An Automaton-Theoretic View of Algebraic Specifications

by

Elad Lahav

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2005

© Elad Lahav 2005

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

We compare two methods for software specification: *algebraic specifications* and automata. While algebraic specifications have been around since the 1970s and have been studied extensively, specification by automata is relatively new. Its origins are in another veteran method called *trace assertions*, which considers a software module as a set of traces, that is, a sequences of function executions. A module is specified by a set of canonical traces and an equivalence relation matching one of the canonical traces to each non-canonical trace. It has been recently shown that trace assertions is an equivalent method to specification by automata. In continuation of this work on trace assertions and automata, we study how automata compare with algebraic specifications. We prove that every specification using an automaton can be converted into an algebraic specification describing the same abstract data type. This conversion utilises a set of canonical words, representing states in the automaton. We next consider varieties of monoids as a heuristic for obtaining more concise algebraic specifications from automata. Finally, we discuss the opposite conversion of algebraic specifications into automata. We show that, while an automaton always exists for every abstract data type described by an algebraic specification, this automaton may not be finitely describable and therefore may not be considered as a viable method for software specification.

Acknowledgements

I would like to thank my supervisor, Prof. John Brzozowski, for his invaluable help in writing this thesis, as well as for supporting my work, both academically and financially.

I would also like to thank the University of Waterloo, the School of Computer Science, for presenting me with the opportunity and the means to pursue my interests.

Finally, I would like to thank the members of the committee: Daniel Berry, Stanley Burris, John Thistle and Richard Trefler for their advice.

To Shelly and Alma

Contents

List of Figures

Chapter 1

Introduction

1.1 The Case For Formal Methods

The increasing complexity of computer programmes makes development of correct, flaw-free software more and more difficult. The emergence of relatively powerful and cheap computers in the 1960s, as well as the development of high-level programming languages and tools, enabled, for the first time, the creation of large-scale software systems. Design techniques were then applied for making these systems more modular, so that an ample project could be handled by dividing tasks among several developers.

This evolution in software development, however, has led to the introduction of new problems with which early programmers were not familiar [8]. The complexity of these new software systems meant that it was ever harder to ensure their correctness. The modularisation techniques implied that programmers working on different sub-projects had to interact to ensure the compatibility and interoperability of the modules under their responsibility. These problems led to the introduction of a new field in computer science, software engineering, which attempts to bring the rigour and formality of engineering into the somewhat "artistic" world of software development.

One of the important tools introduced by software engineering is *software specification*. The specification stage is supposed to bridge the gap in the development process between the requirements (the customer's point of view) and the design (the programmer's point of view) [29].

Specifications may come in different formats. Most notably, a specification is often given using a natural language, as a document provided to the designer. Such documents, however, suffer from many drawbacks, some of which were identified by Meyer [29] as the "seven sins of the specifier": noise (excessive discussion obfuscating the important details), silence (omission of important issues), overspecification, contradiction, ambiguity, forward reference and wishful thinking. But the greatest shortcoming of natural language specifications is the lack of any *verification* tools. This problem is the result of the informality of such specifications, as well as the absence of a mathematical basis.

Formal methods have been designed to overcome the problems presented by informal specification techniques. These methods are mathematical techniques for describing software properties [36]. The introduction of mathematics, and especially of set theory, logic and abstract algebra, provides the basis for a theorem-proving system, which can be used to verify the correctness of the specified modules. Moreover, since mathematics is, by its very nature, a rigorous discipline, formal methods, when applied correctly, remove any ambiguity from software specifications.

And yet, formal methods have not been generally accepted by the software development community [4]. Despite several recorded success stories [3], and despite a growing inclination in the hardware industry for using mathematical verification techniques (see, for example [32] and [17]), it seems that programmers in general, and those outside the academia in particular, are refusing to adopt formal methods.

Why is that so? The answer to this question is beyond the scope of this work. It can be mentioned, however, that, while working on formal methods as part of this thesis, the author has encountered several phenomena that may give some explanation for the absence of formal methods in real-world programming. Among those are cumbersome and confusing notation, lack of sufficient mathematical background and methods which, while appropriate for small examples, do not seem to scale well when the modules to be specified become more complicated. This last caveat in particular seems to be a recurring problem in the more theoretic work on formal methods (including this thesis), and future work may benefit from a larger, and perhaps standardised, set of complex modules against which new methods can be tested.

1.2 On Abstraction and Automata

It was noted earlier that a specification precedes the design stage in the development process. As such, a specification is not supposed to impose any design decisions on the programmer, or, in other words, a specification should be as *abstract* as possible. But what makes a specification abstract? This question has been the topic of much debate among theoreticians of formal methods.

Consider, for example, the following specification of a queue:

> A queue is a linked list with pointers to its first and last nodes. When an item is added to the queue, a new node is created to contain the item and linked after the last item in the list. When an item is removed from the queue, the first node is erased. The "front" operation returns the item held by the first node.

This specification clearly includes several design decisions. A queue may very well be implemented in many different ways, such as using an array instead of a linked list, or by linking a new item to the head of the list rather to its tail. The only trait that describes a queue, and therefore the only requirement that should be imposed by a specification, is the "first-in, first-out" (FIFO) principle. A good specification is thus one that is able to state this requirement, without giving any other restriction, either explicitly or implicitly.

While a natural language specification can certainly be used in this case, we are still left with the question of a formal method that can express the requirements without adding new ones in the process of specification. Moreover, such a method has to be expressive enough so that a specification can be constructed for any kind of module.

Lately it has been suggested that automata can provide a useful model for software specification [6]. The reasoning behind such a model is that a software module is often treated as a state machine by different specification techniques (see [30], for example). If a state machine is already implied, then why not use automata, the mathematical model for such machines, which are well-studied in computer science?

The major objection to the use of automata as a formal method for software specification is the claim that an automaton already gives a design for the module. This notion, however, may be the result of a confusion on the role of automata in the development process, arising from the widespread usage of automata in the implementation of software modules. Writing a specification as an automaton does not mean that the module needs to be implemented as one. Moreover, as Brzozowski and Jürgensen have shown [6], the attempt to stay away from automata sometimes results in what amounts to only a cumbersome description of such machines.

On the other hand, the use of automata gives the developer a concrete model with which the programmer can compare the resulting implementation. Furthermore, automata, because of their ubiquity in software design and implementation, may serve as a good basis for automatic tools for software specification and verification.

The question as to whether automata are abstract enough remains open, though. We shall see in Chapter 3 that algebraic specifications provide a mathematical definition of abstraction, by whose criteria an automaton is only one possible implementation of an abstract data type. While this definition, which uses the concept of an isomorphism class of algebras, seems to be correct, it has only theoretical importance. This class still needs to be specified using one of its elements, and, where appropriate, an automaton is as good a representative as any other class member.

1.3 About This Thesis

Previous work on formal methods can be roughly divided into two categories. The first is theoretical research, which focuses on the properties of software modules as mathematical models. Examples of such work include the trace assertion method, originally by Bartussek and Parnas [1], and algebraic specifications (see Chapter 3 for more information).

The second category involves a more practical approach towards the development of tools for software specification and verification. Work of this kind has resulted in the specification languages Z [33], VDM [26] and Larch [20], as well as in theorem proving tools, such as the Larch Prover (LP) [11]. Usually these tools are derived from theoretical research; for example, the specification language OBJ [16] emerged from the theory of algebraic specifications.

This thesis clearly belongs to the first of the two categories. It attempts to draw the similarities, and differences, between two fields that are closely related to abstract algebra, namely algebraic automaton theory and the theory of algebraic specifications. As such, it is more interested in the mathematical properties of the models then in their applicability to real-world software development.

The current research started as an extension to the work by Brzozowski (who supervised the thesis) and Jürgensen on Parnas's trace-assertion method [6]. Their goal was to show that this method amounted to little more than a specification of software module using infinite, deterministic Mealy machines. The natural question rising from the cited paper was whether the same result can be applied to other specification techniques, or, in other words, if automata can be used as a universal technique for software specifications.

