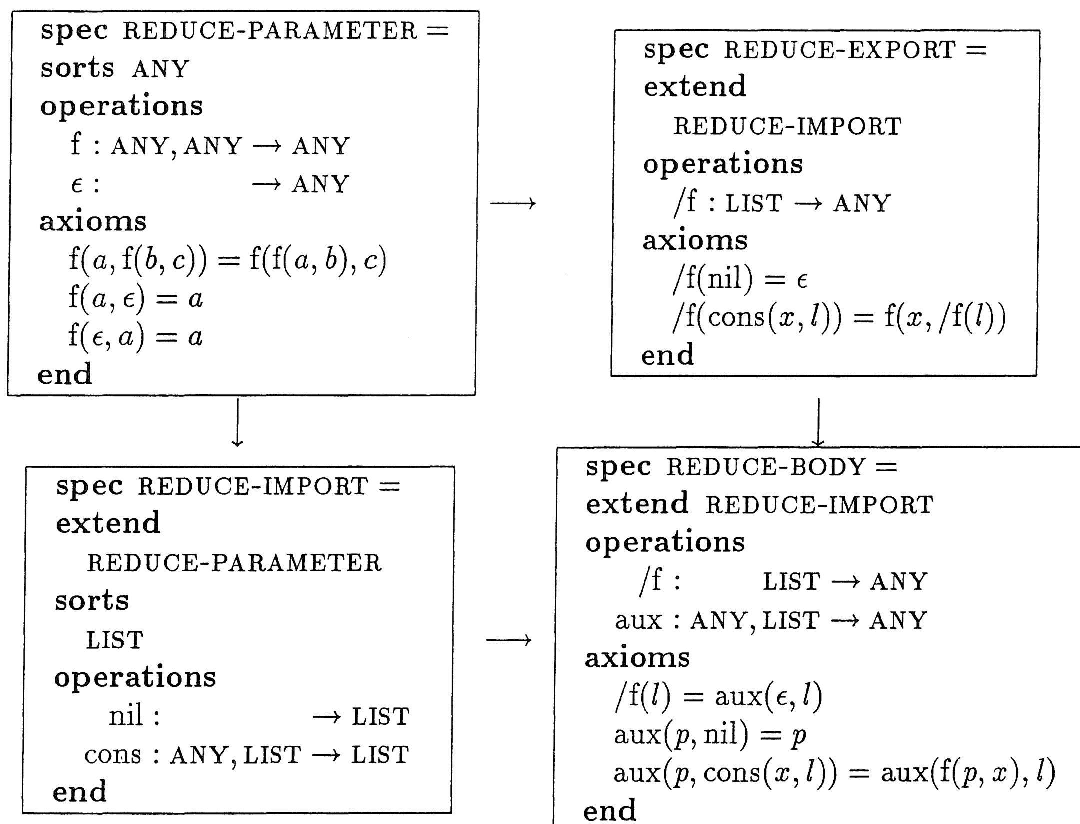
$_|_v : Mod[\text{BOD}] \to Mod[\text{EXP}]$ is the reduct functor,
$\mathcal{F}_s : Mod[\text{IMP}] \to Mod[\text{BOD}]$ is the free functor left adjoint to $_|_s$, and
$\mathcal{R}_e : Mod[\text{EXP}] \to Mod[\text{EXP}]$ is the restriction functor along e. □

The unrestricted semantics is useful for defining the interconnection of modules. The external (visible) semantics of a module is the restricted semantics, $\mathcal{S}^r$.

EXAMPLE 4.34. Here is an example of a module which adds a higher-order operator, "reduce", to the specification of lists. The reduce operator is very similar to that provided in APL and FP [Backus 78]: it takes an associative binary operation as argument, inserts it between the elements of the list, and computes the result. This example illustrates the main features of the module concept: parameterization, implementation, and information hiding. The binary operation is a parameter. The export interface has a recursive definition of the reduce operator. The body contains an iterative version of reduce (called "aux") which is hidden, i.e., not present in the export interface.

We use the **extend** specification building operation rather than **enrich**. The semantics of **extend** is given by a free functor; thus it produces a conservative extension of a specification (see the discussion after definition 4.13).

The specification morphisms in this module are the obvious inclusions of specifications, except EXP $\rightarrow$ BOD, for which the signature morphism is an inclusion but the axioms are different. This signature morphism is indeed a specification morphism because it can be proved (see, for example, [Huet and Lang 78]) that the iterative version of "reduce" implements the recursive version if the binary operation is associative.



□

Module interconnection

There are a number of constructs for putting together modules: composition, actualization, union, product, iteration, etc. We will give here a flavour of these operations by describing

composition. Detailed description of these operations and relationships among them can be found in [Blum et al. 87; Ehrig and Mahr 90; Ehrig et al. 86a; Parisi-Presicce 88].

Modules are composed by matching the import interface of one module with the export interface of another module. Since the parameter part is included in the import and export interfaces, we also have to match the parameters of the two modules.

DEFINITION 4.35: *Module composition.* Given two modules $\mathrm{MOD}_1 = \langle \mathrm{PAR}_1, \mathrm{IMP}_1, \mathrm{EXP}_1, \mathrm{BOD}_1, e_1, s_1, i_1, v_1\rangle$ and $\mathrm{MOD}_2 = \langle \mathrm{PAR}_2, \mathrm{IMP}_2, \mathrm{EXP}_2, \mathrm{BOD}_2, e_2, s_2, i_2, v_2\rangle$, and a pair h of specification morphisms $\langle h_1 \colon \mathrm{PAR}_2 \to \mathrm{PAR}_1, h_2 \colon \mathrm{IMP}_2 \to \mathrm{EXP}_1\rangle$ which match the parameter and import of MOD_2 with the parameter and export of MOD_1 such that $e_1 \circ h_1 = h_2 \circ i_2$, the composition of MOD_2 with MOD_1 via the interface morphism h, written $\mathrm{MOD}_2 \circ_h \mathrm{MOD}_1$, is defined to be the module $\langle \mathrm{PAR}_2, \mathrm{IMP}_1, \mathrm{EXP}_2, \mathrm{BOD}_3, e_2, b_1 \circ s_1, i_1 \circ h_1, b_2 \circ v_2\rangle$ given by the outer rectangle in the diagram below, with BOD_3 obtained as a pushout of $\mathrm{BOD}_1 \xleftarrow{v_1 \circ h_2} \mathrm{IMP}_2 \xrightarrow{s_2} \mathrm{BOD}_2$.

$$
\begin{array}{ccccc}
\mathrm{PAR}_2 & \xrightarrow{\mathrm{id}} & \mathrm{PAR}_2 & \xrightarrow{e_2} & \mathrm{EXP}_2 \\
 & & i_2 \downarrow & & \downarrow v_2 \\
h_1 \downarrow & & \mathrm{IMP}_2 & \xrightarrow{s_2} & \mathrm{BOD}_2 \\
 & & h_2 \downarrow & & \\
\mathrm{PAR}_1 & \xrightarrow{e_1} & \mathrm{EXP}_1 & & \downarrow b_2 \\
i_1 \downarrow & & v_1 \downarrow & & \\
\mathrm{IMP}_1 & \xrightarrow{s_1} & \mathrm{BOD}_1 & \xrightarrow{b_1} & \mathrm{BOD}_3
\end{array}
$$

□

Assuming that the free functors $\mathcal{F}_{s_1}$ and $\mathcal{F}_{s_2}$ in modules MOD_1 and MOD_2 are strongly persistent, and using the fact that strong persistency is preserved by pushouts and composition, the free functor $\mathcal{F}_{b_1 \circ s_1}$ in MOD_3 is also strongly persistent. Under these conditions, the (unrestricted) semantics of MOD_3 is given by

$$\mathcal{S}_3 = \mathcal{S}_2 \circ _|_{h_2} \circ \mathcal{S}_1.$$

Thus, the semantics of the resultant module is compositional. A similar property holds for the restricted semantics under the additional assumption that the restriction functors are conservative (i.e., preserve injectivity of homomorphisms; see [Ehrig and Mahr 90, definition 2.5]).

Specification logics

Most of the work on module specifications uses abstract categorial constructions such as free functors and pushouts. Thus, Ehrig et al. [Ehrig et al. 89] propose a notion called "specification logic" using which one can study the interconnection of modules at an abstract level, independent of the underlying logic. Specification logics are similar to, but weaker (and hence, more general) than institutions.

DEFINITION 4.36: *Specification logic.* A specification logic is a pair $\langle \mathbf{Spec}, \mathit{Mod}{:}\, \mathbf{Spec} \to \mathbf{Cat}^{\mathrm{op}} \rangle$, consisting of a category **Spec** of abstract specifications and specification morphisms, and a (contravariant) functor, *Mod*, assigning a category of models to each specification and a reduct functor to each specification morphism. □

EXAMPLE 4.37. All the institutions described in section 2 are examples of specification logics. □

For a specification logic to be useful for module interconnection, some additional constructions should be possible: free functors, pushouts, amalgamations, etc. Using these, properties of module interconnection, such as distributivity, can be studied at an abstract level. The fruitfulness of this approach is illustrated by the fact that the results obtained by the study of modules with equational logic can be transferred to modules with behavioural specifications (see [Ehrig et al. 89; Orejas et al. 88]) and even to specifications with constraints (section 4.5).

4.5 Constraints

In section 3, we saw how to constrain (or expand) the class of models associated with a specification by adopting different kinds of semantics. This control of semantics applies to the entire specification. In many cases, especially when we build specification in a structured manner, we might want finer control of the semantics: control at the level of sub-specifications. An example of such a situation is the restriction of the interpretation of sub-specifications such as NAT and BOOL to the standard models.

Two kinds of constraints have been proposed in the literature: constraints which require initial semantics for some sub-specifications, and constraints which control how much of the semantics of a specification is preserved when it is imported into a larger specification. We discuss these in the sections below.

Data constraints

A data constraint is a requirement that certain sub-specifications of a specification be interpreted initially or freely. These were proposed by Reichel [Reichel 80; Reichel 87] under the name "initial restrictions" and by Burstall and Goguen [Burstall and Goguen 79] under the name "data constraints" in their work on the specification language Clear. We first give a motivating example for initial restrictions.

EXAMPLE 4.38. We will define the Myhill-Nerode equivalence [Hopcroft and Ullman 79, Theorem 3.9] on the states in a deterministic finite state automaton. Two states are equivalent (behaviourally or observationally equivalent) if there is no sequence of input letters which distinguishes the states with respect to reaching a final state.

In the specification below, we use the specification of boolean functions (example 3.5). The specification for an automaton is essentially a parameter. The specification BOOL is interpreted initially, the specification STRING is interpreted freely with respect to the input alphabet Σ, and the other specifications, DFA and MINIMAL, are interpreted loosely.

spec DFA =
enrich BOOL
sorts
  Q, /* the set of states */
  Σ /* the input alphabet */
operations
  $q_0 : \; \to Q$ /* the initial state */
  is-final : $Q \to$ BOOL /* predicate for final states */
  $\delta : Q, \Sigma \to Q$ /* transition function */
end

spec STRING =
sorts Σ, STRING
operations
  $\epsilon : \; \to$ STRING /* empty string */
  $_\,_ : \Sigma,$ STRING $\to$ STRING /* adjoin letter to string */
end

spec MINIMAL =
enrich DFA, STRING
operations
  $\hat{\delta} : Q,$ STRING $\to Q$ /* extension of δ to strings */
  $_ \equiv _ : \; Q, Q \to$ BOOL /* behavioural equivalence of states */
axioms
  $\hat{\delta}(q, \epsilon) = q$
  $\hat{\delta}(q, a\omega) = \hat{\delta}(\delta(q, a), \omega)$

  $q_1 \equiv q_2$ **if** $\forall p \in$ STRING $\cdot$ is-final$(\hat{\delta}(q_1, p)) =$ is-final$(\hat{\delta}(q_2, p))$
end

□

This example illustrates that initial restrictions allow the convenience of assuming initial semantics for certain specifications while adopting loose semantics in general. In the absence of initial restrictions, the carrier for BOOL may be just one element (implying T = F!) or STRING may be interpreted as sets or bags.

Here is the formal definition of data constraints and initial restrictions.[30]

DEFINITION 4.39: *σ-free.* Let $\sigma: T_1 \rightarrow T_2$ be a theory morphism, and let $\mathcal{F}_\sigma: Mod[T_1] \rightarrow Mod[T_2]$ be the free functor left adjoint to the reduct functor $_|_\sigma: Mod[T_2] \rightarrow Mod[T_1]$. A model $M_2 \in Mod[T_2]$ is said to be σ-free if it is naturally[31] isomorphic to the free model over it σ-reduct, i.e., $\mathcal{F}_\sigma(M_2|_\sigma) \cong M_2$. □

The definition above essentially says that M_2 is a conservative extension of $M_2|_\sigma$, i.e., no new elements are added to the carriers of $M_2|_\sigma$ (no junk) and no two elements of any carrier in $M_2|_\sigma$ are coalesced (no confusion). Alternatively, we can say that $M_2|_\sigma$ "persists" in M_2.

DEFINITION 4.40: *Data constraint.* A Σ-data constraint is a pair $\langle \sigma: T_1 \rightarrow T_2, \theta: \Sigma_2 \rightarrow \Sigma \rangle$ where σ is a theory morphism, Σ_2 is the signature of T_2, and θ is a signature morphism. A Σ-model M satisfies this data constraint if $M|_\theta$ is a model of T_2 and is σ-free. □

An initial restriction on a theory T is just a slight variation, $\langle \sigma: T_1 \hookrightarrow T_2, \theta: T_2 \rightarrow T \rangle$, with σ being an inclusion of theories and θ a theory morphism. In most practical cases, θ too is an inclusion; however, to guarantee the existence of certain categorial constructions, this added generality is needed. A theory T along with a set Δ of initial restrictions is called a *canon* [Reichel 80; Reichel 87].

An important special case of a data constraint or an initial restriction is when θ is the identity and $T_1 = \emptyset$: the constraint then produces the initial model of T_2 (and of T).

The two constraints in the example above can now be expressed as:[32]

1. $\langle \emptyset \hookrightarrow$ BOOL, BOOL $\hookrightarrow$ MINIMAL$\rangle$ which forces the initial interpretation for BOOL; and

2. $\langle$SIGMA $\hookrightarrow$ STRING, STRING $\hookrightarrow$ MINIMAL$\rangle$ which forces the free interpretation of STRING over Σ (here SIGMA is a specification containing just the sort Σ).

Data constraints provide a second-order structural induction principle for the sorts they constrain (all elements of the sort are term-generated) and an equality predicate (two terms are equal only if they are provably equal). Note that this induction principle is crucial for proving the (behavioural) equivalence of minimized automata (obtained using the equivalence defined in the example above) and unminimized automata.

Hierarchies

This approach was proposed by Wirsing et al. [Wirsing et al. 83] and is incorporated in the language CIP-L [Bauer et al. 85]. The basic structuring operation is inclusion, or "building on top of". Each specification contains a distinguished "primitive" part which is expected to be preserved in the overall specification. Thus the two parts of the specification can be understood separately. Hierarchy imposes a different kind of structure on specification than the specification-building operations discussed in section 4.2.