The method chosen for research was the theory of algebraic specifications. It was selected mainly because of its well-studied mathematical basis, as laid down by Goguen *et al.* in several papers during the late 1970s (among those are [15] and [14]). Moreover, this theoretical foundation turned out to be derived from Birkhoff's original work on abstract algebra [2], which is also related to algebraic automaton theory.

Our results are not decisive. It seems that algebraic specifications are both more abstract and more powerful than the selected automaton model. However, much work was left outside the scope of this thesis, such as parameterised specifications, terminal and loose semantics, as well as different automaton models. It is therefore obvious that more research may come up with new, or at least clearer, conclusions.

This thesis is structured as follows: The first two chapters give some background material, with Chapter 2 dedicated to algebraic automaton theory and Chapter 3 describing the initial algebra approach to algebraic specifications. Chapter 4 shows how to construct a correct and complete algebraic specification from an automaton, a result which applies the work of Brzozowski and Jürgensen in [5]. Chapter 5 shows how the specifications in Chapter 4 can be made simpler and more elegant by using properties of varieties of transformation monoids. Finally, Chapter 6 depicts some initial results on the transformation of algebraic specifications into automata, and discusses the limitations of this method. It is this chapter that, we believe, would benefit most from future research.

Chapter 2

Algebraic Automaton Theory

2.1 Introduction

The study of machine models is among the oldest fields in computer science. Its origins lie in the attempt by early computer scientists to capture the properties of real-world machines using simple, yet accurate, mathematical models. This work has culminated in the theories of automata, computation and complexity, which are today considered to be cornerstones of modern computer science. It has also given rise to numerous practical applications of machine models, such as pattern recognition, language parsing, finite state controllers and much more.

Though originally inspired by actual machines, machine models were also studied as purely mathematical entities. It has been observed that some models are strongly related to algebraic structures, such as semigroups and monoids, an observation which led to the development of algebraic automaton theory.[1]

This chapter begins with some concepts in abstract algebra required for the development of an algebraic theory for machine models. It then describes some machine models which will be studied later in relation to algebraic specifications.

Most of the definitions, examples and results in the following sections are adapted from [12] and [24]. References are given only when other sources are used.

[1]Though the common name is "*automata* theory," we shall follow the English convention of using the singular form before "theory" (e.g., graph theory, number theory, etc.). This ubiquitous mistake has been pointed out by Eilenberg and others, though, for some reason, has persisted throughout the years.

2.2 Semigroups and Monoids

An *alphabet* is a non-empty set Σ of *symbols* (or *letters*). A *word over* Σ is a sequence of symbols from the set Σ. If u and v are words over Σ, then their *concatenation* uv is also a word over Σ. The *empty word* ϵ is the word that contains no symbols. The *length* of the word u, denoted by $|u|$, is 0 if $u = \epsilon$, and $|v| + 1$ if $u = v\sigma$, where v is a word over Σ and σ is a symbol.

Definition 2.2.1. Let S be a set. An *associative binary operator* on S is a function $f : S \times S \to S$, such that for all $s_1, s_2, s_3 \in S$, $f(f(s_1, s_2), s_3) = f(s_1, f(s_2, s_3))$. For every pair of elements $s_1, s_2 \in S$, the image $f(s_1, s_2)$ is called the *product* of these elements.

A binary operator is often written using infix notation: if $*$ is such an operator and s_1, s_2 are set elements, the notation $s_1 * s_2$ is equivalent to writing $*(s_1, s_2)$.

Definition 2.2.2. A *semigroup* is a pair $(S, \cdot)$, where S is a set and $\cdot : S \times S \to S$ is an associative binary operator.

As a matter of convenience, we usually omit the $\cdot$ operator symbol when combining two elements of a semigroup. That is, the product $s_1 \cdot s_2$ is written as $s_1 s_2$. Other operator symbols are not omitted.

Definition 2.2.3. A semigroup $(S, \cdot)$ is called a *monoid* if S contains a *unit element* $e \in S$, such that $se = es = s$ for all $s \in S$. A monoid with a set S, an operator $\cdot$ and a unit element $e \in S$ is written as $(S, \cdot, e)$.

Example 2.2.1. Let $\mathbb{N}^+$ denote the set of natural numbers, not including 0 (that is, $\mathbb{N}^+ = \{1, 2, 3, ...\}$). Then $(\mathbb{N}^+, +)$, where $+$ denotes the addition operation on natural numbers, is a semigroup. However, $(\mathbb{N}^+, +)$ is not a monoid: there is no number $e \in (\mathbb{N}^+, +)$ such that for all $n \in \mathbb{N}^+$, $e + n = n + e = n$. Using the multiplication operation $\times$, on the other hand, we get the monoid $(\mathbb{N}^+, \times, 1)$. ■

Definition 2.2.4. For every alphabet Σ, the set of all non-empty words over Σ with the concatenation operator forms a semigroup, called the *free semigroup generated by* Σ. This semigroup is denoted by Σ^+. Similarly, the set of all words over Σ, including the empty word ϵ, with the same operator, forms the *free monoid generated by* Σ, denoted by Σ^*.

A homomorphism is a mapping between two algebraic structures that preserves certain traits of these structures. In this chapter we are interested only in such structure-preserving mappings between semigroups and monoids. Later we shall also encounter other kinds of homomorphisms.

Definition 2.2.5. Let $(S, \cdot)$ and $(T, *)$ be semigroups. A function $h : S \to T$ is called a *semigroup homomorphism* if

$$h(a) * h(b) = h(a \cdot b) \quad \text{for all } a, b \in S$$

The function $h : S \to T$ is an *isomorphism* if it is a bijective homomorphism (that is, if h is a one-to-one mapping onto the set T).

A homomorphism between monoids is defined similarly to a semigroup homomorphism, with the additional requirement that the unit element of the first monoid is mapped to the unit element of the second monoid:

Definition 2.2.6. Let $(S, \cdot, e_S)$ and $(T, *, e_T)$ be monoids. A function $h : S \to T$ is a *monoid homomorphism* if

- h is a semigroup homomorphism
- $h(e_S) = e_T$

Example 2.2.2. Let $\mathbb{R}, \mathbb{R}^+$ be the sets of real numbers and positive real numbers, respectively, and let $\cdot, +$ be the operations of multiplication and addition on real numbers, respectively. The function $\log : \mathbb{R}^+ \to \mathbb{R}$ is a semigroup homomorphism from $(\mathbb{R}^+, \cdot)$ to $(\mathbb{R}, +)$, due to the following property of logarithms:

$$\log(ab) = \log(a) + \log(b) \quad \forall a, b \in \mathbb{R}^+$$

Moreover, since log is a bijection from $\mathbb{R}^+$ to $\mathbb{R}$, the semigroups $(\mathbb{R}^+, \cdot)$ and $(\mathbb{R}, +)$ are isomorphic. ■

Monoid homomorphisms whose domain is a free monoid over some set Σ have a special property: the homomorphism is a unique extension of a function assigning set elements from Σ to elements in the range monoid. This property will enable us to explore the nature of such homomorphism by obtaining results on the underlying assignment function.

Definition 2.2.7. Let Σ be a set, let $(M, *, e)$ be a monoid, and let $f : \Sigma \to M$ be a function. The *homomorphic extension* of f is a function $f^* : \Sigma^* \to M$ defined recursively:

- $f^*(\epsilon) = e$

- $f^*(w\sigma) = f^*(w) * f(\sigma)$ for all $\sigma \in \Sigma, w \in \Sigma^*$

Proposition 2.2.1. *The homomorphic extension of f is unique.*

Proof. Trivial, as the definition constructs the homomorphic extension function deterministically. □

Theorem 2.2.1. *Let Σ be a set, let $(M, *, e)$ be a monoid, and let $f : \Sigma \to M$ be a function. Then*

1. *The homomorphic extension $f^* : \Sigma^* \to M$ is a monoid homomorphism.*
2. *Every monoid homomorphism $h : \Sigma^* \to M$ is the homomorphic extension of some function from Σ to M.*

Proof.

1. First, observe that $f^*(\epsilon) = e$ by the definition of f^*. Let $u, v \in \Sigma^*$. We prove that $f^*(uv) = f^*(u) * f^*(v)$ by induction on $|v|$. If $v = \epsilon$, then $f^*(uv) = f^*(u\epsilon) = f^*(u) = f^*(u) * e = f^*(u) * f^*(v)$. Assume the hypothesis holds when $|v| = k$ for some $k \geq 0$. Let $v = w\sigma$ for $w \in \Sigma^*, \sigma \in \Sigma$ where $|w| = k$. Then $f^*(uv) = f^*(uw\sigma) = f^*(uw) * f(\sigma)$. By the induction hypothesis, $f^*(uw) * f(\sigma) = (f^*(u) * f^*(w)) * f(\sigma)$, and, since $*$ is associative, $(f^*(u) * f^*(w)) * f(\sigma) = f^*(u) * (f^*(w)) * f(\sigma))$. Following the definition of f^*, $f^*(u) * (f^*(w) * f(\sigma)) = f^*(u) * f^*(v)$.

2. Let $h : \Sigma^* \to M$ be a monoid homomorphism. Define a function $f : \Sigma \to M$ as $f(\sigma) = h(\sigma)$ for all $\sigma \in \Sigma$. Then $h(\epsilon) = e$ (since h is a monoid homomorphism), and

$$\begin{aligned} h(w\sigma) &= h(w) * h(\sigma) && (h \text{ is a semigroup homomorphism}) \\ &= h(w) * f(\sigma) && (\text{definition of } f) \end{aligned}$$

which proves that h is the unique homomorphic extension of f.

□

Definition 2.2.8. Let S be a set. A relation $\sim \subseteq S \times S$ is an *equivalence relation* if the following conditions hold for all $a, b, c \in S$:

1. $a \sim a$ (reflexivity)
2. $a \sim b \Rightarrow b \sim a$ (symmetry)
3. $a \sim b$ and $b \sim c \Rightarrow a \sim c$ (transitivity)

An equivalence relation partitions a set into one or more distinct subsets, known as *equivalence classes*. For each element $s \in S$, we denote by $[s]$ the unique equivalence class containing s. If the equivalence relation is not obvious from the context, we shall subscript $[s]$ with a symbol indicating the relevant relation (e.g., $[s]_\sim$ for the relation $\sim$).

Definition 2.2.9. Let $(S, \cdot)$ be a semigroup. A relation $\sim$ on S is a *left congruence* (resp. a *right congruence*) if $\sim$ is an equivalence relation on S and for all $a, b, c \in S$, $a \sim b$ implies $ca \sim cb$ (resp. $a \sim b$ implies $ac \sim bc$).

A relation $\sim$ on S is a *congruence* if it is both a left and a right congruence.

Proposition 2.2.2. *Let $(S, \cdot)$ and $(T, *)$ be semigroups, and let $f : S \to T$ be a semigroup homomorphism. The relation $\sim$ defined as*

$$a \sim b \Leftrightarrow f(a) = f(b) \quad \textit{for all } a, b \in S$$

is a congruence relation on S.

Proof. Reflexivity, symmetry and transitivity follow immediately from the fact that f is a function and from the validity of the same conditions on the equality relation $=$ on T.

Assume that $a \sim b$ for some $a, b \in S$. Then

$$\begin{aligned} f(ac) &= f(a) * f(c) && \text{(def. of semigroup homomorphism)} \\ &= f(b) * f(c) && \text{(assumption)} \\ &= f(bc) \end{aligned}$$

which implies that $ac \sim bc$ as required. The equivalence of ca and cb is proved similarly. □

From now on we shall identify the semigroup $(S, \cdot)$ with the set S. The meaning of the symbol S should be clear from the context. The same convention applies to monoids.

Lemma 2.2.1. *Let M be a monoid, let $m_1, m_2 \in M$ and let $\sim$ be a congruence relation on M. Then $m_1 \sim m_2$ if and only if $mm_1 \sim mm_2$ for all $m \in M$.*

Proof. Assume that $m_1 \sim m_2$. Then, by the congruence property, $mm_1 \sim mm_2$ for all $m \in M$. Conversely, if $mm_1 \sim mm_2$ for all $m \in M$, then, specifically, this equality holds for the unit element $e \in M$. Therefore $em_1 \sim em_2$, which implies that $m_1 \sim m_2$. $\square$

Definition 2.2.10. Let S be a semigroup, and let $\sim$ be a congruence on S. Define the set $S/\!\sim$ to be the set of all equivalence classes defined by $\sim$ on S, and define a binary operator $* : S/\!\sim \to S/\!\sim$ by

$$[a] * [b] = [ab] \quad \text{for all } a, b \in S$$

Then $(S/\!\sim, *)$ is a semigroup, called the *quotient semigroup of S with respect to $\sim$*.

Definition 2.2.11. Let S be a semigroup and let $\sim$ be a congruence relation on S. The *natural homomorphism of S onto $S/\sim$* is a semigroup homomorphism $nat : S \to S/\!\sim$ defined as

$$nat(a) = [a] \quad \text{for all } a \in S$$

2.3 Semiautomata

In this section we describe a primitive machine model, called a *semiautomaton*, which captures the definition of a pure state machine. A state machine is associated with a set of internal values, or states, one of which is the current state at each moment. The machine reacts to external stimuli by changing its current state to some other value in the set.

Definition 2.3.1. A *deterministic initialised semiautomaton* (or simply a *semiautomaton*) is a quadruple $\mathscr{S} = (Q, \Sigma, \delta, q_0)$, where Q is a set (of states), Σ is a set (called the alphabet of the semiautomaton), $\delta : Q \times \Sigma \to Q$ a transition function and $q_0 \in Q$ an initial state. A semiautomaton will be referred to as *finite* if the sets Q and Σ are finite.

Definition 2.3.2. Given a semiautomaton $\mathscr{S} = (Q, \Sigma, \delta, q_0)$, the *extended transition function* $\hat{\delta} : Q \times \Sigma^* \to Q$ is defined recursively:

- $\hat{\delta}(q, \epsilon) = q$ for all $q \in Q$

- $\hat{\delta}(q, ua) = \delta(\hat{\delta}(q, u), a)$ for all $q \in Q, u \in \Sigma^*, a \in \Sigma$

The distinction between an automaton's transition function and its extended transition function can be safely ignored, as every transition function uniquely defines an extended transition function, and vice versa. We shall henceforth use the notation δ for both functions.

Definition 2.3.3. A semiautomaton is *connected* if for every state $q \in Q$ there is a word $w \in \Sigma^*$ such that $\delta(q_0, w) = q$.

We will usually not be interested in disconnected semiautomata, where some states are not reachable from the initial state. Therefore we shall always assume that semiautomata are connected.

Definition 2.3.4. Given the semiautomaton $\mathscr{S}$, define a relation $\equiv_\delta \subseteq \Sigma^* \times \Sigma^*$ such that for $u, v \in \Sigma^*$, $u \equiv_\delta v$ if and only if $\delta(q_0, u) = \delta(q_0, v)$.

Proposition 2.3.1. *The relation $\equiv_\delta$ is a right congruence.*

Proof. Assume that $u \equiv_\delta v$ for some $u, v \in \Sigma^*$. Then, by definition, $\delta(q_0, u) = \delta(q_0, v)$. Let $x \in \Sigma^*$ be an arbitrary word. Then:

$$\begin{aligned} \delta(q_0, ux) &= \delta(\delta(q_0, u), x) && \text{(def. of } \delta) \\ &= \delta(\delta(q_0, v), x) && \text{(assumption)} \\ &= \delta(q_0, vx) && \text{(def. of } \delta) \end{aligned}$$

The equality $\delta(q_0, ux) = \delta(q_0, vx)$ implies $ux \equiv_\delta vx$. ☐

Right congruence relations over the free monoid Σ^* are strongly connected to semiautomata. In fact, every such relation defines a semiautomaton: If $\sim$ is a right congruence on Σ^*, construct a semiautomaton $\mathscr{S}_\sim = (\Sigma^*/\!\sim, \Sigma, nat_\sim, [\epsilon])$, where $nat_\sim : \Sigma^*/\!\sim \times \Sigma \to \Sigma^*/\!\sim$ is defined as $nat_\sim([u], \sigma) = [u\sigma]$ for all $u \in \Sigma^*, \sigma \in \Sigma$.