---

[30] We follow the original papers in defining constraints on theories rather than specifications.

[31] Naturality here means that the component ε_{M_2} of the counit ε determined by the adjoint pair $\mathcal{F}_\sigma \dashv _|_\sigma$ is an isomorphism, $\varepsilon_{M_2}: \mathcal{F}_\sigma(M_2|_\sigma) \cong M_2$. The naturality condition is not vacuous, because it is possible for M_2 and $\mathcal{F}_\sigma(M_2|_\sigma)$ to be isomorphic via a morphism other than ε_{M_2}.

[32] We denote the theory corresponding to a specification by the same name.

DEFINITION 4.41: *Hierarchical specification.* A hierarchical specification T is either a pair $\langle\Sigma, \Phi\rangle$ comprising a signature and a set of axioms, or is a triple $\langle\Sigma, \Phi, P\rangle$ where the hierarchical specification P (called the *primitive specification* of T) is contained in T, i.e., $\Sigma_P \subseteq \Sigma$ and $\Phi_P \subseteq \Phi$. □

NOTATION. Given an inclusion of signatures $\iota: \Sigma' \to \Sigma$ and a Σ-model M, its ι-reduct $M|_\iota$ will also be denoted by $M|_{\Sigma'}$.

DEFINITION 4.42: *Hierarchical model.* If $T = \langle\Sigma, \Phi, P\rangle$ is a hierarchical specification, a $\langle\Sigma, \Phi\rangle$-model M is called a hierarchical model of T if its Σ_P-reduct, $M|_{\Sigma_P}$ is a hierarchical model of P. When P is empty, this definition reduces to the normal definition of a (non-hierarchical) model. □

The condition imposed by the definition above is called a *hierarchy constraint.* It is a second-order axiom which restricts the class of models of T. Wirsing et al. [Wirsing et al. 83] investigate semantic and proof-theoretic conditions under which all the models of a specification satisfy this hierarchy constraint. We need some additional notions to define these conditions. We denote by $\vdash$ the deducibility relation in a specification (the proof-theoretic analogue of $\models$).

DEFINITION 4.43: *Hierarchy properties.* A hierarchical specification $T = \langle\Sigma, \Phi, P\rangle$ is called

hierarchy-preserving if for every model M of T, the primitive part of M, $M|_{\Sigma_P}$, is a model of P;

hierarchy-faithful if for every model M' of P, there is a model M of T such that M' is a subalgebra of $M|_{\Sigma_P}$;

hierarchy-persistent if it is both hierarchy-preserving and hierarchy-faithful;

sufficiently complete if for every ground term $t \in T_{\Sigma,p}$ of primitive sort p, either $T \vdash \neg D(t)$ or $T \vdash t = t'$ form some primitive ground term $t' \in T_{\Sigma_P}$;

hierarchy-conservative if for every primitive ground formula φ of P, $(T \vdash \varphi) \Rightarrow (P \vdash \varphi)$.

□

The first three are model-theoretic properties; the last two are proof-theoretic properties. Hierarchy-preservation is a "downward" property: it says that models of a bigger specification can be restricted to models of a smaller specification. Hierarchy-faithfulness is an "upward" property: it says that models of a smaller specification can be extended to models of a bigger specification. Sufficient completeness says that no new terms (apart from those already in the primitive part) are introduced in a specification. Hierarchy-conservation says that no new theorems (apart from those already in the primitive part) are introduced in a specification.

For the institution of partial first-order logic, Wirsing et al. [Wirsing et al. 83] show that sufficient completeness is a sufficient condition for hierarchy-preservation. Under the additional assumptions that the specification is hierarchy-conservative, the axioms are in

prenex form (all quantifiers at the beginning), and the existential quantifiers are uniform (see definition 3.18), a specification is also hierarchy-faithful, and thus hierarchy-persistent.

[Bauer et al. 87] contains an example of a large specification (that of the CIP transformation system) which is hierarchically built.

Stratified loose semantics

The condition of hierarchy-persistency only guarantees that the class of models of T, as a *whole*, reflects the hierarchical structure of the specification. If we consider any particular model of T, we cannot, in general, obtain its hierarchical structure. The hierarchical structure of models becomes important if we want to implement the specification T and it primitive part, P, separately. Thus Bidoit [Bidoit 87] proposes that certain functors be attached to the models of T, indicating the models of P from which they arose. The semantics of T is a class of functors showing all the ways in which models of P can be extended to models of T.

DEFINITION 4.44: *Stratified loose semantics.* Let $T = \langle \Sigma, \Phi, P \rangle$ be a hierarchical specification, with the corresponding forgetful functor $U\colon Mod[T] \to Mod[P]$ induced by the inclusion of specifications $P \subseteq T$ (this is the same as the reduct functor $_|_{\Sigma_P}$). Let $\mathcal{M}_P$ be class of (stratified)[33] models of P. Let $\widehat{\mathcal{M}}_T$ be the class of term-generated hierarchy-preserving models of T, i.e.,

$$\widehat{\mathcal{M}}_T = \{\, M \in Gen[T] \mid U(M) \in \mathcal{M}_P \,\}.$$

The semantics of the hierarchical specification T is the class $\mathcal{F}$ of (total) functors from $\mathcal{M}_P$ to $\widehat{\mathcal{M}}_T$ which are right inverses of U, i.e.,

$$\mathcal{F} = \{\, F\colon \mathcal{M}_P \to \widehat{\mathcal{M}}_T \mid U \circ F = \mathrm{id}_{\mathcal{M}_P} \}.$$

The class of (stratified) models $\mathcal{M}_T$ of T is given by

$$\mathcal{M}_T = \bigcup_{F \in \mathcal{F}} F(\mathcal{M}_P).$$

The model class $\mathcal{M}_T$ is said to be *stratified* by the functors F. □

Stratified loose semantics is a combination of loose semantics and initial semantics in that the semantics is the class of persistent functors. The model class of T is parameterized by the model class of P. The constraint imposed on the models (stratification) is higher-order and cannot be simulated directly by parameterization because it is recursively applied to all primitive specifications (e.g., the model class $\mathcal{M}_P$ above). The language Pluss [Bidoit 89; Bidoit et al. 89] (see also section 5.7) systematically incorporates stratified loose semantics into all its constructs.

Constraints in an arbitrary institution

Constraints behave just like sentences in an institution in that they restrict the class of models. Using this observation, Goguen and Burstall [Goguen and Burstall 83] show that

[33] As given by this definition, recursively. The recursion stops when there is no primitive part in a specification.

constraints can be added to any liberal institution. A liberal institution is one in which, for every signature morphism $\sigma: \Sigma \to \Sigma'$, the reduct functor $_|_\sigma: \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$ has a left adjoint $\mathcal{F}_\sigma: \mathbf{Mod}(\Sigma) \to \mathbf{Mod}(\Sigma')$. A Σ-constraint $\langle \sigma: T_1 \to T_2, \theta: \Sigma_2 \to \Sigma \rangle$ can be translated to a Σ'-constraint via a signature morphism $\theta': \Sigma \to \Sigma'$ by composing with θ', $\langle \sigma: T_1 \to T_2, \theta' \circ \theta: \Sigma_2 \to \Sigma' \rangle$. This generates a functor which extends $\mathbf{Sen}: \mathbf{Sig} \to \mathbf{Set}$. The satisfaction condition is also satisfied (see [Goguen and Burstall 83]). Thus we get an institution in which sentences can be either the original sentences or data constraints.

A weaker form of hierarchy constraint can also be added to an arbitrary institution, by relaxing the freeness requirement for M_2 in definition 4.39 above. If the counit morphism $\varepsilon_{M_2}: \mathcal{F}_\sigma(M_2|_\sigma) \to M_2$ is injective (monomorphic) then M_2 is hierarchy faithful, if it is surjective (epimorphic) then M_2 is hierarchy preserving.

An alternative approach is that followed by Ehrig and Mahr [Ehrig and Mahr 90, Chapter 7]. They define a logic of constraints to be a pair $\langle \text{Constr}, \models \rangle$ consisting of a functor Constr from the category of specifications to the category of sets[34] and a satisfaction relation between models and constraints. A theory of parameterized specifications and module specifications with constraints is also developed.

4.6 Vertical structuring and algebraic implementation

The specification building operations we have seen until now impose what is called a "horizontal" structure [Goguen and Burstall 80] on a specification. In software development there are specifications at multiple levels of abstraction, from the requirements level down to the program level. The process of software development may be viewed as adding details to a specification by making design decisions, and producing other, more concrete, specifications which realize the current specification [Lehman et al. 84; Maibaum and Turski 84; Turski and Maibaum 87]. Thus, specifications at different levels are linked by the "is a realization of" relation. The structure induced by these links on a collection of specifications is called "vertical" structure.

A natural question to ask in this context is "what does it mean for a specification SP to be implemented by a specification SP'?". The intuitive meaning of the specification SP being implemented by the specification SP' is that SP' satisfies all the properties required (specified) by SP. In practice, SP' is also more concrete (less abstract) than SP, or, equivalently, more design decisions are made in SP' than in SP. We give below the institution-independent definition of implementation [Sannella and Tarlecki 88b] which captures this intuition.

DEFINITION 4.45: *Refinement.* Given two Σ-specifications SP and SP', we say that the specification SP refines to the specification SP', written $SP \rightsquigarrow SP'$, if $Mod[SP'] \subseteq Mod[SP]$. □

This is a direct translation of the intuition described above. Since every model of SP' satisfies the specification SP (since it is also a model of SP), the specification SP' can be said to satisfy SP, and hence an implementation of SP. If the models of SP' form a strict subset of the models of SP (i.e., there are models of SP which are not models of SP'), then we can say that there are more design decisions in SP' than in SP.

[34]The objects in this category have to be classes. So either we have to use the category of classes or the category of sets in some appropriate universe.

This definition of implementation is quite restrictive because it is only defined between specifications having the same signature. This is frequently not the case in real implementations, e.g., the implementation of integers as signed natural numbers. Such implementations are obtained by building new components on top of the target specification so as to realize the properties of the source specification. "Building on top of" can be thought of as a function which transforms models to other models, possibly with a different signature. This leads to the notion of a *constructor* and a *constructor implementation.*

DEFINITION 4.46: *Constructor.* Given a function $\kappa: \mathbf{Mod}(\Sigma) \rightarrow \mathbf{Mod}(\Sigma')$ transforming Σ-models into Σ'-models, the constructor determined by κ is a specification-building operation (ambiguously denoted by the same symbol) $\kappa: \mathbf{Spec}(\Sigma) \rightarrow \mathbf{Spec}(\Sigma')$ defined by

$$\begin{aligned} Sig[\kappa(SP)] &= \Sigma' \\ Mod[\kappa(SP)] &= \{\, \kappa(M) \mid M \in Mod[SP] \,\} \end{aligned}$$

□

DEFINITION 4.47: *Constructor implementation.* A Σ-specification SP is said to be implemented by a Σ'-specification SP' via a constructor $\kappa: \mathbf{Spec}(\Sigma') \rightarrow \mathbf{Spec}(\Sigma)$, written $SP \underset{\kappa}{\rightsquigarrow} SP'$, if $SP \rightsquigarrow \kappa(SP')$, i.e., the constructor κ transforms every model of SP' into a model of SP. □

In many cases, we want to implement only part of a specification, the relevant behaviour[35] (cf. observational equivalence, section 3.4). For example, in implementing regular expressions by finite state automata, we only want to implement the recognition of strings, ignoring the building up of regular expressions. Such implementations are described using specification-building operations called abstractors, which close the model class of a specification under observational equivalence.

DEFINITION 4.48: *Abstractor.* Given an equivalence relation $\equiv\, \subseteq |\mathbf{Mod}(\Sigma)| \times |\mathbf{Mod}(\Sigma)|$ on Σ-models, the abstractor determined by the equivalence $\equiv$ is a specification-building operation $\alpha_{\equiv}: \mathbf{Spec}(\Sigma) \rightarrow \mathbf{Spec}(\Sigma)$ defined by

$$\begin{aligned} Sig[\alpha_{\equiv}(SP)] &= \Sigma \\ Mod[\alpha_{\equiv}(SP)] &= \{\, M \mid M \in \mathbf{Mod}(\Sigma) \text{ and } M \equiv N \text{ for some } N \in Mod[SP] \,\} \end{aligned}$$

□

[35] It may be argued that rather than designate certain parts of a specification as "relevant", we should not specify anything unnecessary in the first place, i.e., everything in every specification is relevant. Unfortunately, this is not possible. Depending on the institution used, auxiliary (hidden) functions may be necessary to specify enough characterizing properties of other functions. Moreover, auxiliary functions may also help in understanding a specification; abstract model specification [Liskov and Berzins 79] (see also section 3.4) is akin to specification-by-example, and frequently more intuitive than axiomatic characterizations.

DEFINITION 4.49: *Abstractor implementation.* A Σ-specification SP is said to be implemented by a Σ'-specification SP' with respect to an abstractor $\alpha\colon \mathbf{Spec}(\Sigma) \to \mathbf{Spec}(\Sigma)$ via a constructor $\kappa\colon \mathbf{Spec}(\Sigma') \to \mathbf{Spec}(\Sigma)$, written $SP \overset{\alpha}{\underset{\kappa}{\leadsto}} SP'$, if $\alpha(SP) \leadsto \kappa(SP')$, i.e., the constructor κ transforms every model of SP' into a model which is $\equiv_\alpha$-equivalent to some model of SP. □

A typical scheme for implementation of a specification SP (called the "source") by a specification SP' (called the "target") is the following three-step sequence:

1. extend the target specification with new sorts and operations corresponding to those in the source; add axioms characterizing the new operations;
2. hide the low-level operations in the target and the auxiliary ones introduced during the previous step; and
3. identify multiple representations of the source components in the target (generally, by imposing a congruence via some equations).

A number of variations of this basic scheme have been proposed in the literature: [Goguen et al. 78; Ehrig et al. 82; Broy et al. 86; Ehrich 82; Ehrich 81; Hupbach 80; Ganzinger 83; Turski and Maibaum 87]. The variations have to do with the institution used, the sequence of steps, the kinds of intermediate operations permitted, and properties (such as composability) of the implementation.

We now define three specification-building operations corresponding to the three steps above (taken from [Sannella and Tarlecki 88b]):

extend SP **by sorts** S **operations** Ω **axioms** Φ

derive from SP **by** σ

quotient SP **wrt** E

The operation **extend** conservatively extends a specification (see the discussion after definition 4.13). The operation **derive** is used to hide some components of a specification (see definition 4.14). The operation **quotient** imposes the congruence generated by the equations E on the models (see definition 3.2).