Specifically, when the right congruence is $\equiv_\delta$, the resulting semiautomaton is isomorphic to the original semiautomaton (which defines the transition function δ):

Theorem 2.3.1. *Let $\mathscr{S} = (Q, \Sigma, \delta, q_0)$ be a semiautomaton. Then $\mathscr{S}_{\equiv_\delta}$ is isomorphic to $\mathscr{S}$.*

Proof. Define a function $f : \Sigma^*/{\equiv_\delta} \to Q$ as $f([w]) = \delta(q_0, w)$ for all $w \in \Sigma^*$. Note that this function is well-defined: the state $\delta(q_0, w)$ is the same regardless of the representative chosen for the equivalence class $[w]$ (by definition of $\equiv_\delta$). We shall now prove that f is an isomorphism.

The function f is certainly surjective: every state $q \in Q$ has a word $w \in \Sigma^*$ such that $\delta(q_0, w) = q$ (remember that we assume that all semiautomata are connected). The word w belongs to some equivalence class in $\Sigma^*/{\equiv_\delta}$ (namely $[w]$), and so every image q has a source $[w]$. This function can also be proved to be injective. Let q_1, q_2 be states in $\mathscr{S}$, and let w_1, w_2 be words leading to these states. Assume that $q_1 = q_2$. Then $\delta(q_0, w_1) = \delta(q_0, w_2)$, which, by the definition of $\equiv_\delta$, means that $w_1 \equiv_\delta w_2$ and hence $[w_1] = [w_2]$, as required.

Finally, we need to show that f preserves the structure of the semiautomaton. This is trivial for the initial state, as $f([\epsilon]) = \delta(q_0, \epsilon) = q_0$. To prove the required result on the transition functions, let $q \in Q$ be a state, let $w \in \Sigma^*$ be a word such that $\delta(q_0, w) = q$, and let $\sigma \in \Sigma$. Then:

$$\begin{aligned} f(nat_\sim([w], \sigma)) &= f([w\sigma]) && \text{(def. of } nat_\sim\text{)} \\ &= \delta(q_0, w\sigma) && \text{(def. of } f\text{)} \\ &= \delta(\delta(q_0, w), \sigma) && \text{(def. of } \delta\text{)} \\ &= \delta(q, \sigma) \end{aligned}$$

□

For the sake of brevity, we use the notation $\mathscr{S}_\delta$ to refer to the semiautomaton generated from $\mathscr{S}$ by the right congruence $\equiv_\delta$ (where δ is the transition function of $\mathscr{S}$).

In the relation $\equiv_\delta$ two words are considered equivalent if they take the semiautomaton from the initial state to the same state. Similarly, we can define a relation on words in Σ^* where two words are equivalent if, starting from *any* state, they take the semiautomaton to the same state. The importance of this relation is that it defines a full congruence on Σ^* (as opposed to the right congruence defined by $\equiv_\delta$).

Definition 2.3.5. Let $\mathscr{S} = (Q, \Sigma, \delta, q_0)$ be a semiautomaton. Define a relation $\equiv_T \subseteq \Sigma^* \times \Sigma^*$ such that for $u, v \in \Sigma^*$, $u \equiv_T v$ if and only if $\delta(q, u) = \delta(q, v)$ for all $q \in Q$.

Proposition 2.3.2. *The relation $\equiv_T$ is a congruence on Σ^*.*

Proof. This relation is reflexive, symmetric and transitive since δ is a function and since $=$ is an equivalence relation on Q.

Assume $u \equiv_T v$ for some $u, v \in \Sigma^*$. Then for all $q \in Q$:

$$\begin{aligned} u \equiv_T v &\Rightarrow \delta(q,u) = \delta(q,v) && \text{(def. of } \sim) \\ &\Rightarrow \delta(\delta(q,u),w) = \delta(\delta(q,v),w) && \text{(since } \delta \text{ is a function)} \\ &\Rightarrow \delta(q,uw) = \delta(q,vw) && \text{(def. of the ext. transition function)} \\ &\Rightarrow uw \equiv_T vw && \text{(def. of } \sim) \end{aligned}$$

which proves the right congruence property. To prove left congruence, let $q \in Q$ be some state and let $w \in \Sigma^*$ be an element of the free monoid. There exists a state $q' \in Q$ such that $\delta(q,w) = q'$. By the assumption, $\delta(q',u) = \delta(q',v)$, and hence, following the definition of δ, we have that $\delta(q,wu) = \delta(q,wv)$. Since this equality holds for an arbitrary choice of q, we conclude that $wu \equiv_T wv$, as required. □

The congruence $\equiv_T$ gives rise to a special structure, called the transformation monoid of the semiautomaton. This structure will prove to be very useful in subsequent chapters.

Proposition 2.3.3. *Let $\mathscr{S}$ be a semiautomaton, let $\equiv_T$ be the relation defined above and let $nat_T : \Sigma^*/{\equiv_T} \times \Sigma^*/{\equiv_T} \longrightarrow \Sigma^*/{\equiv_T}$ be a function defined as $nat_T([u],[v]) = [uv]$ for all $u, v \in \Sigma^*$. Then the triplet $(\Sigma^*/{\equiv_T}, nat_T, \epsilon)$ is a monoid.*

Observe that this result does not apply to an arbitrary relation over Σ^*. For example, the relation $\equiv_\delta$ does not, in the general case, generate a monoid. The reason is that for the monoid's operator to be defined properly, the relation has to be a full congruence. If $\equiv_\delta$ is not a full congruence (and it is usually not), then there exist words $u, v, w \in \Sigma^*$ such that $u \equiv_\delta v$ but $wu \not\equiv_\delta wv$. Then the mapping nat_δ, defined similarly to nat_T, is not a function since $nat_\delta([w],[u]) \neq nat_\delta([w],[v])$ even though $[u] = [v]$.

Definition 2.3.6. Let $\mathscr{S}$ be a semiautomaton. The *transformation monoid of $\mathscr{S}$* is the monoid $T(\mathscr{S}) = (\Sigma^*/{\equiv_T}, nat_T, \epsilon)$.

Brzozowski and Jürgensen have recently developed a theory of semiautomata where states are associated with special words, known as canonical words [5]. For each state of a semiautomaton, a canonical word is a representative chosen from the equivalence class corresponding to that state under the relation $\equiv_\delta$.

Definition 2.3.7. Let $\mathscr{S}$ be a semiautomaton with alphabet Σ. A set $C \subseteq \Sigma^*$ is *canonical* if

1. $\forall u, v \in C : u \not\equiv_\delta v$
2. $\forall v \in \Sigma^* \; \exists u \in C : u \equiv_\delta v$

The above definition implies that, for every word in $u \in \Sigma^*$, there is exactly one word $v \in C$ such that $u \equiv_\delta v$. The word v will be referred to as the *canonical representative* of the word u. We can therefore define a function mapping each word to its canonical representative:

Definition 2.3.8. Let $C \subseteq \Sigma^*$ be a canonical set. Define a function $\xi : \Sigma^* \to C$ such that for all $u \in \Sigma^*, v \in C$, $\xi(u) = v$ if and only if $u \equiv_\delta v$.

Brzozowski and Jürgensen use a slightly different view when talking about canonical words. In their work canonical words are assigned to states rather than to words:

Definition 2.3.9. Let $\mathscr{S} = (Q, \Sigma, \delta, q_0)$ be a semiautomaton, and let $C \subseteq \Sigma^*$ be a canonical set. Define a function $\chi : Q \to C$ which satisfies $\delta(q_0, \chi(q)) = q$ for all $q \in Q$.

Note that the two views are equivalent. Brzozowski and Jürgensen assign canonical words to states using the function χ, which is a bijection. In this work we first identify a state with the subset of Σ^* leading to this state, and then assign the same canonical word to all words leading to the same state. The relations between these two views are summarised in the following proposition:

Proposition 2.3.4. *For all $u, v \in \Sigma^*$:*

1. $\xi(u) = \chi(\delta(q_0, u))$
2. $\xi(u) = \xi(v)$ *if and only if* $u \equiv_\delta v$

Definition 2.3.10. Let $\mathscr{S} = (Q, \Sigma, \delta, q_0)$ be a semiautomaton, and let $C \subseteq \Sigma^*$ be a canonical set. The set $G(C)$ is the set of all pairs $(w\sigma, \xi(w\sigma))$, where $w \in C, \sigma \in \Sigma$ and $w\sigma \notin C$. The set $\hat{G}(C)$ is defined as $\hat{G}(C) = G(C)$, if $\epsilon \in C$, and $\hat{G}(C) = G(C) \cup (\epsilon, \xi(\epsilon))$ otherwise.

Definition 2.3.11. The relation $\equiv$ is the smallest right congruence containing $\hat{G}(C)$.

Theorem 2.3.2 ([5])**.** *For every canonical set C, $\equiv = \equiv_\delta$.*

2.4 Mealy Machines

As mentioned in the previous section, semiautomata are pure state machines, whose only reaction to input is the change of an internal value. As such, they provide a simple theoretical model, but the lack of any externally visible behaviour makes them of little use for the purpose of software specification.

Software modules are often described using a black-box approach, which means that the internal behaviour of the module is of little importance to the user, as long as conditions on the externally visible behaviour are met. These conditions are usually specified by a relation between sequences of input and output values. To specify a software module using such a relation, we extend our state machine model to include output values.

The model studied here is called a *Mealy machine* [28]. Mealy machines are state machines, in which transitions may also result in output values. A Mealy machine is an obvious selection for software specification, as it provides a natural way for describing module functions via input and output alphabets. Other model options are Moore machines and full automata (recognisers). Note, however, that Moore machines are equivalent to Mealy machines [25], and that a recogniser can be easily modelled using a Moore machine by assigning output values that distinguish between accepting and non-accepting states.

Definition 2.4.1. A *Mealy machine* is a six-tuple $\mathscr{M} = (Q, \Sigma, \delta, q_0, \Theta, \theta)$, where (Q, Σ, δ, q_0) is a semiautomaton, Θ is a set, called the *output alphabet*, and $\theta : Q \times \Sigma \to \Theta$ is a partial function, called the output function.

Given an input string, a Mealy machine switches between states, producing an output symbol for each transition on which the output function is defined. If we concatenate all those output symbols, we obtain a string in Θ^*. A Mealy machine can therefore be viewed as a mapping from Σ^* to Θ^*. Formally, we can construct an extended output function which defines this mapping:

Definition 2.4.2. Let $\mathscr{M} = (Q, \Sigma, \delta, q_0, \Theta, \theta)$ be a Mealy machine. The *extended output function* $\hat{\theta} : Q \times \Sigma^* \to \Theta^*$ is defined recursively:

- $\hat{\theta}(q, \epsilon) = \epsilon$
- $\hat{\theta}(q, \sigma w) = \theta(q, \sigma) \circ \hat{\theta}(\delta(q, \sigma), w)$

where $\circ$ is a concatenation operator defined as

$$\theta(q,\sigma) \circ w = \begin{cases} \theta(q,\sigma)w & \text{if } \theta(q,\sigma) \text{ is defined} \\ w & \text{otherwise} \end{cases}$$

for all $w \in \Theta^*, q \in Q, \sigma \in \Sigma$.

To determine the output of a Mealy machine given an input $w \in \Sigma^*$, we compute $\hat{\theta}(q_0, w)$.

As stated earlier, output values provide the only kind of a machine behaviour that is visible to an external viewer. Therefore, from the viewer's perspective, two states are indistinguishable if they generate the same sequences of output values from the same input strings. This observation results in the following equivalence relation on the states of a Mealy machine:

Definition 2.4.3. Let $\mathscr{M} = (Q, \Sigma, \delta, q_0, \Theta, \theta)$ be a Mealy machine. The relation $\equiv_\theta \subseteq Q \times Q$ is defined as $q_1 \equiv_\theta q_2$ if and only if $\hat{\theta}(q_1, w) = \hat{\theta}(q_2, w)$ for all words $w \in \Sigma^*$.

Note that this equivalence relation can be easily converted into a relation on words in Σ^*, by treating the words u and v as equivalent if and only if

$$\hat{\theta}(\delta(q_0, u), w) = \hat{\theta}(\delta(q_0, v), w)$$

for all $w \in \Sigma^*$. We shall subsequently use the notation $\equiv_\theta$ for both the relation on the state set Q and the relation on the free monoid Σ^* described above. The relation referred to can be inferred by the type of objects on both sides of the relation symbol.

Proposition 2.4.1. *For every Mealy machine* $\mathscr{M} = (Q, \Sigma, \delta, q_0, \Theta, \theta)$, $\equiv_\delta \subseteq \equiv_\theta$.

Proof. Let $u, v \in \Sigma^*$ such that $u \equiv_\delta v$. By definition, $\delta(q_0, u) = \delta(q_0, v)$, and therefore $\hat{\theta}(\delta(q_0, u), w) = \hat{\theta}(\delta(q_0, v), w)$ for all $w \in \Sigma^*$. This implies that $u \equiv_\theta v$. □

Definition 2.4.4. A Mealy machine $\mathscr{M} = (Q, \Sigma, \delta, q_0, \Theta, \theta)$ is called *reduced* if $\equiv_\delta = \equiv_\theta$.

The equivalence relation $\equiv_\theta$ is also known as the *generalised Nerode equivalence* or *observational equivalence* [6].

Chapter 3

Algebraic Specifications

3.1 Introduction

Abstract data types have played a major role in computer science since the early 1970s. It was realised during these years that some commonly used data structures can be defined solely by their behaviour, and independently of any implementation issues. This observation resulted in a distinction between *data types*, which specify the type of the stored data as well as the operations that are used to manipulate it, and *abstract data types* (ADTs), for which the actual data being stored is not important. Thus, an abstract data type defines a class of objects, which is completely characterised by a set of operations shared by these objects [27].

Throughout the years there have been several suggestions for ways in which abstract data types can be specified formally. The emphasis has always been that such a specification should describe the operations of the ADT without suggesting any details related to its implementation. A specification method having this trait is often referred to as "abstract", a somewhat elusive concept, which is yet to be properly defined.

Algebraic specifications, as a formal method for describing abstract data types, were first suggested by Guttag in his Ph.D thesis [18]. Guttag's specification consisted of two parts, syntactic and semantic. The first, known as a signature, specifies the operations that define the ADT. An operation consists of a name, the types of arguments it can accept, and the type of its result. The second part is a set of axioms, which defines the behaviour of the operations. Guttag further developed his method in subsequent years, along with Horowitz and Musser [19, 21, 22].

The main problem with this work, however, was that it remained somewhat informal. First of all, it was unclear what can be used in the axioms. For example, Guttag used if-then-else constructs in his axioms, but did not give sufficient reason as to why such a construct was allowed while others were not. Secondly, the correctness of the method for specifying abstract data types was mainly intuitive: for simple ADTs, one could be rather easily convinced that a given specification is correct. But the method was still lacking more formal justification.

Such a justification, through an extensive mathematical foundation, was given by Goguen, Thatcher, Wagner and Wright, who collaborated during the 1970s in what was known as the "ADJ group". The theory proposed by this group was an extension to the work by Birkhoff [2] on universal algebra. Goguen *et al.* suggested that algebraic specifications should be treated within the same framework as universal algebra, only extended to many-sorted algebras. This resulted in the theory of initial algebra semantics [15], which considers the class of initial algebras in a category admitted by the specification as the semantics of the abstract data type. It is this work which is presented in this chapter, and on which the next chapters are based.

It should be mentioned that the initial algebra approach is not the only way to attach formal semantics to algebraic specifications. Several authors have criticised this theory, and suggested other approaches, such as final algebra semantics [35]. Some types of specifications, such as parameterised specifications [34] cannot be handled by the initial algebra approach. Instead, they require *loose semantics* that are neither initial nor final. We shall not consider these approaches in this thesis, nor shall we handle any extensions to the initial algebra approach (e.g., hidden functions), meant to solve some problems that were found in this method.

On a final note, we would like to mention a few endeavours to turn algebraic specifications into a viable method for practical use in day-to-day programming. A first attempt by Burstall and Goguen resulted in the specification language CLEAR [7]. Goguen continued to work in this area, and later created the language OBJ, which, in its latest incarnations (OBJ3, CafeOBJ) is in active use today. As a more recent example of work in this area, see Henkel and Diwan's work on systematically generating algebraic specifications from Java code [23].