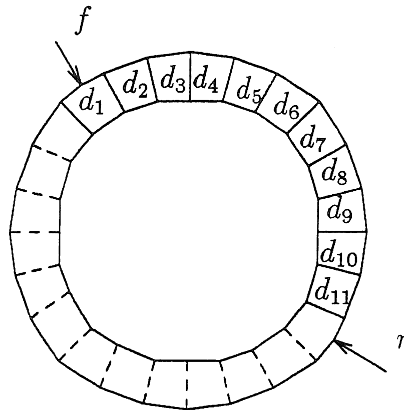
EXAMPLE 4.50. We now show an example of a constructor implementation of bounded queues by circular arrays. We use the specification of bounded queues from example 2.44, in the institution of partial first-order logic. Circular arrays are defined below. The specification NAT-MOD below is to be interpreted initially (see definition 4.40).

```
spec NAT-MOD =
based on
  BOOL,
  NAT        /* the standard specification for natural numbers */
             /* with addition, subtraction, and multiplication */
operations
  _ ≡ _ ⟦mod _⟧ : NAT, NAT, NAT → BOOL
axioms
  m ≡ n ⟦mod p⟧
       if ∃k · m = n + k × p or ∃k · n = m + k × p
end
```

```
spec C-ARRAY =
based on NAT-MOD
sorts C-ARRAY, VALUE
operations
        ⊥ :                          → VALUE     /* "undefined" or error value */
    empty :                      NAT → C-ARRAY   /* create array of a specified size */
     _[_] :             C-ARRAY, NAT → VALUE     /* retrieve value stored at an index */
  _[_←_] : C-ARRAY, NAT, VALUE → C-ARRAY         /* assign a value at an index */
     size :                  C-ARRAY → NAT
axioms
  size(empty(n)) = n
  size(a[i←x]) = size(a)

  empty(n)[i] = ⊥
    /* what is stored can be retrieved */
  a[i←x][i] = x
  a[i←x][j] = a[j] if i ≠ j
    /* later assignments override; independent assignments commute */
  a[i←x][i←y] = a[i←y]
  a[i←x][j←y] = a[j←y][i←x] if i ≠ j
    /* circularity */
  a[i] = a[j] if i ≡ j ⟦mod size(a)⟧
end
```

A queue of bound n is represented by a circular array of size $n+1$ (the extra cell is to distinguish between a full queue and an empty queue) together with two pointers (indices), f and r, into the array to indicate the front and rear of the queue, respectively. Here is a queue of bound 23 with data elements d_1 through d_{11} in it; f points to the front of the queue, and r points to the cell after the end of the queue.



Here is the extend step of the implementation. A queue is empty if $f = r$ and full if $f = r + 1$ (modulo the size of the array).

spec QUEUE-1 =
extend C-ARRAY **by**
sorts QUEUE
operations

makeq :	C-ARRAY, NAT, NAT $\rightarrow$ QUEUE
emptyq :	NAT $\rightarrow$ QUEUE
putq :	VALUE, QUEUE $\rightarrow$ QUEUE
getq :	QUEUE $\rightarrow$ VALUE
popq :	QUEUE $\rightarrow$ QUEUE
bound :	QUEUE $\rightarrow$ NAT
length :	QUEUE $\rightarrow$ NAT

axioms
$\text{emptyq}(n) = \text{makeq}(\text{empty}(n+1), 0, 0)$
$\text{putq}(x, \text{makeq}(a, f, r)) = \text{makeq}(a[r \leftarrow x], f, r+1)$ **if** $f \not\equiv r + 1$ $[\![\text{mod size}(a)]\!]$
$\text{getq}(\text{makeq}(a, f, r)) = a[f]$
$\text{popq}(\text{makeq}(a, f, r)) = \text{makeq}(a, f+1, r)$
$\text{bound}(\text{makeq}(a, f, r)) = \text{size}(a) - 1$
$\text{length}(\text{makeq}(a, f, r)) = l$ **where** $l < \text{size}(a) \wedge f + l \equiv r$ $[\![\text{mod size}(a)]\!]$
end

The next step is to hide the array operations from QUEUE-1 and rename VALUE to DATA.

spec QUEUE-2 =
derive from QUEUE-1 **by**

NAT $\mapsto$ NAT
QUEUE $\mapsto$ QUEUE
DATA $\mapsto$ VALUE
emptyq $\mapsto$ emptyq
putq $\mapsto$ putq
getq $\mapsto$ getq
popq $\mapsto$ popq
length $\mapsto$ length
bound $\mapsto$ bound

end

Finally, we identify multiple representations of the same queue: the relevant portion of a queue is the part of the circular array which is between the indices f and r. Moreover, the particular values of f and r do not matter, only the relative position is significant.

spec QUEUE-3 =
quotient QUEUE-2 **wrt**
let $q = \text{makeq}(a, f, r),\ q' = \text{makeq}(a', f', r')$ **in**

$q = q'$
if $\text{length}(q) = \text{length}(q') \wedge \text{size}(a) = \text{size}(a') \wedge$
$\forall i \in \text{NAT} \cdot (0 \leq i < \text{length}(q)) \Rightarrow a[f+i] = a'[f'+i]$

end

Every model of QUEUE-3 looks exactly like a model of BOUNDED-QUEUE, and thus we have a constructor via which bounded queues are implemented by circular arrays. □

4.7 Concluding remarks on structure

Structure is inherent to software: whether it is modelled as stepwise refinement [Wirth 71; Ehrig et al. 80a; Lehman et al. 84; Turski and Maibaum 87], or modelled using 2-categories [Goguen and Burstall 80]. Structure is essential to the economy of reasoning: for humans the difference may be between clarity and opqaueness of specifications; for computers the difference may be between decidability and undecidability. The theory of institutions enables us to study structure at an abstract level, without being tied down to some underlying logic.

As an example, consider the results regarding compatibility of horizontal and vertical specification-building operations. Vertical structure arises because of layers of description of a system at different levels of abstraction. Horizontal structure arises because of modularity within each abstraction level. Compatibility between horizontal and vertical structure means that, if a specification is built (horizontally) out of some pieces, then each piece can be implemented (vertically) independently, such that when the implementations are composed (horizontally), the result implements the original specification. An institution-independent proof that parameterization commutes with implementation is given in [Sannella and Tarlecki 88b]. Slightly less general results regarding other horizontal operations are given in [Wirsing 86]. Results for the institution of equational logic are given in [Ehrig and Kreowski 82].

5 LANGUAGES

As is apparent from the rest of this paper, we feel that the primary aspect of a specification language is the structuring mechanisms it provides. In this section, we highlight the key structuring constructs provided by a few well-known algebraic specification languages; the underlying institutions only get a passing mention. For detailed descriptions, we refer the reader to papers by the inventors of these languages. A summary of language features is given in figure 5.

5.1 ACT-ONE [Ehrig and Mahr 85]

ACT-ONE uses equational specifications with total algebras and initial semantics. It provides a few simple constructs for putting together parameterized specifications: combine, rename, actualize.

Basic specifications in ACT-ONE are of three kinds:

spec Σ, Ω, Φ, with initial semantics;

formalspec $\Sigma_P, \Omega_P, \Phi_P$ (parameter) with loose semantics; and

pspec $\Sigma_P, \Omega_P, \Phi_P, \Sigma, \Omega, \Phi$ (parameterized specification) with a free functor as semantics

The combine operation

$\langle\text{pspec}\rangle_1$ **and** $\langle\text{pspec}\rangle_2$ **and** ... **and** $\langle\text{pspec}\rangle_n$

produces the union of specifications. Name clashes need to be explicitly avoided by renaming. All the sort and function names used in a specification are global; thus the semantics of the combine operation allows the sharing of specifications. Renaming is done by providing an injective signature morphism:

def $\langle\text{new-pspec}\rangle$ **is**
  $\langle\text{old-pspec}\rangle$ **renamed by**
   $\langle\text{sort-names}\rangle$ $\langle\text{new-sort}\rangle_1$ **for** $\langle\text{old-sort}\rangle_1$
      $\ddots$
      $\langle\text{new-sort}\rangle_n$ **for** $\langle\text{old-sort}\rangle_n$
   $\langle\text{op-names}\rangle$ $\langle\text{new-op}\rangle_1$ **for** $\langle\text{old-op}\rangle_1$
      $\ddots$
      $\langle\text{new-op}\rangle_n$ **for** $\langle\text{old-op}\rangle_n$
end of def

Actualization is the instantiation of the parameter of a parameterized specification by providing an actual parameter along with a signature morphism. The syntax is similar to that of renaming. The resulting specification is obtained by substituting the formal parameter part of the original specification with the actual parameter. Unlike parameter passing by

	ACT-ONE	ASL	CIP-L	Clear	Larch	OBJ	Pluss
Institution	Equational Logic	Partial First-order Logic	Partial First-order Logic	Conditional Eqn Logic	Equational Logic	Order-sorted Logic	Predicate Calculus
Semantics	Initial	Loose	Loose with hierarchies	Loose with data constraints	Pred. Calc. Theories	Initial	Stratified, initial, loose
Incremental building	**combine**	+	**based on** **include**	+ **enrich**	**imports** **includes** **assumes**	**protecting** **extending** **using**	**use** **enrich**
Sharing	–	–	–	Based theories	–	–	Implicit
Parameterized spec. semantics	Free functor	λ-expression	–	Theory morphism	–	Free functor	Stratified
Parameter passing	Pushout	Application	Textual substitution	Colimit	–	Pushout	Pushout
Renaming	**rename**	Signature morphism	**rename**	View	**with**	*	View
Name hiding	–	**derive**	Export interface	View	–	**hidden**	**export** **forget**
Model management	–	**reachable** **observe**	–	**derive** Data theories	**generated by** **partitioned by**	–	Generators **sketch, draft**
Error handling	–	Partial functions	Partial fns Domain conditions	Error algebras	**exempts** Incomplete specs	Subsorts Error supersorts	Partial fns Preconditions
Special features	Two-level semantics	Inst-indep Obs. equiv. Recursive specs	Hierarchy constraints	Based theories Colimits	Two-tiered Incomplete specs	Executable Retracts Attributes	Inst-indep Stratified sem Spec stages

Legend:

language keywords are in boldface

"–" stands for "not supported"

Figure 5: Summary of language features

pushout, it is assumed that there are no name clashes between the actual parameter and the target specification. The semantics is obtained by the free functor which translates models of the parameter to a free extension which is a model of the target specification. To ensure correctness of parameter passing, all parameterized specifications are assumed to have persistent free functors as semantics. Methodologically, this is achieved by always adding new specifications to a persistent library to produce a new persistent library (see [Ehrig and Mahr 85, p. 292]).

An interesting aspect of ACT-ONE is that its semantics is given at two levels, the specification level (specifications and signature morphisms), and the model level (persistent free functors), in such a way that the two semantics are compatible in the sense explained below.

Using the notion of a context-free grammar G as an initial algebra T_G, a semantic function which provides denotations in a semantic domain D is a homomorphism $m: T_G \rightarrow D$, provided the semantics is compositional [Goguen et al. 77; Janssen and vanEmde Boas 82] (cf. section 4.2). Now, if $T_{\text{ACT-ONE}}$ is the initial algebra for the grammar for ACT-ONE, the two levels of semantics are homomorphisms into semantic domains, say D and $D^{\#}$:

$$[\![_]\!]: \; T_{\text{ACT-ONE}} \rightarrow D$$
$$[\![[\![_]\!]]\!] \; T_{\text{ACT-ONE}} \rightarrow D^{\#}$$

The two levels of semantics of ACT-ONE are compatible in the sense that there is a function $_^{\#}: D \rightarrow D^{\#}$ such that the following diagram commutes (see [Ehrig and Mahr 85, section 10.21]):

$$\begin{array}{ccc} & T_{\text{ACT-ONE}} & \\ {\scriptstyle [\![_]\!]} \swarrow & & \searrow {\scriptstyle [\![[\![_]\!]]\!]} \\ D & \xrightarrow{\;_^{\#}\;} & D^{\#} \end{array}$$

5.2 ASL [Astesiano and Wirsing 86; Wirsing 86]

ASL is a kernel language which provides a small set of simple constructs for building algebraic specifications in a modular fashion. The intent of ASL is to isolate the core structuring concepts around which other specification languages, perhaps with better and friendlier syntax, can be designed.

ASL is a higher-order applicative language supporting, as usual, functional abstraction, function application, and (recursive) declaration. However, unlike conventional functional languages which deal with lists, sequences, etc., the entities of interest in ASL are the ingredients comprising algebraic specifications: sorts, operations, terms, formulas, signature morphisms, etc.

The primary entity of concern in ASL is a specification. The denotation of a specification can be taken to be a class of algebras, or a theory. The semantics of ASL proposed in [Wirsing 86] uses the institution of partial first-order logic. However, the structuring mechanisms in ASL are largely independent of the institution. ASL provides six constructs to handle specifications:

1. build a specification from a signature and axioms;
2. form the sum of two specifications;
3. translate a specification via a signature morphism;
4. constrain the models of a specification to be reachable;
5. expand the class of models using observational equivalence;
6. parameterize a specification.

The semantics of these constructs is very similar to the institution-independent operations we described in section 4.2. Let $Alg[_]$ denote the class of algebras associated with a signature or a specification in the institution of partial first-order logic. Types in ASL are called "**modes**". In the description below, Σ and Σ' are signatures, SP is a Σ-specification and SP' is a Σ'-specification.

signature Σ **axioms** Φ

signature: Σ
models: $\{ A \in Alg[\Sigma] \mid A \models \Phi \}$

$SP + SP'$

signature: $\Sigma \cup \Sigma' \quad (= \Sigma'')$
models: $\{ A \in Alg[\Sigma''] \mid A|_\Sigma \in Alg[SP]$ and $A|_{\Sigma'} \in Alg[SP'] \}$

reachable SP **on** S **with** F (where $S \subseteq sorts[\Sigma], F \subseteq ops[\Sigma]$)

signature: Σ
models: $\{ A \in Alg[SP] \mid$ for each sort $s \in S$,
all elements of A_s are interpretations of terms of sort s
generated by operations in F with variables in $(sorts[\Sigma] - S) \}$

derive from SP' **by** σ(where $\sigma: \Sigma \to \Sigma'$ is a signature morphism)

signature: Σ
models: $\{ A|_\sigma \mid A \in Alg[SP'] \}$

observe SP **wrt** W (where $W \subseteq T_\Sigma(X)$ is a set of terms)

signature: Σ
models: $\{ A \in Alg[\Sigma] \mid \exists B \in Alg[SP] \cdot A \equiv_W B \}$ (see definition 5.1 below)

$\langle$mode$\rangle$**funct** $f \equiv \lambda \mathbf{m}\, x{:}\, r \cdot e$

application: $f(a) = e[x/a]$ if a is of mode **m** and satisfies the requirement r

The **observe** operation requires some explanation because it is slightly different from the observational equivalence discussed in section 3.4. The equivalence on algebras $\equiv_W$ is defined by:

DEFINITION 5.1: *W-equivalence.* If A and A' are Σ-algebras and W is a set of Σ-terms with variables in X, then A and A' are W-equivalent, written $A \equiv_W A'$, if there exist surjective valuations $v: X \to A$ and $v': X \to A'$ such that for all t, t' in W,

$$A \models_v t = t' \text{ iff } A' \models_{v'} t = t', \text{ and}$$
$$A \models_v D(t) \text{ iff } A' \models_{v'} D(t).$$

□

Sorts, operations, formulas, signatures, and specifications are all first class objects in ASL and operators are provided to decompose them into their constituents. In addition, ASL provides sets as a primitive type. Besides operations for creating sets by explicit enumeration, union, intersection, and complement, ASL provides a powerful set-building constructor, comprehension: $\{ x \mid \varphi(x) \}$. Using this construct, we can build infinitary specifications containing operator or axiom schemes.