The material in this chapter is derived mostly from the book by Ehrig and Mahr [9] and from an article by Wirsing [37]. For the most part, we adopt the notation used in [37].

3.2 Sorts

A key notion in the theory of algebraic specifications is that of *sorts*. A sort can be thought of as a tag that is attached to an element in a set, and is used to specify its type. From a programming language point of view, sorts are analogous to data types, such as `int` and `char` in C/C++.

Definition 3.2.1. Let S be a set of sorts. An *S-sorted set* is a set X, such that each element $x \in X$ is of some sort $s \in S$. The subset of all elements in X of sort $s \in S$ is denoted by X_s.

It is important to note that the sets $\{X_s\}_{s\in S}$ are subsets of X, and not elements of X. Thus an S-sorted set is composed of elements, each of which may be of a different sort. Note also that X_s may be empty for some $s \in S$.

Example 3.2.1. Let $S = \{nat, alpha\}$ be a set of sorts. Let 1, 2, and 3 be elements of sort *nat* and let a and b be elements of sort *alpha*. Then $X = \{1, 2, a, 3, b\}$ is an S-sorted set, with $X_{nat} = \{1, 2, 3\}$ and $X_{alpha} = \{a, b\}$. ■

Sorts can also be used to define the syntax of functions. Given a set S of sorts, the notation $f : s_1, \ldots, s_n \to s$ for some $s_1, \ldots, s_n, s \in S$, declares a function whose ith parameter is of sort s_i, and whose result is of sort s.

Example 3.2.2. Let S be the set of sorts in Example 3.2.1. A function name $f : nat, alpha \to nat$ takes as an argument an ordered pair whose first element is of sort *nat* and the second of sort *alpha*, and returns a value of sort *nat*. ■

3.3 Signatures

Signatures are used to provide the syntactic definition of abstract data types. This syntax is composed of a set of sorts and a set of function prototypes. The prototype of a function includes its name and its type, the latter being a tuple of sorts.

We denote by S^n the set of all n-tuples over some set S, i.e., the Cartesian product $S \times \ldots \times S$ of size n, and by S^+ the union $\bigcup_{n=1}^{\infty} S^n$.

Definition 3.3.1. A *signature* is a pair $\Pi = (S, F)$, where S is a set of *sorts*, and F is an S^+-sorted set.

We regard F as a set of function symbols. For each $f \in F$, the *type* of f is a tuple of size n in S^+. The first $n-1$ elements in this tuple specify the sort of the function's domain, and the last element specifies the sort of its range.

We denote by $f : s_1, \ldots, s_n \to s$ a function whose domain is of sort $(s_1, \ldots, s_n)$ and whose range is of the sort s. Note, however, that this notation does not specify the domain and range sets themselves.

Definition 3.3.2. A *constant of sort $s \in S$* in a signature $\Pi = (S, F)$ is a function whose type is a singleton, i.e., a function $f :\to s \in F$.

Example 3.3.1. Consider the following sets:

$$\begin{aligned}
S &= \{\mathit{bool}, \mathit{nat}\} \\
F &= \{\mathit{true} :\to \mathit{bool}, \\
&\qquad \mathit{false} :\to \mathit{bool}, \\
&\qquad \mathit{zero} :\to \mathit{nat}, \\
&\qquad \mathit{succ} : \mathit{nat} \to \mathit{nat}, \\
&\qquad \mathit{not} : \mathit{bool} \to \mathit{bool}, \\
&\qquad \mathit{equal} : \mathit{nat}, \mathit{nat} \to \mathit{bool}\}
\end{aligned}$$

The pair $\Pi = (S, F)$ is a signature. The type of *true* is (bool), and the type of *equal* is $(\mathit{nat}, \mathit{nat}, \mathit{bool})$. The functions *true*, *false* and *zero* denote constants in Π. ■

The sorts and functions in Example 3.3.1 may suggest that these names should be interpreted according to their meaning in English. This is a pitfall that should be avoided. It is important to keep in mind that *a signature, by itself, provides no semantic information.* Thus, the use of *nat* and *bool* for sorts does not imply in any way that this signature will be used in the specification of an abstract data type involving natural numbers and Boolean values. Likewise, *true*, *false*, *zero*, *succ* and *equal* do not suggest any semantics for the functions in the data type. In fact, as we shall see in Section 3.4, these functions can be assigned meanings that are completely different from those suggested by the selected names.

To emphasise this point, consider the following signature:

Example 3.3.2. $\Pi' = (S', F')$ with

$$\begin{aligned}
S' &= \{\alpha, \beta\} \\
F' &= \{a :\to \beta, \\
&\qquad b :\to \beta, \\
&\qquad c :\to \alpha, \\
&\qquad f : \alpha \to \alpha, \\
&\qquad g : \beta \to \beta, \\
&\qquad h : \alpha, \alpha \to \beta\}
\end{aligned}$$

The signatures defined by Π and Π' are identical, up to sort and function renaming. In other words, Π and Π' are isomorphic. ■

Given a signature, we can use its functions, along with variables of the appropriate sorts, to construct *terms.* Terms represent syntactically correct expressions.

Definition 3.3.3. Let $\Pi = (S, F)$ be a signature, and let X be an S-sorted set. For every sort $s \in S$, the set $T(\Pi, X)_s$ of *terms of sort s* is defined recursively:

- If $x \in X_s$, then $x \in T(\Pi, X)_s$;
- If $f :\to s \in F$, then $f \in T(\Pi, X)_s$;
- For all non-constant function names $f : s_1, \ldots, s_n \to s \in F$ and for all terms $t_1 \in T(\Pi, X)_{s_1}, \ldots, t_n \in T(\Pi, X)_{s_n}$, $f(t_1, \ldots, t_n) \in T(\Pi, X)_s$.

X is referred to as the set of *free variables.*

Definition 3.3.4. The set $T(\Pi, X)$ is the union of the sets $T(\Pi, X)_s$ over all sorts $s \in S$.

Definition 3.3.5. The set of *ground terms* $T(\Pi, \emptyset)$, also denoted $T(\Pi)$, is the set of all terms that do not include free variables.

Example 3.3.3. Let Π be the signature in Example 3.3.1, and let $X = \{x, y, b\}$ be an S-sorted set, with $X_{nat} = \{x, y\}, X_{bool} = \{b\}$. Then the following are examples of terms in $T(\Pi, X)_{bool}$:

1. b
2. *true*

3. $not(equal(zero, zero))$

4. $equal(succ(succ(y)), succ(zero))$

Of these, (2) and (3) are ground terms. ■

Structural induction is an important tool for proving assertions on data types defined using algebraic specifications. This induction mechanism depends on the following theorem:

Theorem 3.3.1. *Let $\Pi = (S, F)$ be a signature, X an S-sorted set and $p : T(\Pi, X) \to \{\mathbf{true}, \mathbf{false}\}$ a predicate. Then $p(t) = \mathbf{true}$ for all terms $t \in T(\Pi, X)$ if the following conditions are satisfied:*

- *$p(t) = \mathbf{true}$ for all constants $t :\to s \in F$ and for all variables $t \in X_s$;*
- *For each term $f(t_1, \ldots, t_n) \in T(\Pi, X)$, $p(t_i) = \mathbf{true}$ for all $i = 1, \ldots, n$, implies $p(f(t_1, \ldots, t_n)) = \mathbf{true}$.*

Proof. see [9], §1.8. □

3.4 Algebras

Algebras are mathematical structures that are used to assign semantics to signatures. An algebra is a set, called the *carrier* of the algebra, and a family of functions on that set. This definition can be easily extended to S-sorted algebras for some set S of sorts, by using an S-sorted carrier set and an S^+-sorted set of functions.

Definition 3.4.1. Let $\Pi = (S, F)$ be a signature. A Π-*algebra* is given by two sets A and F^A, where:

1. A is an S-sorted carrier set;
2. For each function symbol $f : s_1, \ldots, s_n \to s \in F$, F^A contains a total function $f^A : A_{s_1} \times \ldots \times A_{s_n} \to A_s$.
3. If $f :\to s$ is a constant, then f^A is an element in A_s.

We will identify the set A with the algebra $A = (A, F^A)$.

Though this is not stated in the definition, we assume that a Π-algebra's carrier set contains at least one element of every sort. The reason for this is that many-sorted algebras with empty carrier subsets require a different derivation system for equational logic than the one used for one-sorted algebras [13]. By requiring all Π-algebras to have non-empty carrier subsets (for every sort), we can use a generalised version of one-sorted derivation for proving claims on many-sorted systems.

Example 3.4.1. Consider the signature in Example 3.3.1. Let A be an algebra with the carrier sets:[1]

$$\begin{aligned} A_{nat} &= \mathbb{N} \\ A_{bool} &= \{T, F\} \end{aligned}$$

and the following functions:

$$\begin{array}{lll} \mathit{true}^A \overset{\text{def}}{=} T & \mathit{false}^A \overset{\text{def}}{=} F & \\ \mathit{zero}^A \overset{\text{def}}{=} 0 & \mathit{succ}^A(n) \overset{\text{def}}{=} n+1 & \forall n \in A_{nat} \\ \mathit{not}^A(b) \overset{\text{def}}{=} \begin{cases} F & b = T \\ T & b = F \end{cases} & \mathit{equal}^A(n,m) \overset{\text{def}}{=} \begin{cases} T & \text{if } n = m \\ F & \text{otherwise} \end{cases} & \forall n, m \in A_{nat} \end{array}$$

Then A is a Π-algebra.

The Π-algebra A in Example 3.4.1 assigns semantics to the sorts and function symbols of the signature Π. In this case, the semantics is closely related to the usual mathematical interpretation of the words *nat*, *true*, *zero*, etc. This, however, need not be the case for a general Π-algebra. Recall from Section 3.3 that a signature only carries syntactic information, which can be interpreted in many ways. Consider, for instance, the following example:

Example 3.4.2. Let B be an algebra with the carrier sets:

$$\begin{aligned} B_{nat} &= \{\text{red,orange,yellow,green,blue,indigo,violet}\} \\ B_{bool} &= \{\text{gorgeous,awful}\} \end{aligned}$$

[1]The set $\mathbb{N}$ of natural numbers includes 0.

and the following functions:

$$\begin{aligned} \mathit{true}^B &\stackrel{\text{def}}{=} \text{gorgeous} & \mathit{false}^B &\stackrel{\text{def}}{=} \text{awful} \\ \mathit{zero}^B &\stackrel{\text{def}}{=} \text{red} & \mathit{succ}^B(n) &\stackrel{\text{def}}{=} \text{violet} \quad \text{for all } n \in B_{nat} \\ \mathit{not}^B(\text{gorgeous}) &\stackrel{\text{def}}{=} \text{awful} & \mathit{not}^B(\text{awful}) &\stackrel{\text{def}}{=} \text{gorgeous} \end{aligned}$$

$$\mathit{equal}^B(n,m) \stackrel{\text{def}}{=} \begin{array}{l}\text{The expected response of a fashion designer to a} \\ \text{combination of these colours}\end{array}$$

Even though the meanings assigned to the sorts and functions of Π by the algebra B are counter-intuitive, B is still a valid Π-algebra. ■

In order to relate different algebras of the same signatures, we need the concept of a Π-homomorphism. Recall that a homomorphism is a mapping between two mathematical structures that preserves the operations defined on these structures. A Π-homomorphism is therefore a mapping between two Π-algebras that preserves the operations defined by their common signature.

Definition 3.4.2. A Π-*homomorphism* $h : A \to B$, where A and B are Π-algebras, is a set of functions $\{h_s : A_s \to B_s\}_{s \in S}$ such that, for each $f : s_1, \ldots, s_n \to s \in F$ and each $a_1 \in A_{s_1}, \ldots, a_n \in A_{s_n}$, $h_s(f^A(a_1, \ldots, a_n)) = f^B(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$.

We usually omit the sort index from the name of a homomorphism, and treat it as a single mapping rather than a collection. Thus, for a term t of sort s the notation $h(t)$ is equivalent to $h_s(t)$.

Example 3.4.3. Let $\Pi = (S, F)$ be a signature with:

$$\begin{aligned} S =& \{\mathit{bool}\} \\ F =& \{\mathit{true} :\to \mathit{bool}, \\ & \mathit{false} :\to \mathit{bool}, \\ & \mathit{not} : \mathit{bool} \to \mathit{bool}\} \end{aligned}$$

Let A, B be Π-algebras, defined as follows:

$$\begin{aligned} A_{bool} &= \{1,0\} & B_{bool} &= \{U\} \\ true^A &= 1 & true^B &= U \\ false^A &= 0 & false^B &= U \\ not^A(b) &= 1-b \quad \forall b \in A_{bool} & not^B(b) &= U \quad \forall b \in B_{bool} \end{aligned}$$

Define a mapping $h : A \to B$ as $h(1) = h(0) = U$. It is easy to see that h preserves the functions, as defined in the algebra A. For example:

$$h(not^A(true^A)) = h(0) = U = not^B(true^B)$$

Thus, h is a Π-homomorphism. On the other hand, it can be easily verified that no Π-homomorphism exists from B to A. Assume, towards contradiction, that $h : B \to A$ is a Π-homomorphism. Without loss of generality, let $h(U) = 1$. Then $h(false^B) = h(U) = 1 \neq false^A$. ■

Every signature, when coupled with a set of free variables, gives rise to a special algebra, known as the *term algebra*. The carrier sets of this algebra are the terms created from the signature's functions and the free variables.

Definition 3.4.3. Let $\Pi = (S, F)$ be a signature, and X an S-sorted set, such that $T(\Pi, X)_s \neq \emptyset$ for all $s \in S$. Then the *term algebra*, denoted $T(\Pi, X)$ is a Π-algebra, defined as follows:

- For each $s \in S$, the carrier set is $T(\Pi, X)_s$;
- For each $f : s_1, \ldots, s_n \to s \in F$ and $t_1 \in T(\Pi, X)_{s_1}, \ldots, t_n \in T(\Pi, X)_{s_n}$, $f^{T(\Pi,X)}(t_1, \ldots, t_n) \overset{\text{def}}{=} f(t_1, \ldots, t_n)$.

When dealing with free variables, we need a mechanism that maps these variables to concrete values in the carrier sets of some algebra. Mappings from free variables to carrier set elements are provided by *valuations*. The extensions of these mappings to terms that include free variables are known as *interpretations*.

Definition 3.4.4. Let A be a Π-algebra, and let X be an S-sorted set. A *valuation* (or an *assignment*) is a function $\nu : X \to A$. A valuation assigns elements from the carrier sets of A to free variables in X.

Definition 3.4.5. Let $\nu : X \to A$ be a valuation. An *interpretation* of a term $t \in T(\Pi, X)$ in A with respect to ν is a map $\nu^* : T(\Pi, X) \to A$, defined recursively:

- For all $x \in X$, $\nu^*(x) \overset{\text{def}}{=} \nu(x)$;
- For all constants $f \in F$, $\nu^*(f) \overset{\text{def}}{=} f^A$;
- For all non-constant function names $f : s_1, \ldots, s_n \to s \in F$ and all terms $t_1 \in T(\Pi, X)_{s_1}, \ldots, t_n \in T(\Pi, X)_{s_n}$,

$$\nu^*(f(t_1, \ldots, t_n)) \overset{\text{def}}{=} f^A(\nu^*(t_1), \ldots, \nu^*(t_n))$$

From the last two points it is evident that the interpretation of a ground term t is independent of ν. This unique interpretation is denoted as t^A.

It is easy to see that an interpretation is a Π-homomorphism from the term algebra $T(\Pi, X)$ to some other Π-algebra A.