The semantics of ASL is given in terms of domains (complete partial orders). All operations of ASL are monotonic. As a result, one can define recursive specifications such as Lisp-like lists: a list is either an atom or a pair of lists.

5.3 CIP-L [Bauer et al. 85; Bauer et al. 81; Wirsing et al. 83; Bauer et al. 78; Bauer et al. 89]

CIP-L is a wide-spectrum language incorporating the wide variety of styles used in a transformational software development process. CIP-L has facilities for describing both data structures and control structures, using algebraic types and the "scheme" language, respectively. The algebraic types provide semantics for the basic symbols which are manipulated by the language. The scheme language is an applicative language with a variety of control structures, and procedural, imperative, parallel, and non-deterministic constructs. In this section, we will only describe the algebraic component of the language. For a complete definition of the language, see Bauer et al. [Bauer et al. 85].

Algebraic specifications in CIP-L are hierarchically built specifications with loose semantics, using the institution of partial first-order logic. A typical type definition looks like

type $T = S_v, C_v, \Omega_v$:
  based on P;
  sort S,
  funct C, Ω;
  laws Φ
end of type

where $\Sigma_v = \langle S_v, C_v, \Omega_v \rangle$ is the signature of visible sorts, constants, and operations, P is a collection of primitive types, $\Sigma = \langle S, C, \Omega \rangle$ is the signature of the type T, and Φ is a set of axioms. Hierarchy constraints are imposed on the type T (cf. section 4.5). T is said to be persistent if for every model A_P of a primitive type P, there is a model A of $\langle \Sigma, \Phi \rangle$ such that the Σ_P-reduct of A is isomorphic to A_P. A model A of $\langle \Sigma, \Phi \rangle$ is said to be hierarchy-preserving if for every primitive type P, its Σ_P-reduct, $A|_{\Sigma_P}$ is a model of P. The semantics of a persistent type T is given by the Σ_v-reducts of the class of term-generated, hierarchy-preserving models of $\langle \Sigma, \Phi \rangle$.

Another way to build types in CIP-L is using the **include** construct; no hierarchy constraints are imposed in this case. The semantics is given by the type obtained by textually substituting the included type. Name clashes have to be explicitly avoided by renaming.

The parameterization mechanism provided in CIP-L is simple textual substitution. Parameterized types are called "type schemes":

type $T = (\langle \text{parameter} \rangle) \Sigma_v$:
  $\langle \text{body} \rangle$
end of type

A parameter consists of a signature and (optionally) a collection of axioms. Instantiation is done by the **include** construct,

include ⟨type⟩ **as** ⟨renaming⟩

where renaming is done by providing a list of names for the visible constituents of the parameterized type. The included type has to satisfy the axioms specified in the parameter; otherwise, the resultant type may be inconsistent. Parameter theories can be described by using the **type** construct but without any visible constituents.

CIP-L also provides shorthand notations for commonly used types such as sums and products. Such entities are called **modes** and expand to normal types via transformation rules. Here is an example,

mode nat-pair = pair(**nat** left, **nat** right)

which specifies a pair of **nat**s with constructor "pair" and selectors "left" and "right".

5.4 Clear [Burstall and Goguen 77; Burstall and Goguen 79; Burstall and Goguen 81]

Clear was one of the earliest algebraic specification languages to explicitly address the issue of building specifications in a structured manner. Clear manipulates theories, closed sets of sentences, in the institution of equational logic with loose semantics. Theories may be shared. To account for this, the semantics of Clear is given in terms of based theories and the theory building operations are interpreted as colimits in the category of based theories.

The basic operation for producing a theory is to specify its signature and the which generate it axioms (called a theory presentation):

theory sorts S **opns** O **eqns** E **endth**

More complex theories can be built by taking the sum of two already existing theories (actually, the coproduct, to take into account shared theories) or by enriching an existing theory with new sorts, operations, and equations.

$T + T'$

enrich T **by sorts** S **opns** O **eqns** E **enden**

The semantics of the "enrich" operation is a theory morphism from the original theory to the enriched theory.

We can also forget some parts of a richer theory using "derive". "Derive" needs a signature morphism to specify the parts of the theory which are to be retained.

derive sorts S **opns** O **from** T **by** σ **endde**

where $\sigma\colon \langle S, O\rangle \rightarrow Sig[T]$ is a signature morphism into the signature of the theory T. The semantics of the derive operation in Clear is different from the semantics of the institution-independent operation described in section 4.2. The derived theory contains all the sentences of the original theory translated via the (inverse of the) signature morphism.
Clear expressions can be named using

const $N = \langle \text{expression} \rangle$

Parameterized theories are specified as

proc $N(P\!:\!T) = \langle \text{expression} \rangle$

where N is the name of the parameterized theory, P is the name of the parameter and T is the requirement theory an actual parameter has to satisfy. Instantiation of a parameter is done using a pushout of based theories, as explained in section 4.3.

Clear provides data constraints (definition 4.40) to force certain theories to be interpreted initially or freely. This is done by using the syntax **data sorts** rather than **sorts** in the "enrich" and "derive" operations.

All functions in Clear are total. Errors are handled by using error algebras [Goguen 78].

5.5 OBJ [Goguen 89; Goguen and Winkler 88; Futatsugi et al. 85; Goguen and Tardo 79; Goguen et al. 85; Gnaedig et al. 90]

OBJ[36] is a logical programming language[37] rigorously based on order-sorted logic. It can be seen as an implementation of Clear. The top-level entities in OBJ are objects (executable code), theories (nonexecutable properties), parameterized modules with interface theories, views for specializing theories, and module expressions for building modules.

Objects and theories are introduced by the following syntax.

obj ⟨object-name⟩ **is**	**th** ⟨theory-name⟩ **is**
sorts ...	**sorts** ...
subsorts ...	**subsorts** ...
ops ...	**ops** ...
vars ...	**vars** ...
eq ...	**eq** ...
endo	**endth**

OBJ has both a denotational semantics—initial order-sorted algebras for objects and loose semantics for theories—and an operational semantics based on order-sorted rewriting. User-definable evaluation strategies (eager, lazy, precedence, etc.) are available on an operation by operation basis. Common properties such as associativity and commutativity can be declared as attributes to operations rather than specifying them using equations.

Since OBJ is based on order-sorted logic, it supports subsorts. Operator names can be overloaded, thus providing both subsort-polymorphism and ad-hoc-polymorphism. A key feature of OBJ is "retracts", which are used to conservatively type unparsable expressions; if the retracts are not removed at rewrite time, they provide informative error messages (see section 2.9).

OBJ provides three ways of importing modules:

[36] In this section, by OBJ we mean the latest version, OBJ3.

[37] The phrase "logical programming language" has a technical meaning [Goguen 87b]. Programs are built from sentences in an underlying institution. Computation is deduction in the institution. The denotational semantics is given by the models in the institution and is usually initial semantics. Moreover, the deduction system is expected to be sound and complete.

1. protecting (conservative extension),
2. extending (consistent extension),
3. using (no guarantees).

Parameterized objects and theories have interface theories in the header:

obj $P[X :: T]$ **is**

where X is the parameter and T is the interface theory. Parameters are instantiated by default views or by an explicit specification of a signature morphism as in

$P[X] * (\langle\text{sort}\rangle$ **to** $\langle\text{sort}\rangle\ \langle\text{op}\rangle$ **to** $\langle\text{op}\rangle)$.

Parameter theories, at present, only serve a descriptive purpose: the implementation does not verify the semantic conditions for views.

OBJ does not provide a "derive" operation for building theories (as in Clear or ASL); however, operations can be hidden using the attribute "hidden". Also, sorts and operations can be hidden in module expressions using the $*$-syntax of parameter instantiation above.

5.6 Larch [Guttag et al. 85; Guttag 86; Guttag and Horning 86b; Guttag and Horning 86a; Guttag et al. 82]

The Larch family of specification languages is designed with pragmatics being the primary concern. Larch is based on a two-tiered approach to specification:

- a set of *interface* languages (one for each programming language) are used to describe programming language dependent features such as side effects, error handling, communication with the environment, etc.;
- the *shared* language is used to describe the underlying abstractions using theories which are independent of any programming language.

The Larch shared language is based on algebraic specification, and is derived from one of the earliest works in this area [Guttag 75]. We will only describe the Larch shared language here.

The basic unit of specification in Larch[38] is a *trait*,

⟨trait-name⟩: **trait introduces** ⟨OpSig⟩ **constrains** ⟨Ops⟩ **so that** ⟨Equations⟩

which introduces some operators (along with their signatures) and equations for some subset of the operators. A trait is not required to completely define all the operators it introduces. Every trait denotes a first order predicate calculus theory generated by its equations, the single inequation `true` $\neq$ `false`. There are no meta-rules that the theory is initial or final. Richer theories corresponding to the initial or final assumptions are obtained by

⟨Sort⟩ **generated by** [⟨OpList⟩]

[38]The name "Larch", when not qualified will mean the Larch shared language.

which adds a structural induction rule for a sort, and

⟨Sort⟩ **partitioned by** [⟨OpList⟩]

which declares a set of observers for distinguishing elements of a sort.

Larch is intended to support incremental development of specifications, and provides three operators for building upon other theories:

1. ⟨Trait-1⟩ **imports** ⟨Trait-2⟩ which requires that ⟨Trait-1⟩ be a conservative extension of ⟨Trait-2⟩;

2. ⟨Trait-1⟩ **includes** ⟨Trait-2⟩ which produces the union of the two theories; and

3. ⟨Trait-1⟩ **assumes** ⟨Trait-2⟩ which is similar to **includes** except that it is used for parameterizing a trait—when such a trait is used by another trait, the assumption has to be discharged by the using trait.

Sort and operator names in an imported trait can be remaned using the clause

⟨Trait⟩ **with** [⟨NewName⟩ **for** ⟨OldName⟩].

Concordant with the philosophy of incrementally developing specifications, all theory building operators in Larch are monotonic, in that they can only add (but not delete) sentences to a theory. Larch provides two operators for explicitly stating intended consequences of a theory—these do not add any new sentences to a theory; however, they introduce redundancy and are useful for checking. These two operators are:

1. ⟨Trait⟩ **implies** [⟨Properties⟩] which specifies properties that should be derivable from the trait; and

2. ⟨Trait⟩ **converts** [⟨OpList⟩] which specifies that the operators in [⟨OpList⟩] are completely defined in terms of other operators in the trait.

As mentioned before, errors are handled in the interface languages. However, some terms can be explicitly declared to be undefined using

exempts ⟨TermList⟩.

This is especially useful for excluding some terms from the influence of a **converts** clause.

It is interesting to note that the two-tiered approach allows Larch the luxury of not having to deal with issues like hiding of names,[39] errors, non-determinism, and parameterization.[40]

[39]In one sense, the entire shared language can be considered hidden from the interface languages.

[40]Although the **assumes** clause permits a limited form of parameterization.

5.7 Pluss [Bidoit 89; Bidoit et al. 89; Bidoit 87]

Pluss is a specification language based on the kernel language ASL and designed for building structured specifications. Like ASL, Pluss is largely independent of the underlying logic. Here, we will assume first-order predicate calculus with partial operations.

Pluss has two distinguishing features:

1. A careful distinction is made between completed specifications and specifications under development, introduced by the keywords **spec** and **sketch**, respectively.[41]

2. Pluss adopts stratified loose semantics, which is a generalization of initial and loose semantics (cf. section 4.44 and [Bidoit 87]).

To enhance readability, Pluss supports a syntax which operation names that are natural language sentences with slots for arguments:

the object at _ in _: Path $*$ Directory $\rightarrow$ (File $\cup$ Directory)

_ plus _ added under _: Directory $*$ (File $\cup$ Directory) $*$ Path $\rightarrow$ Directory

Sort unions are a convenient way of overloading operations (when uses as domains) and of specifying partial operations with results in one of many sorts (when used as codomains). Overloading and coercion are also allowed, thus providing capabilities similar to those of order-sorted logic. The domains of definition of partial operations can specified using **preconditions**:

$\langle$operation$\rangle$ **defined when** $\langle$boolean term$\rangle$

Associated with a **spec** is a fixed class of finitely term-generated models. Each sort in a **spec** is required to have a set of **generators**. **Sketch** specifications are specifications under development. The have more flexible semantics: sorts need not have generators, and *all* models, including non-finitely generated models are allowed. To avoid trivial models, initial semantics can be specified by using the **basic spec** construct (in which case the axioms are restricted to be equations or Horn clauses).

The fundamental structuring construct is enrichment: denoted by the keyword **use** in the case of **specs** and **enrich** in the case of **sketch**es. When using a **spec**, a hierarchy constraint is imposed: the **use**d specification should be protected, i.e., the enriched specification is a conservative extension. This allows the understanding and implementing of specifications independent of the context of their use. The semantics of the **use** construct is stratified (see definition 4.44).

No hierarchy constraint is imposed when **sketch**es are enriched.[42] The semantics is defined to be the class of models of the union of the two specifications.

Sketch specifications can be converted into completed ones using

[41] In his thesis [Bidoit 89], Bidoit describes another kind of specification, **draft**, which is intermediate between **spec** and **sketch**. The only difference between **draft**s and **sketch**es is that the former have generators.

[42] This is a choice which is determined by the underlying institution [Bidoit 89, p. 145]. A common use of enrichment is to extend the domain of definition of partial functions. Thus, it is unreasonable to impose hierarchical consistency.

spec ⟨completed-spec-name⟩ **from** ⟨sketch-spec-name⟩

and providing generators for each sort. Visibility of parts of a specification can be controlled by using **export** and **forget**, which are essentially signature morphisms mapping hidden sorts and operations onto inaccessible names (compare with the **derive** operation in ASL).

Parameterized specifications are defined using the keywords **generic sketch** or **generic spec**. The semantics of parameterized specifications is the same as that of enrichment (stratified for **spec**s and loose for **sketch**es), with the target specification considered as an enrichment of the parameter specification.

Formal parameter specifications are defined using the **par** construct. Having a separate construct for formal parameters allows producing reusable parameter specifications. Instantiation is done using the **as** construct and providing a fitting morphism. The semantics of an instantiated specification is obtained as the model class of the target specification enriched by the actual parameter, subject to the renaming defined by the fitting morphism. Correctness of parameter passing is similar to that in section 4.3, i.e., the semantics obtained by syntactic substitution agrees with the model translation by the stratification functors.

Sharing of modules is implicitly imposed in Pluss. For example, if S enriches S_1 and S_2, and the latter two specifications have sorts or operations with the same name, then for the module S to be well defined, entities with the same should have been imported from the same specification S_{12} into S_1 and S_2.

related in ways which cannot be captured by imposing equations on operations. For example, one sort may be a subset of another, or one sort may be the disjoint union of two other sorts. Thus, in addition to axioms on operations, *sort constraints* have to be introduced. Such constraints are used in order-sorted specifications to specify the domains of definition of partially defined functions. However, order-sorted specifications cannot describe the sum of two sorts, for this one needs a more powerful logic [Barr and Wells 85]. Similarly, the addition of higher-order functions to algebras implies that we have to impose sort constraints saying that some sorts are function spaces with domains and codomains in other sorts [Möller 86; Parsaye-Ghomi 82]. Thus we can see that specifications have evolved from triples of sorts, operations, and axioms to four-tuples which contain sort constraints in addition. A unifying approach to such more expressive formalisms is categorical logic [Kock and Reyes 77; Barr 86].

6.2 Some limitations

Algebraic specification until now has been purely applicative, i.e., specifications do not have any internal state and the operations cannot therefore produce any side-effects. The notion of state is a powerful abstraction mechanism, and there have attempts to include state into algebraic specifications [Goguen and Meseguer 87b; Wagner 89; Wagner 85]. However, the semantics of such specifications is quite complex. Another area in which algebraic specification can be extended is to add non-deterministic and concurrent operations.

6.3 Specifying-in-the-large

Researchers in algebraic specification have been concerned from the very beginning with building specifications in a structured manner. The notion of institution-independent building operations reflects this concern. An indication of the success of the study of structuring methods is the specification of the CIP-S transformation system [Bauer et al. 87]: one of the largest algebraic specifications published in the literature. This specification is quite easy to understand. On the other hand, as far as specifications go, that specification might be considered small! So it is natural to ask if methods currently known will scale up. The answer depends on using algebraic specifications in practice, something which is just beginning to happen.

6.4 Future directions

The abstract study of the structure of specifications is still in its early stages. The theory of institutions is a promising step in this direction. However, there are several open problems, e.g., monotonicity and persistency of structuring operations, structured proofs about specifications, and specification in several institutions. All these are concerned with intellectual and computational economy, the guiding theme being complexity control (complexity of understanding, complexity of computation, complexity of reasoning, etc.). Consider persistency: it means that the class of models associated with a specification is preserved when it is used to build a larger specification; otherwise, the work done in characterizing the original model class is wasted. Compositional semantics and monotonicity of theories are also reflections of the same concern. Persistency is also essential for building specifications out

of reusable components. However, this is somewhat a circular problem. To design good reusable components, we have to understand and design specification-building operations which are persistent. Much work needs to be done in this area; stratified loose semantics [Bidoit 87; Gaudel and Moineau 88] and the synthesis of implementations using properties of building operations [Wirsing et al. 88; Wirsing et al. 89] are recent proposals addressing such issues.

There has been a tendency in algebraic specification to use more expressive logics. However, there is a tradeoff between expressiveness and the complexity of the deductive system associated with a logic. Thus, researchers in algebraic specification have adopted a middle ground: using a simple logic for axioms in a specification and using higher-order constraints on the models (initiality, data constraints, hierarchy constraints, stratified semantics). The interactions of such constraints with specification-building operations has not been thoroughly studied. On the other hand, the integration of more expressive formalisms into algebraic specification is also an open problem, e.g., the higher-order clichés which are characteristic of functional programming, and the state- and inheritance-oriented techniques of object-oriented programming. [Möller et al. 87; Möller et al. 88] and [Wagner 89] are recent papers addressing such an integration. [Sannella and Tarlecki 89; Sannella and Tarlecki 85] is an attempt to transfer the structuring operations of algebraic specification to a concrete language, ML.

Finally, we note that, pragmatically, algebraic specification is sandwiched between two other activities of the software lifecycle: the acquisition of specifications, and the transformation of specifications into programs. For the research in algebraic specification to translate into practice, all three aspects of this larger problem have to be addressed.

7 BIBLIOGRAPHIC NOTES

Books on algebraic specification

The monographs by Ehrig and Mahr are a textbook style introduction to equational specifications, initial semantics, modules, and specifications with constraints [Ehrig and Mahr 85; Ehrig and Mahr 90]. The monograph by Reichel describes equational specifications with partial operations and initial restrictions [Reichel 87]. The book by Turski and Maibaum introduces algebraic specification in the context of software development; the book is well written, with balanced proportions of formal and informal material [Turski and Maibaum 87]. The book by Bergstra et al. is less theoretical, with the emphasis on using algebraic specification for describing the semantics of programming languages [Bergstra et al. 89]. The monograph by Padawitz discusses Horn clause specifications, with an emphasis on rewriting techniques [Padawitz 88]. The chapter on abstract data types in [Bauer and Wössner 82] is a good introduction to the algebraic approach; the density of good concepts in this book demands careful reading. [Horebeek and Lewi 89] is an introduction to algebraic specification with an emphasis on applying it in software engineering; a variety of concepts are introduced at an intuitive level, and some non-trivial examples are included.

General papers

The three papers, [Burstall and Goguen 82], [Goguen 89], and [Meseguer and Goguen 84], form an excellent introduction to algebraic specification, especially for the uninitiated reader who wishes to understand the algebraic approach to specification. However, since these papers use initial semantics, they should be later supplemented by papers describing other approaches to semantics, e.g., [Wirsing et al. 83].

Algebra

[Cohn 81] and [Grätzer 79] are textbooks on universal algebra, a subject which forms the basis of algebraic specification. Birkhoff and Lipson generalized universal algebra (which uses one-sorted algebras) to include many-sorted algebras [Birkhoff and Lipson 70]. [Mac Lane and Birkhoff 67] is a textbook introduction to algebra, which also introduces category theory and illustrates its use to uniformly describe constructions in algebra.

[Burstall and Goguen 82] provides an excellent introduction to algebras for a computer science audience. The authors use simple, clear examples, and take great care to explain the intuition behind the various definitions and constructions.

Category theory

A computer science oriented introduction to category theory is provided in [Rydeheard and Burstall 88], [Pierce 88], and [Pitt et al. 85]. Other, very readable, introductions are [Goldblatt 84, Chapters 2 and 3], and [Herrlich and Strecker 73]. The book by Mac Lane, one of the founders of category theory, is an uncluttered, succinct, and precise treatment [Mac Lane 71]. Schubert's book is terse but comprehensive, and thus suitable as a reference book [Schubert 72]. Goguen introduces concepts from category theory while addressing substitution

and unification [Goguen 88b]. The series of papers by Goguen et al. is a description of concepts especially useful for understanding the theory underlying algebraic specification [Goguen and Burstall 84a; Goguen and Burstall 84b; Tarlecki et al. 88]. Lambek and Scott provide a brief introduction to category theory while exploring the relationship between category theory and logic [Lambek and Scott 86].

Institutions

Institutions were originally introduced in [Goguen and Burstall 83]. An earlier version of the idea, under the name "languages", can be found in the paper on the semantics of Clear [Burstall and Goguen 79]. Closely related ideas on abstract model theory were proposed by Barwise [Barwise 74]. Goguen also provides a somewhat less formal introduction to institutions and logical programming languages in [Goguen 87b]. In [Goguen and Burstall 85b], the authors show how to systematically build institutions suitable for algebraic specification. Meseguer generalizes institutions to general logics by including proofs [Meseguer 89]. Tarlecki studies institutions in the abstract, e.g., institution morphisms, and categories of institutions [Tarlecki 85a].

Specification logics, which are used to abstractly describe module interconnection, are introduced in [Ehrig et al. 89].

Institution-independent studies

Institutions can be used to abstractly study how to build algebraic specifications. Sannella and Tarlecki propose a general set of specification-building operations in [Sannella and Tarlecki 88a], a general notion of observational equivalence in [Sannella and Tarlecki 87], a general notion of implementation in [Sannella and Tarlecki 88b], and a general notion of program development in [Sannella and Tarlecki 85; Sannella and Tarlecki 89]. Tarlecki investigates the notion of freeness in an arbitrary institution [Tarlecki 85b]. The language ASL is an attempt to isolate the core set of operations needed to build specifications [Sannella and Wirsing 83; Wirsing 86; Astesiano and Wirsing 86]. Pluss is an institution-independent language built on top of ASL [Bidoit 89; Bidoit et al. 89]. Specifications in an arbitrary specification logic are studied in [Ehrig et al. 89].

Logics

Equational logic. Many of the earlier papers on algebraic specification used equational logic [Goguen et al. 78; Guttag 75; Guttag and Horning 78; Liskov and Zilles 75; Zilles 79]. The monograph by Ehrig and Mahr is a comprehensive introduction to algebraic specification using equational logic and initial semantics [Ehrig and Mahr 85]. The monograph by Reichel treats algebraic specification using equational logic and partial algebras [Reichel 87].

First-order logic. [Wirsing et al. 83] is an excellent introduction to the CIP group's approach towards algebraic specification; they use axioms in first-order logic, partial algebras, and hierarchically built specifications.

Order-sorted logic. [Goguen and Meseguer 88] is a theoretical treatment of order-sorted logic. [Goguen 89] is a less formal, and very readable, introduction to order-sorted logic, the concept of parameterized programming, and the language OBJ3. Applications of order-sorted logic can be found in the papers [Goguen 87a] and [Goguen and Meseguer 87a]. [Goguen and Winkler 88] is a user-manual for the language OBJ3 and its implementation. Rewriting techniques for executing order-sorted equations are discussed in [Gnaedig et al. 90].

Higher-order logic. The thesis by Parsaye-Ghomi was the earliest proposal to include higher-order functions in algebraic specifications [Parsaye-Ghomi 82]. The approach is based on a many-sorted version of Lawvere theories [Lawvere 63]. Poigné follows a related approach, also using categorical logic, but with different results [Poigné 86]. Some other authors generalize the CIP group's approach without using categorical logic [Broy 87; Möller 86; Möller 87]. The papers by Möller et al. attempt to add other higher-order constructs, such as power sets, to algebraic specifications [Möller et al. 88; Möller et al. 87]. [Maibaum and Lucena 80] is yet another approach in which algebras themselves are treated as first-class objects.

Goguen argues that higher-order functions are not necessary in algebraic specification, and shows how to achieve the effect of higher-order constructs while still in a first-order setting (of the language OBJ3) [Goguen 88a].

Predicate calculus, Horn clauses. The book by Turski and Maibaum introduces predicate calculus theories in the context of a step-by-step approach to software development [Turski and Maibaum 87]. The specification language Pluss uses axioms written in predicate calculus [Bidoit et al. 89; Bidoit 89]. Horn clauses, a specialization of predicate calculus formulas are considered in [Padawitz 88; Makowsky 87].

Continuous algebras. Continuous algebras and ordered algebras allow the description of infinite structures using the techniques of algebraic specification. They are described in [Goguen et al. 77; Levy and Maibaum 82; Möller 85; Tarlecki and Wirsing 86].

Categorical logic. Categorical logic is the use of category theory to study logic and model theory. The field was initiated by Lawvere's thesis, which is summarized in [Lawvere 63]. Introductory material can be found in [Lawvere 65; Lawvere 75; Kock and Reyes 77; Goguen and Burstall 84a; Goguen and Burstall 84b]. Categorical treatments of first-order logic and higher-order logic may be found in [Makkai and Reyes 77] and [Lambek and Scott 86]. Barr and Wells use a more graphical form (commuting diagrams) of specification [Barr and Wells 85; Barr 86; Wells and Barr 87].

An alternative approach of representing theories by monads is treated in [Manes 76; Rydeheard and Burstall 84].

Semantics

General. A systematic study of models of algebraic specifications, without using category theory, is contained in [Wirsing and Broy 82; Broy et al. 84; Wirsing and Broy 80]. General conditions for the existence of initial and terminal models are given in [Wirsing et al. 83].

Initial semantics. Meseguer and Goguen provide an excellent introduction to the concept of initiality [Meseguer and Goguen 84] (see also [Burstall and Goguen 82]). The ADJ group was the earliest to advocate initial algebra semantics [Goguen et al. 78; Goguen et al. 77]. The books by Ehrig and Mahr are comprehensive treatments of initial semantics [Ehrig and Mahr 85; Ehrig and Mahr 90]. Mahr and Makowsky characterize languages which admit initial semantics [Mahr and Makowsky 84; Mahr and Makowsky 83]. The monograph by Reichel treats initial semantics in the context of partial algebras [Reichel 87].

Final semantics. The work by Guttag on abstract data types implicitly uses final semantics [Guttag 75; Guttag and Horning 78]. Final semantics was explicitly advocated as an alternative to initial semantics in [Wand 79; Hornung and Raulefs 80; Kamin 83]. Parameterized specifications with final semantics are investigated in [Hornung and Raulefs 81; Ganzinger 83]. The effect of existential quantifiers and disjunctions on final semantics is discussed in [Broy et al. 79].

Loose semantics, Stratified loose semantics. Loose semantics is advocated by the CIP group in [Wirsing et al. 83]. The lattice structure of models is investigated in [Wirsing and Broy 82; Broy et al. 84; Wirsing and Broy 80]. The principle of generation for term-generated models in introduced in [Bauer and Wössner 82].

Stratified loose semantics, a variant of loose semantics with hierarchy constraints, is proposed at an intuitive level in [Bidoit 87]; details may be found in the thesis [Bidoit 89].

Behavioural semantics. Observability, a concept borrowed from automata theory, was first applied to algebraic specification in [Giarratana et al. 76]. Reichel used behavioural equivalence to unify initial and final semantics [Reichel 81] (see also the behavioural canons of [Reichel 87]). The concept of abstract modules satisfying a given behaviour was introduced in [Goguen and Meseguer 82]. Parameterization and implementation in the context of behavioural semantics is discussed in [Ganzinger 83]. [Sannella and Tarlecki 87] is an institution-independent study of observational equivalence. [Nivela and Orejas 87] proposes an institution for behavioural specifications. Structuring constructs for behavioural specifications are discussed in [Ehrig et al. 89; Orejas et al. 88].

Structuring mechanisms

Parameterization. The use of free functors to describe the semantics of parameterized specifications was introduced in [Thatcher et al. 82; Ehrig et al. 81; Ehrig et al. 80c]; free functors are also used in [Burstall and Goguen 79; Ehrich 82; Ehrig and Mahr 85]. The semantics of parameterized specifications in the context of final semantics is investigated in [Hornung and Raulefs 81; Ganzinger 83]. An institution-independent concept of parameterized specifications is given in [Sannella and Tarlecki 88a]. Parameterization for order-sorted algebras is considered in [Poigné 90].

Parameter passing by pushouts is considered in [Ehrig et al. 81; Ehrig and Mahr 85; Burstall and Goguen 79; Ehrich 82]. Several flavours of parameter passing are described in [Wirsing and Broy 82].

Constraints. Data constraints, or initial restrictions, are described in [Burstall and Goguen 79; Reichel 80; Reichel 87]. Institution-independent versions of data constraints are given in [Goguen and Burstall 83; Sannella and Tarlecki 88a].

Hierarchy constraints are described in [Wirsing et al. 83]. Stronger hierarchy constraints are introduced in [Bidoit 87; Bidoit 89].

A logic of constraints, which generalizes and includes other kinds of constraints, is proposed in [Ehrig 88; Ehrig and Mahr 90]. Specifications with constraints are discussed in [Ehrig 81; Ehrig et al. 86b; Ehrig and Mahr 90].

Implementation. An institution-independent notion of implementation is given in [Sannella and Tarlecki 88b]. Variations can be found in [Goguen et al. 78; Ehrig et al. 82; Broy et al. 86; Ehrich 82; Ganzinger 83; Turski and Maibaum 87]. Implementations of parameterized specifications are considered in [Sannella and Tarlecki 88b; Sannella and Wirsing 82; Goguen and Burstall 80; Ehrich 81; Hupbach 80]. A general implementation relation between specifications is part of the ASL language [Sannella and Wirsing 83; Wirsing 86; Astesiano and Wirsing 86].

The vertical structure of specifications is considered in a general setting in [Goguen and Burstall 80; Ehrig et al. 88].

[Bauer and Wirsing 88] considers several two-way relations between specifications.

Modules. Module specifications, which integrate parameterization and implementation, are introduced in [Blum et al. 87]. [Weber and Ehrig 86] is a more informal presentation. Modules are also the subject of the monograph [Ehrig and Mahr 90]. Modules with constraints are considered in [Ehrig et al. 86b]. Module interconnection is studied in [Ehrig and Mahr 90; Ehrig et al. 88; Ehrig et al. 89; Ehrig et al. 86a; Parisi-Presicce 88].

Languages

ACT. ACT-ONE is described in [Ehrig and Mahr 85, Appendix]. ACT-TWO is described in [Ehrig and Mahr 90, Chapter 9] and [Hansen 87].

ASL. [Astesiano and Wirsing 86] is an informal introduction. Wirsing provides a comprehensive treatment in [Wirsing 86]. An earlier version of ASL can be found in [Sannella and Wirsing 83].

CIP-L. [Bauer et al. 85] is a complete description of the language. Less formal descriptions are found in [Bauer et al. 89] and the earlier [Bauer et al. 78]. A collection of examples is contained in [Bauer et al. 81]. The theoretical basis for the algebraic part of CIP-L, hierarchical abstract data types, is described in [Wirsing et al. 83]. A specification of the CIP-S transformation system written in CIP-L is contained in [Bauer et al. 87].

Clear. The original proposal for Clear was in [Burstall and Goguen 77]. A formal semantics is provided in [Burstall and Goguen 79]. An informal introduction is provided in [Burstall and Goguen 81]. A set-theoretic semantics for Clear is described in [Sannella 84; Sannella 82].

OBJ. [Goguen 89] is an informal, and very readable, introduction to OBJ3. [Goguen and Winkler 88] is a user-manual for the language OBJ3 and its implementation. Earlier versions of the language are described in [Goguen and Tardo 79; Futatsugi et al. 85]. Operational semantics for OBJ3 is described in [Goguen et al. 85; Gnaedig et al. 90].

Larch. [Guttag et al. 85] is a complete description of the language; parts of this report are published as [Guttag and Horning 86b; Guttag and Horning 86a; Guttag 86]. The Larch interface languages are described in [Wing 87; Wing 89].

Pluss. The thesis by Bidoit is a complete description of Pluss and its semantics [Bidoit 89]. An informal introduction and an example are contained in [Bidoit et al. 89]. Stratified loose semantics (which is used in Pluss) is introduced in [Bidoit 87].

Miscellaneous

Bibliography. A comprehensive bibliography on abstract data types, until the end of the year 1982, is contained in [Kutzler and Lichtenberger 83].

Computability. A short survey on the specification of computable algebras is contained in [Meseguer and Goguen 84]. Results on the power of the initial algebra and final algebra approaches are contained in [Bergstra and Tucker 83]. The power of hierarchical specifications is investigated in [Bergstra et al. 81].

8 REFERENCES

[Astesiano and Wirsing 86]

ASTESIANO, E., AND WIRSING, M. An introduction to ASL. In *IFIP TC2/WG2.1 Working Conference of Program Specification and Transformation* (Bad Tölz, FRG, Apr. 1986), L. G. L. T. Meertens, Ed., North-Holland, pp. 343–365.

[Backus 78]

BACKUS, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM 21*, 8 (Aug. 1978), 613–641.

[Barr 86]

BARR, M. Models of sketches. *Cahiers de Topologie et Géométrie Différentielle Catégoriques XXVII*, 2 (1986), 93–107.

[Barr and Wells 85]

BARR, M., AND WELLS, C. *Toposes, Triples and Theories.* Springer-Verlag, New York, 1985.

[Barwise 74]

BARWISE, K. J. Axioms for abstract model theory. *Annals of Mathematical Logic 7* (1974), 221–265.

[Bauer and Wirsing 88]

BAUER, F. L., AND WIRSING, M. Crypt-equivalent algebraic specifications. *Acta Inf. 25* (1988), 111–153.

[Bauer and Wössner 82]

BAUER, F. L., AND WÖSSNER, H. *Algorithmic Language and Program Development.* Springer-Verlag, Berlin, 1982.

[Bauer et al. 78]

BAUER, F. L., BROY, M., GANTZ, R., HESSE, W., KREIG-BRÜCKNER, B., PARTSCH, H., PEPPER, P., AND WÖSSNER, H. Towards a wide spectrum language to support program specification and program development. In *International Summer School on Program Construction* (Marktoberdorf, July 1978), *Lecture Notes in Computer Science,* Vol. 69, Springer-Verlag, pp. 543–552.

[Bauer et al. 81]

BAUER, F. L., BROY, M., DOSCH, W., GANTZ, R., KREIG-BRÜCKNER, B., LAUT, A., LUCKMANN, M., MATZNER, T., MÖLLER, B., PARTSCH, H., PEPPER, P., SAMELSON, K., STEINBRÜGGEN, R., AND WIRSING, M. Programming in a wide spectrum language: A collection of examples. *Sci. Comput. Programming 1* (1981), 73–114.

[Bauer et al. 85]

BAUER, F. L., BERGHAMMER, R., BROY, M., DOSCH, W., GEISELBRECHTINGER, F., GNATZ, R., HANGEL, E., HESSE, W., KRIEG-BRÜCKNER, B., LAUT, A., MATZNER, T., MÖLLER, B., NICKL, F., PARTSCH, H., PEPPER, P., SAMELSON, K., WIRSING, M., AND WÖSSNER, H. *The Munich Project CIP, Volume I: The Wide Spectrum Language CIP-L, Lecture Notes in Computer Science,* Vol. 183. Springer-Verlag, Berlin, 1985.

[Bauer et al. 87]

BAUER, F. L., EHLER, H., HORSCH, A., MÖLLER, B., PARTSCH, H., PAUKNER, O., AND PEPPER, P. *The Munich Project CIP, Volume II: The Program Transformation System CIP-S*, *Lecture Notes in Computer Science,* Vol. 292. Springer-Verlag, Berlin, 1987.

[Bauer et al. 89]

BAUER, F. L., MÖLLER, B., PARTSCH, H., AND PEPPER, P. Formal program construction by transformations—computer-aided, intuition-guided programming. *IEEE Trans. Softw. Eng. 15*, 2 (Feb. 1989), 165–180.

[Bergstra and Tucker 83]

BERGSTRA, J. A., AND TUCKER, J. V. Initial and final algebra semantics for data type specifications: Two characterization theorems. *SIAM J. Comput. 12*, 2 (May 1983), 366–387.

[Bergstra et al. 81]

BERGSTRA, J. A., BROY, M., TUCKER, J. V., AND WIRSING, M. On the power of algebraic specifications. In *10th MFCS* (Štrbské Pleso, Czechoslovakia, Sept. 1981), *Lecture Notes in Computer Science,* Vol. 118, Springer-Verlag, pp. 193–204.

[Bergstra et al. 89]

BERGSTRA, J. A., HEERING, J., AND KLINT, P., Eds. *Algebraic Specification.* ACM Press, New York, 1989.

[Bidoit 87]

BIDOIT, M. The stratified loose approach: A generalization of initial and loose semantics. In *Recent Trends in Data Type Specification* (Gullane, Scotland, Sept. 1987), *Lecture Notes in Computer Science,* Vol. 332, Springer-Verlag, pp. 1–22.

[Bidoit 89]

BIDOIT, M. *Pluss, un langage pour le développement de spécifications algébriques modulaires.* PhD thesis, Université de Paris-Sud, Centre d'Orsay, May 1989.

[Bidoit et al. 85]

BIDOIT, M., CHOPPY, C., AND VOISIN, F. The Asspegique specification environment: Motivations and design. In *Recent Trends in Data Type Specification*, H. J. Kreowski, Ed., *Informatik-Fachberichte,* Vol. 116. Springer-Verlag, 1985, pp. 54–72.

[Bidoit et al. 89]

BIDOIT, M., GAUDEL, M.-C., AND MAUBOUSSIN, A. How to make algebraic specifications more understandable: An experiment with the Pluss specification language. *Sci. Comput. Programming 12* (1989), 1–38.

[Birkhoff and Lipson 70]

BIRKHOFF, G., AND LIPSON, J. D. Heterogeneous algebras. *Journal of Combinatorial Theory 8* (1970), 115–133.

[Blum et al. 87]

BLUM, E. K., EHRIG, H., AND PARISI-PRESICCE, F. Algebraic specification of modules and their basic interconnections. *J. Comput. Syst. Sci. 34* (1987), 293–339.

[Broy 87]

BROY, M. Equational specification of partial higher order algebras. In *Logic of Programming and Calculi of Discrete Design*, M. Broy, Ed., *NATO ASI Series,* Vol. F36. Springer-Verlag, Berlin, 1987, pp. 185–241.

[Broy and Wirsing 82]

BROY, M., AND WIRSING, M. Partial abstract types. *Acta Inf. 18* (1982), 47–64.

[Broy et al. 79]

BROY, M., DOSCH, W., PARTSCH, H., PEPPER, P., AND WIRSING, M. Existential quantifiers in abstract data types. In *6th ICALP* (Graz, Austria, July 1979), *Lecture Notes in Computer Science,* Vol. 71, Springer-Verlag, pp. 73–87.

[Broy et al. 84]

BROY, M., WIRSING, M., AND PAIR, C. A systematic study of models of abstract data types. *Theoretical Comput. Sci. 33* (1984), 139–174.

[Broy et al. 86]

BROY, M., MÖLLER, B., PEPPER, P., AND WIRSING, M. Algebraic implementations preserve program correctness. *Sci. Comput. Programming 7* (1986), 35–53.

[Burstall 69]

BURSTALL, R. M. Proving properties of programs by structural induction. *Computer Journal 12*, 1 (Feb. 1969), 41–48.

[Burstall and Goguen 77]

BURSTALL, R. M., AND GOGUEN, J. A. Putting theories together to make specifications. In *Proc. 5th Int. Joint Conf. on Artificial Intelligence* (1977), pp. 1045–1058.

[Burstall and Goguen 79]

BURSTALL, R. M., AND GOGUEN, J. A. The semantics of Clear, a specification language. In *Abstract Software Specifications* (Copenhagen, 1979), D. Bjorner, Ed., *Lecture Notes in Computer Science,* Vol. 86, Springer-Verlag, pp. 294–332.

[Burstall and Goguen 81]

BURSTALL, R. M., AND GOGUEN, J. A. An informal introduction to Clear, a specification language. In *The Correctness Problem in Computer Science*, R. Boyer and J. Moore, Eds. Academic Press, 1981, pp. 185–213. Also in [Gehani and McGettrick 86, pp. 363-389].

[Burstall and Goguen 82]

BURSTALL, R. M., AND GOGUEN, J. A. Algebras, theories, and freeness: An introduction to computer scientists. In *Theoretical Foundations of Programming Methodology,* M. Broy and G. Schmidt, Eds. D. Reidel Pub. Co., 1982, pp. 329–349.

[Cohn 81]

COHN, P. M. *Universal Algebra.* D. Reidel Pub. Co., Dordrecht, Holland, 1981.

[Ehrich 81]

EHRICH, H. D. On realization and implementation. In *10th MFCS* (Štrbské Pleso, Czechoslovakia, Sept. 1981), *Lecture Notes in Computer Science,* Vol. 118, Springer-Verlag, pp. 271–280.

[Ehrich 82]

EHRICH, H. D. On the theory of specification, implementation, and parameterization of abstract data types. *J. ACM 29*, 1 (Jan. 1982), 206–227.

[Ehrig 81]

EHRIG, H. Algebraic theory of parameterized specifications with requirements. In *6th CAAP* (Genoa, Mar. 1981), *Lecture Notes in Computer Science,* Vol. 112, Springer-Verlag, pp. 1–24.

[Ehrig 88]

EHRIG, H. A categorical concept of constraints for algebraic specifications. In *Categorical Methods in Computer Science, with Aspects from Topology* (Berlin, Sept. 1988), *Lecture Notes in Computer Science,* Vol. 393, Springer-Verlag, pp. 1–15.

[Ehrig and Kreowski 82]

EHRIG, H., AND KREOWSKI, H. J. Parameter passing commutes with implementation of parameterized data types. In *9th ICALP* (Aarhus, Denmark, July 1982), *Lecture Notes in Computer Science,* Vol. 140, Springer-Verlag, pp. 197–211.

[Ehrig and Mahr 85]

EHRIG, H., AND MAHR, B. *Fundamentals of Algebraic Specification 1: Equational and Initial Semantics*, *EATCS Monographs on Theoretical Computer Science,* Vol. 6. Springer-Verlag, Berlin, 1985.

[Ehrig and Mahr 90]

EHRIG, H., AND MAHR, B. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, *EATCS Monographs on Theoretical Computer Science,* Vol. 21. Springer-Verlag, Berlin, 1990.

[Ehrig et al. 80a]

EHRIG, H., KREOWSKI, H. J., MAHR, B., AND PADAWITZ, P. Compound algebraic implementations: An approach to stepwise refinement of software systems. In *9th MFCS* (Rydzyna, Poland, Sept. 1980), *Lecture Notes in Computer Science,* Vol. 88, Springer-Verlag, pp. 231–245.

[Ehrig et al. 80b]

EHRIG, H., KREOWSKI, H. J., AND PADAWITZ, P. Algebraic implementation of abstract data types: Concept, syntax, semantics and correctness. In *7th ICALP* (Noordwijkerhout, Netherlands, July 1980), *Lecture Notes in Computer Science,* Vol. 85, Springer-Verlag, pp. 142–156.

[Ehrig et al. 80c]

EHRIG, H., KREOWSKI, H. J., THATCHER, J., WAGNER, E., AND WRIGHT, J. Parameterized data types in algebraic specification languages. In *7th ICALP* (Noordwijkerhout, Netherlands, July 1980), *Lecture Notes in Computer Science,* Vol. 85, Springer-Verlag, pp. 157–168.

[Ehrig et al. 81]

EHRIG, H., KREOWSKI, H. J., THATCHER, J., WAGNER, E., AND WRIGHT, J. Parameter passing in algebraic specification languages. In *Proceedings, Workshop on*

Program Specification (Aarhus, Denmark, Aug. 1981), *Lecture Notes in Computer Science,* Vol. 134, Springer-Verlag, pp. 322–369.

[Ehrig et al. 82]

EHRIG, H., KREOWSKI, H.-J., MAHR, B., AND PADAWITZ, P. Algebraic implementation of abstract data types. *Theoretical Comput. Sci. 20* (1982), 209–263. Revised and extended version of [Ehrig et al. 80a] and [Ehrig et al. 80b].

[Ehrig et al. 86a]

EHRIG, H., FEY, W., AND PARISI-PRESICCE, F. Distributive laws for composition and union of module specifications for software systems. In *IFIP TC2/WG2.1 Working Conference of Program Specification and Transformation* (Bad Tölz, FRG, Apr. 1986), L. G. L. T. Meertens, Ed., North-Holland, pp. 293–312.

[Ehrig et al. 86b]

EHRIG, H., FEY, W., PARISI-PRESICCE, F., AND BLUM, E. K. Algebraic theory of module specifications with constraints. In *12th MFCS* (Bratislava, Czech., Aug. 1986), *Lecture Notes in Computer Science,* Vol. 233, Springer-Verlag, pp. 59–77.

[Ehrig et al. 88]

EHRIG, H., FEY, W., HANSEN, H., LÖWE, M., AND PARISI-PRESICCE, F. Categories for the development of algebraic module specifications. In *Categorical Methods in Computer Science, with Aspects from Topology* (Berlin, Sept. 1988), *Lecture Notes in Computer Science,* Vol. 393, Springer-Verlag, pp. 157–184.

[Ehrig et al. 89]

EHRIG, H., PEPPER, P., AND OREJAS, F. On recent trends in algebraic specification. In *16th ICALP* (Stresa, Italy, July 1989), *Lecture Notes in Computer Science,* Vol. 372, Springer-Verlag, pp. 262–288.

[Enderton 72]

ENDERTON, H. B. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.

[Futatsugi et al. 85]

FUTATSUGI, K., GOGUEN, J. A., JOUANNAUD, J.-P., AND MESEGUER, J. Principles of OBJ2. In *Symposium on Principles of Programming Languages* (1985), ACM, pp. 52–66.

[Ganzinger 83]

GANZINGER, H. Parameterized specifications: Parameter passing and implementation with respect to observability. *ACM Trans. Program. Lang. Syst. 5*, 3 (July 1983), 318–354.

[Gaudel and Moineau 88]

GAUDEL, M.-C., AND MOINEAU, T. A theory of software reusability. In *European Symposium on Programming* (Nancy, France, Mar. 1988), *Lecture Notes in Computer Science,* Vol. 300, Springer-Verlag, pp. 115–130.

[Gehani and McGettrick 86]

GEHANI, N., AND MCGETTRICK, A. D., Eds. *Software Specification Techniques.* Addison-Wesley, Wokingham, England, 1986.

[Giarratana et al. 76]

GIARRATANA, V., GIMONA, F., AND MONTANARI, U. Observability concepts in abstract data type specification. In 5th *MFCS* (Gdańsk, Sept. 1976), *Lecture Notes in Computer Science,* Vol. 45, Springer-Verlag, pp. 576–587.

[Gnaedig et al. 90]

GNAEDIG, I., KIRCHNER, C., AND KIRCHNER, H. Equational completion in order-sorted algebras. *Theoretical Comput. Sci. 72* (1990), 169–202.

[Goguen 73]

GOGUEN, J. A. Realization is universal. *Mathematical Systems Theory 6*, 4 (1973), 359–374.

[Goguen 78]

GOGUEN, J. A. Abstract errors for abstract data types. In *Formal Descriptions of Programming Concepts*, E. J. Neuhold, Ed. North-Holland, 1978, pp. 491–525.

[Goguen 87a]

GOGUEN, J. A. Modular algebraic specification of some basic geometrical constructions. *Artificial Intelligence 37* (1987), 123–153.

[Goguen 87b]

GOGUEN, J. A. One, none, a hundred thousand specification languages. Tech. Rep. CSLI-87-96, CSLI, Stanford University, Mar. 1987.

[Goguen 88a]

GOGUEN, J. A. Higher order functions considered unnecessary for higher order programming. Tech. Rep. SRI-CSL-88-1R, Computer Science Laboratory, SRI International, Apr. 1988.

[Goguen 88b]

GOGUEN, J. A. What is unification? A categorical view of substitution, equation and solution. Tech. Rep. CSLI-88-124, CSLI, Stanford University, Apr. 1988.

[Goguen 89]

GOGUEN, J. A. Principles of parameterized programming. In *Software Reusability*, T. J. Biggerstaff and A. J. Perlis, Eds. ACM Press, New York, 1989, pp. 159–225.

[Goguen and Burstall 80]

GOGUEN, J. A., AND BURSTALL, R. M. CAT, A system for the correct elaboration of correct programs from structured specifications. Tech. Rep. CSL-118, SRI International, Oct. 1980.

[Goguen and Burstall 83]

GOGUEN, J. A., AND BURSTALL, R. M. Introducing institutions. In *Proceedings, Logics of Programs Workshop* (Pittsburgh, June 1983), *Lecture Notes in Computer Science,* Vol. 164, Springer-Verlag, pp. 221–255.

[Goguen and Burstall 84a]

GOGUEN, J. A., AND BURSTALL, R. M. Some fundamental algebraic tools for the semantics of computation, Part 1: Comma categories, colimits, signatures and theories. *Theoretical Comput. Sci. 31*, 2 (1984), 175–209.

[Goguen and Burstall 84b]

GOGUEN, J. A., AND BURSTALL, R. M. Some fundamental algebraic tools for the semantics of computation, Part 2: Signed and abstract theories. *Theoretical Comput. Sci. 31*, 3 (1984), 263–295.

[Goguen and Burstall 85a]

GOGUEN, J. A., AND BURSTALL, R. M. Institutions: Abstract model theory for computer science. Tech. Rep. CSLI-85-30, SRI International, 1985.

[Goguen and Burstall 85b]

GOGUEN, J. A., AND BURSTALL, R. M. A study in the foundations of programming methodology: Specifications, institutions, charters and parchments. In *Proceedings, Workshop on Category Theory and Computer Programming* (Guildford, UK, Sept. 1985), *Lecture Notes in Computer Science,* Vol. 240, Springer-Verlag, pp. 313–333.

[Goguen and Meseguer 82]

GOGUEN, J. A., AND MESEGUER, J. Universal realization, persistent interconnection and implementation of abstract modules. In *9^{th}ICALP* (Aarhus, Denmark, July 1982), *Lecture Notes in Computer Science,* Vol. 140, Springer-Verlag, pp. 265–281.

[Goguen and Meseguer 84]

GOGUEN, J. A., AND MESEGUER, J. Completeness of many-sorted equational logic. Tech. Rep. CSLI-84-15, CSLI, Stanford University, Sept. 1984.

[Goguen and Meseguer 87a]

GOGUEN, J. A., AND MESEGUER, J. Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems. Tech. Rep. CSLI-87-92, CSLI, Stanford University, Mar. 1987.

[Goguen and Meseguer 87b]

GOGUEN, J. A., AND MESEGUER, J. Unifying functional, object-oriented and relational programming with logical semantics. In *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner, Eds. MIT Press, Cambridge, Massachusetts, 1987, pp. 417–477. Also available as Report No. CSLI-87-93, CSLI, Stanford University.

[Goguen and Meseguer 88]

GOGUEN, J. A., AND MESEGUER, J. Order-sorted algebra I: Equational deduction for multiple inheritance, polymorphism, and partial operations. Draft technical report, Computer Science Laboratory, SRI International, 1988.

[Goguen and Tardo 79]

GOGUEN, J. A., AND TARDO, J. J. An introduction to OBJ: A language for writing and testing formal algebraic specifications. In *Proceedings, Conference on Specifications of Reliable Software.* IEEE Computer Society Press, 1979, pp. 170–189. Also in [Gehani and McGettrick 86, pp. 391-419].

[Goguen and Winkler 88]

GOGUEN, J. A., AND WINKLER, T. Introducing OBJ3. Tech. Rep. SRI-CSL-88-9, SRI International, Aug. 1988.

[Goguen et al. 77]

GOGUEN, J. A., THATCHER, J. W., WAGNER, E. G., AND WRIGHT, J. B. Initial algebra semantics and continuous algebras. *J. ACM 24*, 1 (Jan. 1977), 68–95.

[Goguen et al. 78]

GOGUEN, J. A., THATCHER, J. W., AND WAGNER, E. G. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In *Data Structuring*, R. T. Yeh, Ed., *Current Trends in Programming Methodology,* Vol. IV. Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 80–149.

[Goguen et al. 85]

GOGUEN, J. A., JOUANNAUD, J.-P., AND MESEGUER, J. Operational semantics for order-sorted algebra. In *12th ICALP* (Nafplion, Greece, July 1985), *Lecture Notes in Computer Science,* Vol. 194, Springer-Verlag, pp. 221–231.

[Goldblatt 84]

GOLDBLATT, R. *Topoi: The Categorial Analysis of Logic.* North-Holland, Amsterdam, 1984.

[Grätzer 79]

GRÄTZER, G. *Universal Algebra.* Springer-Verlag, New York, 1979.

[Guttag 75]

GUTTAG, J. V. *The Specification and Application to Programming of Abstract Data Types.* PhD thesis, University of Toronto, 1975.

[Guttag 86]

GUTTAG, J. V. The Larch shared language. *IEEE Software 19*, 2 (Feb. 1986), 16–28.

[Guttag and Horning 78]

GUTTAG, J. V., AND HORNING, J. J. The algebraic specification of abstract data types. *Acta Inf. 10* (1978), 27–52.

[Guttag and Horning 86a]

GUTTAG, J. V., AND HORNING, J. J. A Larch shared language handbook. *Sci. Comput. Programming 6* (1986), 135–157.

[Guttag and Horning 86b]

GUTTAG, J. V., AND HORNING, J. J. Report on the Larch shared language. *Sci. Comput. Programming 6* (1986), 103–134.

[Guttag et al. 82]

GUTTAG, J. V., HORNING, J. J., AND WING, J. M. Some notes on putting formal specifications to productive use. *Sci. Comput. Programming 2* (1982), 53–68.

[Guttag et al. 85]

GUTTAG, J. V., HORNING, J. J., AND WING, J. M. Larch in five easy pieces. Tech. Rep. 5, DEC Systems Research Center, July 1985.

[Hansen 87]

HANSEN, H. The ACT-System: Experiences and future enhancements. In *Recent Trends in Data Type Specification* (Gullane, Scotland, Sept. 1987), *Lecture Notes in Computer Science,* Vol. 332, Springer-Verlag, pp. 113–130.

[Herrlich and Strecker 73]

HERRLICH, H., AND STRECKER, G. E. *Category Theory: An Introduction.* Allyn and Bacon, Boston, 1973.

[Hopcroft and Ullman 79]

HOPCROFT, J. E., AND ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[Horebeek and Lewi 89]

HOREBEEK, I. V., AND LEWI, J. *Algebraic Specifications in Software Engineering: An Introduction.* Springer-Verlag, Berlin, 1989.

[Hornung and Raulefs 80]

HORNUNG, G., AND RAULEFS, P. Terminal algebra semantics and retractions for abstract data types. In *7th ICALP* (Noordwijkerhout, Netherlands, July 1980), *Lecture Notes in Computer Science,* Vol. 85, Springer-Verlag, pp. 310–323.

[Hornung and Raulefs 81]

HORNUNG, G., AND RAULEFS, P. Initial and terminal algebra semantics of parameterized abstract data type specifications with inequalities. In *6th CAAP* (Genoa, Mar. 1981), *Lecture Notes in Computer Science,* Vol. 112, Springer-Verlag, pp. 224–237.

[Huet and Lang 78]

HUET, G., AND LANG, B. Proving and applying program transformations expressed with second-order patterns. *Acta Inf. 11* (1978), 31–55.

[Huet and Oppen 80]

HUET, G., AND OPPEN, D. C. Equations and rewrite rules: A survey. In *Formal Language Theory: Perspectives and Open Problems*, R. V. Book, Ed. Academic Press, New York, 1980, pp. 349–405.

[Hupbach 80]

HUPBACH, U. L. Abstract implementation of abstract data types. In *9th MFCS* (Rydzyna, Poland, Sept. 1980), *Lecture Notes in Computer Science,* Vol. 88, Springer-Verlag, pp. 291–304.

[Janssen and vanEmde Boas 82]

JANSSEN, T. M. V., AND VAN EMDE BOAS, P. Some observations on compositional semantics. In *Logic of Programs*, *Lecture Notes in Computer Science,* Vol. 131. Springer-Verlag, 1982, pp. 137–149.

[Kamin 83]

KAMIN, S. Final data types and their specification. *ACM Trans. Program. Lang. Syst. 5*, 1 (Jan. 1983), 97–123.

[Knuth 68]

KNUTH, D. E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison-Wesley, Reading, Massachusetts, 1968.

[Kock and Reyes 77]

KOCK, A., AND REYES, G. E. Doctrines in categorical logic. In *Handbook of Mathematical Logic*, J. Barwise, Ed. North-Holland Publishing Company, Amsterdam, 1977, pp. 283–313.

[Kutzler and Lichtenberger 83]

KUTZLER, B., AND LICHTENBERGER, F. *Bibliography on Abstract Data Types, Informatik-Fachberichte,* Vol. 68. Springer-Verlag, 1983.

[Lambek and Scott 86]

LAMBEK, J., AND SCOTT, P. J. *Introduction to Higher Order Categorical Logic.* Cambridge University Press, Cambridge, 1986.

[Lawvere 63]

LAWVERE, F. W. Functorial semantics of algebraic theories. *Proc. Nat. Acad. Sci., USA 50* (1963), 869–872.

[Lawvere 65]

LAWVERE, F. W. Algebraic theories, algebraic categories, and algebraic functors. In *Symposium on the Theory of Models.* North-Holland, 1965, pp. 413–418.

[Lawvere 75]

LAWVERE, F. W. Introduction. In *Model Theory and Topoi*, F. W. Lawvere, C. Maurer, and G. C. Wraith, Eds., *Lecture Notes in Mathematics,* Vol. 445. Springer-Verlag, 1975, pp. 3–14.

[Lehman et al. 84]

LEHMAN, M. M., STENNING, V., AND TURSKI, W. M. Another look at software design methodology. *ACM SIGSOFT Software Engineering Notes 9*, 2 (Apr. 1984), 38–53.

[Levy and Maibaum 82]

LEVY, M. R., AND MAIBAUM, T. S. E. Continuous data types. *SIAM J. Comput. 11*, 2 (May 1982), 201–216.

[Liskov and Berzins 79]

LISKOV, B. H., AND BERZINS, V. An appraisal of program specifications. In *Research Directions in Software Technology*, P. Wegner, Ed. MIT Press, 1979, pp. 276–301. Also in [Gehani and McGettrick 86, pp. 3-23].

[Liskov and Zilles 75]

LISKOV, B., AND ZILLES, S. Specification techniques for data abstraction. *IEEE Trans. Softw. Eng. SE-1* (1975), 7–19.

[Mac Lane 71]

MAC LANE, S. *Categories for the Working Mathematician.* Springer-Verlag, New York, 1971.

[Mac Lane and Birkhoff 67]

MAC LANE, S., AND BIRKHOFF, G. *Algebra.* The Macmillan Company, New York, 1967.

[Mahr and Makowsky 83]

MAHR, B., AND MAKOWSKY, J. A. An axiomatic approach to semantics of specification and languages. In *Conference on Theoretical Computer Science*, *Lecture Notes in Computer Science,* Vol. 145. Springer-Verlag, Dortmund, Jan. 1983, pp. 211–219.

[Mahr and Makowsky 84]

MAHR, B., AND MAKOWSKY, J. A. Characterizing specification languages which admit initial semantics. *Theoretical Comput. Sci. 31* (1984), 49–59.

[Maibaum and Lucena 80]

MAIBAUM, T. S. E., AND LUCENA, C. J. Higher order data types. *J. Comput. Inf. Sci. 9*, 1 (1980), 31–53.

[Maibaum and Turski 84]

MAIBAUM, T., AND TURSKI, W. On what exactly is going on when software is developed step-by-step. In *Proceedings, 7th Int. Conf. Softw. Eng.* (1984), IEEE, pp. 528–533.

[Makkai and Reyes 77]

MAKKAI, M., AND REYES, G. E. *First Order Categorical Logic, Lecture Notes in Mathematics,* Vol. 611. Springer-Verlag, 1977.

[Makowsky 87]

MAKOWSKY, J. A. Why Horn formulas matter in Computer Science: Initial structures and generic examples. *J. Comput. Syst. Sci. 34* (1987), 266–292.

[Manes 76]

MANES, E. G. *Algebraic Theories.* Springer-Verlag, New York, 1976.

[Meseguer 88]

MESEGUER, J. Relating models of polymorphism. Tech. Rep. CSLI-88-133, CSLI, Stanford University, Oct. 1988.

[Meseguer 89]

MESEGUER, J. General logics. In *Logic Colloquium'87*, H.-D. Ebbinghaus et al., Eds. North-Holland, 1989, pp. 275–329.

[Meseguer and Goguen 84]

MESEGUER, J., AND GOGUEN, J. A. Initiality, induction, and computability. In *Algebraic Methods in Semantics*, M. Nivat and J. Reynolds, Eds. Cambridge University Press, 1984, pp. 459–541.

[Milner 77]

MILNER, R. Fully abstract models of typed λ-calculi. *Theoretical Comput. Sci. 4* (1977), 1–22.

[Möller 85]

MÖLLER, B. On the algebraic specification of infinite objects—Ordered and continuous models of algebraic types. *Acta Inf. 22* (1985), 537–578.

[Möller 86]

MÖLLER, B. Algebraic specifications with higher order operators. In *IFIP TC2/WG2.1 Working Conference of Program Specification and Transformation* (Bad Tölz, FRG, Apr. 1986), L. G. L. T. Meertens, Ed., North-Holland, pp. 367–398.

[Möller 87]

MÖLLER, B. *Higher-Order Algebraic Specifications.* Habilitationsschrift, fakultät für mathematik und informatik, Technische Universität München, Feb. 1987.

[Möller et al. 87]

MÖLLER, B., TARLECKI, A., AND WIRSING, M. Algebraic specifications of reachable higher-order algebras. In *Recent Trends in Data Type Specification* (Gullane, Scotland, Sept. 1987), *Lecture Notes in Computer Science,* Vol. 332, Springer-Verlag, pp. 154–169.

[Möller et al. 88]

MÖLLER, B., TARLECKI, A., AND WIRSING, M. Algebraic specifications with built-in domain constructions. In *13th CAAP* (Nancy, France, Mar. 1988), *Lecture Notes in Computer Science,* Vol. 299, Springer-Verlag, pp. 132–148.

[Nivela and Orejas 87]

NIVELA, M. P., AND OREJAS, F. Initial behaviour semantics for algebraic specifications. In *Recent Trends in Data Type Specification* (Gullane, Scotland, Sept. 1987), *Lecture Notes in Computer Science,* Vol. 332, Springer-Verlag, pp. 184–207.

[Orejas et al. 88]

OREJAS, F., NIVELA, M. P., AND EHRIG, H. Semantical constructions for categories of behavioural specifications. In *Categorical Methods in Computer Science, with Aspects from Topology* (Berlin, Sept. 1988), *Lecture Notes in Computer Science,* Vol. 393, Springer-Verlag, pp. 220–243.

[Padawitz 87]

PADAWITZ, P. Parameter-preserving data type specifications. *J. Comput. Syst. Sci. 34* (1987), 179–209.

[Padawitz 88]

PADAWITZ, P. *Computing in Horn Clause Theories, EATCS Monographs on Theoretical Computer Science,* Vol. 16. Springer-Verlag, Berlin, 1988.

[Parisi-Presicce 88]

PARISI-PRESICCE, F. Product and iteration of module specifications. In *13th CAAP* (Nancy, France, Mar. 1988), *Lecture Notes in Computer Science,* Vol. 299, Springer-Verlag, pp. 149–164.

[Parsaye-Ghomi 82]

PARSAYE-GHOMI, K. *Higher Order Abstract Data Types.* PhD thesis, Univ. of California, Los Angeles, 1982.

[Pierce 88]

PIERCE, B. C. A taste of category theory for computer scientists. Tech. Rep. CMU-CS-88-203, Computer Science Dept, Carnegie Mellon University, Pittsburgh, 1988.

[Pitt et al. 85]

PITT, D., ABRAMSKY, S., POIGNÉ, A., AND RYDEHEARD, D., Eds. *Category Theory and Computer Programming, Tutorial and Workshop* (Guildford, UK, Sept. 1985), *Lecture Notes in Computer Science,* Vol. 240, Springer-Verlag.

[Poigné 86]

POIGNÉ, A. On specifications, theories, and models with higher types. *Inf. and Comput. 68* (1986), 1–46. A summary appeared in STACS 84, *LNCS 166,* pp. 174–185.

[Poigné 90]

POIGNÉ, A. Parameterization for order-sorted algebraic specification. *J. Comput. Syst. Sci. 40* (1990), 229–268.

[Reichel 80]

REICHEL, H. Initially-restricting algebraic theories. In *9th MFCS* (Rydzyna, Poland, Sept. 1980), *Lecture Notes in Computer Science,* Vol. 88, Springer-Verlag, pp. 504–514.

[Reichel 81]

REICHEL, H. Behavioural equivalence—A unifying concept for initial and final specification methods. In *Proceedings, Third Hungarian Computer Science Conference* (Budapest, Jan. 1981), Akademiai Kiado, pp. 27–39.

[Reichel 87]

REICHEL, H. *Initial Computability, Algebraic Specifications, and Partial Algebras.* Oxford University Press, 1987.

[Rydeheard and Burstall 84]

RYDEHEARD, D. E., AND BURSTALL, R. M. Monads and theories: A survey for computation. In *Algebraic Methods in Semantics*, M. Nivat and J. Reynolds, Eds. Cambridge University Press, 1984, pp. 575–605.

[Rydeheard and Burstall 88]

RYDEHEARD, D., AND BURSTALL, R. M. *Computational Category Theory.* Prentice-Hall, 1988.

[Sannella 82]

SANNELLA, D. T. *Semantics, Implementation, and Pragmatics of Clear, A Program Specification Language.* PhD thesis, Dept. of Computer Science, University of Edinburgh, July 1982.

[Sannella 84]

SANNELLA, D. T. A set-theoretic semantics for Clear. *Acta Inf. 21* (1984), 443–472.

[Sannella and Tarlecki 85]

SANNELLA, D., AND TARLECKI, A. Extended ML: An institution-independent framework for formal program development. In *Proceedings, Workshop on Category Theory and Computer Programming* (Guildford, UK, Sept. 1985), *Lecture Notes in Computer Science,* Vol. 240, Springer-Verlag, pp. 364–389. Also, technical report ECS-LFCS-86-16, University of Edinburgh, December 1986.

[Sannella and Tarlecki 87]

SANNELLA, D., AND TARLECKI, A. On observational equivalence and algebraic specification. *J. Comput. Syst. Sci. 34* (1987), 150–178.

[Sannella and Tarlecki 88a]

SANNELLA, D., AND TARLECKI, A. Specifications in an arbitrary institution. *Inf. and Comput. 76* (1988), 165–210. Preliminary version in Semantics of Data Types, *LNCS 123,* pp. 337–356.

[Sannella and Tarlecki 88b]

SANNELLA, D., AND TARLECKI, A. Towards formal development of programs from algebraic specifications: Implementations revisited. *Acta Inf. 25* (1988), 233–281.

[Sannella and Tarlecki 89]

SANNELLA, D., AND TARLECKI, A. Toward formal development of ML programs: Foundations and methodology. In *TAPSOFT89, vol. 2* (Barcelona, Spain, Mar. 1989), *Lecture Notes in Computer Science,* Vol. 352, Springer-Verlag, pp. 375–389.

[Sannella and Wirsing 82]

SANNELLA, D., AND WIRSING, M. Implementation of parameterized specifications. In *9th ICALP* (Aarhus, Denmark, July 1982), *Lecture Notes in Computer Science,* Vol. 140, Springer-Verlag, pp. 473–488. Extended abstract.

[Sannella and Wirsing 83]

SANNELLA, D., AND WIRSING, M. A kernel language for algebraic specification and implementation. In *Proceedings, International Foundations of Computation Theory Conference* (Borgholm, Sweden, Aug. 1983), *Lecture Notes in Computer Science,* Vol. 158, Springer-Verlag, pp. 413–427.

[Schubert 72]

SCHUBERT, H. *Categories.* Springer-Verlag, Berlin, 1972.

[Simon 70]

SIMON, H. A. *The Sciences of the Artificial.* MIT Press, Cambridge, Massachusetts, 1970.

[Srinivas 90]

SRINIVAS, Y. V. Category theory: Definitions and examples. Tech. Rep. 90-14, Dept. of ICS, University of California, Irvine, Feb. 1990.

[Tarlecki 85a]

TARLECKI, A. Bits and pieces of the theory of institutions. In *Proceedings, Workshop on Category Theory and Computer Programming* (Guildford, UK, Sept. 1985), *Lecture Notes in Computer Science,* Vol. 240, Springer-Verlag, pp. 334–363.

[Tarlecki 85b]

TARLECKI, A. On the existence of free models in abstract algebraic institutions. *Theoretical Comput. Sci. 37* (1985), 269–304.

[Tarlecki and Wirsing 86]

TARLECKI, A., AND WIRSING, M. Continuous abstract data types. *Fundamenta Informaticae IX* (1986), 95–125.

[Tarlecki et al. 88]

TARLECKI, A., BURSTALL, R. M., AND GOGUEN, J. A. Some fundamental algebraic tools for the semantics of computation, Part 3: Indexed categories. Tech. Rep. ECS-LFCS-88-60, University of Edinburgh, July 1988.

[Thatcher et al. 82]

THATCHER, J. W., WAGNER, E. G., AND WRIGHT, J. B. Data type specification: Parameterization and the power of specification techniques. *ACM Trans. Program. Lang. Syst. 4*, 4 (Oct. 1982), 711–732. Earlier version in 10th SIGACT Symposium, San Diego, May 1978.

[Turski and Maibaum 87]

TURSKI, W. M., AND MAIBAUM, T. S. E. *The Specification of Computer Programs.* Addison-Wesley, 1987.

[Wagner 85]

WAGNER, E. G. Categorical semantics, or extending data types to include memory. In *Recent Trends in Data Type Specification*, H. J. Kreowski, Ed., *Informatik-Fachberichte,* Vol. 116. Springer-Verlag, 1985, pp. 1–21.

[Wagner 89]

WAGNER, E. An algebraically specified language for data directed design. In *Conference on Algebraic Methodology and Software Technology* (Iowa City, May 1989), pp. 145–163.

[Wand 79]

WAND, M. Final algebra semantics and data type extensions. *J. Comput. Syst. Sci. 19* (1979), 27–44.

[Weber and Ehrig 86]

WEBER, H., AND EHRIG, H. Specification of modular systems. *IEEE Trans. Softw. Eng. SE-12*, 7 (July 1986), 784–797.

[Wells and Barr 87]

WELLS, C., AND BARR, M. The formal description of data types using sketches. In *Mathematical Foundations of Programming Language Semantics, 3rd Workshop* (Tulane, Apr. 1987), *Lecture Notes in Computer Science,* Vol. 298, Springer-Verlag, pp. 490–527.

[Wing 87]

WING, J. M. Writing Larch interface language specifications. *ACM Trans. Program. Lang. Syst. 9*, 1 (Jan. 1987), 1–24.

[Wing 89]

WING, J. M. Specifying Avalon objects in Larch. In *TAPSOFT89, vol. 2* (Barcelona, Spain, Mar. 1989), *Lecture Notes in Computer Science,* Vol. 352, Springer-Verlag, pp. 61–80.

[Wirsing 86]

WIRSING, M. Structure algebraic specifications: A kernel language. *Theoretical Comput. Sci. 42* (1986), 123–249. A slight revision of his Habilitationsschrift, Technische Universität München, 1983.

[Wirsing and Broy 80]

WIRSING, M., AND BROY, M. Abstract data types as lattices of finitely generated models. In *9th MFCS* (Rydzyna, Poland, Sept. 1980), *Lecture Notes in Computer Science,* Vol. 88, Springer-Verlag, pp. 673–685.

[Wirsing and Broy 82]

WIRSING, M., AND BROY, M. An analysis of semantic models for algebraic specifications. In *Theoretical Foundations of Programming Methodology*, M. Broy and G. Schmidt, Eds. D. Reidel Pub. Co., 1982, pp. 351–413.

[Wirsing et al. 83]

WIRSING, M., PEPPER, P., PARTSCH, H., DOSCH, W., AND BROY, M. On hierarchies of abstract data types. *Acta Inf. 20* (1983), 1–33.

[Wirsing et al. 88]

WIRSING, M., HENNICKER, R., AND BREU, R. Reusable specification components. In *16thMFCS* (Nancy, France, Aug. 1988), *Lecture Notes in Computer Science,* Vol. 324, Springer-Verlag, pp. 121–137.

[Wirsing et al. 89]

WIRSING, M., HENNICKER, R., AND STABL, R. MENU—An example for the systematic reuse of specifications. Tech. Rep. MIP-8930, Fakultät für Mathematik und Informatik, Universität Passau, Oct. 1989.

[Wirth 71]

WIRTH, N. Program development by stepwise refinement. *Commun. ACM 14*, 4 (Apr. 1971), 221–227.

[Zilles 79]

ZILLES, S. N. An introduction to data algebras. In *Abstract Software Specifications* (Copenhagen, 1979), D. Bjorner, Ed., *Lecture Notes in Computer Science,* Vol. 86, Springer-Verlag, pp. 248–272.