Example 3.4.4. Let Π be a signature and X a set of free variables, as in Example 3.3.3. The carrier sets of $T(\Pi, X)$ are:

$$\begin{aligned} T(\Pi, X)_{\mathit{bool}} &= \{\mathit{true}, b, \mathit{not}(b), \mathit{equal}(\mathit{zero}, \mathit{zero}), \mathit{equal}(\mathit{succ}(x), \mathit{zero}), \ldots\} \\ T(\Pi, X)_{\mathit{nat}} &= \{\mathit{zero}, x, y, \mathit{succ}(\mathit{zero}), \mathit{succ}(\mathit{succ}(x)), \ldots\} \end{aligned}$$

The result of applying the total function $\mathit{succ}^{T(\Pi,X)}$ on the term y is simply $\mathit{succ}(y)$, which, by definition, is a term in $T(\Pi, X)_{\mathit{nat}}$.

Now consider the algebra A of Example 3.4.1. A valuation $\nu : X \to A$ can be defined as follows:

$$\nu(x) = 5, \nu(y) = 3, \nu(b) = F$$

Let $t = \mathit{equal}(\mathit{succ}(\mathit{succ}(y)), x)$ be a term in $T(\Pi, X)_{\mathit{bool}}$. Then

$$\begin{aligned} \nu^*(t) &= \mathit{equal}^A(\nu^*(\mathit{succ}(\mathit{succ}(y))), \nu^*(x)) \\ &= \mathit{equal}^A(\mathit{succ}^A(\nu^*(\mathit{succ}(y))), \nu(x)) \\ &= \mathit{equal}^A(\mathit{succ}^A(\mathit{succ}^A(\nu^*(y))), \nu(x)) \\ &= \mathit{equal}^A(\mathit{succ}^A(\mathit{succ}^A(\nu(y))), \nu(x)) \\ &= \mathit{equal}^A(3 + 1 + 1, 5) \\ &= T \end{aligned}$$

■

The ground term algebra is a special case of a term algebra, where the set of free variables is empty.

Definition 3.4.6. Let $\Pi = (S, F)$ be a signature. The *ground term algebra* $T(\Pi)$ is the term algebra $T(\Pi, \emptyset)$.

Proposition 3.4.1. *For every* Π*-algebra* A*, the interpretation* $\nu^* : T(\Pi) \to A$ *is unique.*

Proof. Follows immediately from the definition of the term algebra $T(\Pi, \emptyset)$, since ν^* is uniquely defined for all ground terms. $\square$

Definition 3.4.7. Let $\Pi = (S, F)$ be a signature and let A be a Π-algebra. The unique interpretation function $\nu^* : T(\Pi) \to A$ defines the *semantics of signature* Π *with respect to* A.

One important consequence of Definition 3.4.2 is that a Π-homomorphism from A to B must map the interpretation of a ground term under algebra A to the interpretation of the *same* ground term under B.

Proposition 3.4.2. *Let* A, B *be* Π*-algebras, and let* $t \in T(\Pi)$ *be a ground term. Then for every* Π*-homomorphism* $h : A \to B$*,* $h(t^A) = t^B$*.*

Proof. By structural induction. Let $f :\to s \in F$ be a constant of sort $s \in S$. Then, by Definition 3.4.2, $h(f^A) = f^B$.

Now assume the lemma holds for the terms $t_1 \in T(\Pi)_{s_1}, \ldots, t_n \in T(\Pi)_{s_n}$, and let $f : s_1, \ldots, s_n \to s \in F$ be a function symbol. If $t = f(t_1, \ldots, t_n)$ then

$$\begin{aligned} h(t^A) &= h(f^A(t_1^A, \ldots, t_n^A)) && \text{(def. of } t^A) \\ &= f^B(h(t_1^A), \ldots, h(t_n^A)) && \text{(def. 3.4.2)} \\ &= f^B(t_1^B, \ldots, t_n^B) && \text{(induction hypothesis)} \\ &= t^B \end{aligned}$$

$\square$

Many Π-algebras contain elements that are not interpretations of any ground terms, and are therefore of no importance for the study of algebras as the semantics of signatures. We would like to omit these algebras from any further discussion of algebraic specifications.

Definition 3.4.8. A Π-algebra A is called *reachable* if for every sort $s \in S$ and every element $a \in A_s$ there is a ground term $t \in T(\Pi)_s$ such that $a = t^A$. In other words, a Π-algebra A is reachable if $\nu^* : T(\Pi) \to A$ is surjective.

Proposition 3.4.3. *For every pair A, B of reachable Π-algebras there is at most one Π-homomorphism $h : A \to B$.*

Proof. Let $h_1, h_2 : A \to B$ be two Π-homomorphisms. We need to show that, for every sort $s \in S$ and every element $a \in A_s$, $h_1(a) = h_2(a)$. Let $t \in T(\Pi)_s$ be a term such that $a = t^A$ (such a term exists since A is reachable). According to Proposition 3.4.2, $h(t^A) = t^B$ for any Π-homomorphism from A to B, and hence $h_1(a) = h_1(t^A) = h_2(t^A) = h_2(a)$. □

It turns out that every Π-algebra can replaced by a reachable Π-algebra that provides the same semantics.

Definition 3.4.9. Let $\Pi = (S, F)$ be a signature, and let A be a Π-algebra. A Π-algebra B is called a Π-*subalgebra* of A if

1. For every sort $s \in S$, $B_s \subseteq A_s$;
2. For every function name $f : s_1, \ldots, s_n \to s$ and for all carrier set elements $b_1 \in B_{s_1}, \ldots, b_n \in B_{s_n}$, $f^B(b_1, \ldots, b_n) = f^A(b_1, \ldots, b_n)$.

Proposition 3.4.4. *Every Π-algebra A contains a reachable Π-subalgebra $R(A)$, such that $\nu^* : T(\Pi) \to A$ is equal to $\nu^* : T(\Pi) \to R(A)$.*

Proof. By construction. Let A be a Π-algebra. Define a Π-algebra $R(A)$ as follows:

1. For each sort $s \in S$ and element $a \in A_s$, $a \in R(A)_s$ iff there exists a term $t \in T(\Pi)_s$ such that $a = t^A$;
2. For all function symbols $f : s_1, \ldots, s_n \to s \in F$ and all elements $a_1 \in R(A)_{s_1}, \ldots, a_n \in R(A)_{s_n}$, $f^{R(A)}(a_1, \ldots, a_n) = f^A(a_1, \ldots, a_n)$.

By definition, $R(A)$ is a Π-subalgebra of A. It is reachable, because it contains only elements of A which are interpretations of ground terms in $T(\Pi)$.

Consider the interpretation function $\nu^* : T(\Pi) \to A$, which we know to be a Π-homomorphism. By definition, the image of ν^* is $R(A)$ and therefore ν^* is a Π-homomorphism from $T(\Pi)$ to $R(A)$. Since both $T(\Pi)$ and $R(A)$ are reachable, ν^* is the unique Π-homomorphism, and hence also the interpretation function from $T(\Pi)$ to $R(A)$. □

Recall that a Π-algebra A defines the semantics of a signature Π through the interpretation function $\nu^* : T(\Pi) \to A$. By Proposition 3.4.4, the interpretation functions from the ground term algebra to A and to $R(A)$ are identical, which means that we can substitute $R(A)$ for A, without changing the semantics.

As a consequence of the above analysis, we shall henceforth consider only reachable Π-algebras.

3.5 Algebraic Specifications

An algebraic specification of an abstract data type is composed of a signature, and a set of axioms. Only those algebras of the specification's signature that adhere to all the given axioms can be regarded as "implementing" the data type (this somewhat-vague description will become formal in a short while). The axioms can be given using various logical systems. For instance, in [34] axioms are conditional formulae (Horn clauses). We shall, however, follow [14], and restrict our specifications to equational logic. In this system all axioms are equations of terms.

Definition 3.5.1. Let Π be a signature, and let X be a set of free variables. A Π-*equation* is a pair (t_1, t_2) (usually written as $t_1 = t_2$), where $t_1, t_2 \in T(\Pi, X)_s$. Note that t_1 and t_2 are of the same sort $s \in S$.

We can now finally define algebraic specifications:

Definition 3.5.2. A *specification* $D = (\Pi, X, E)$ is an ordered triplet of a signature Π, an S-sorted set X and a set E of Π-equations.[2]

Example 3.5.1. Let $\Pi = (S, F)$ be the signature defined in Example 3.3.1, and let E be a set of Π-equations over the set of free variables X. Symbols S, F, X and E are defined as follows (the definition of Π is repeated for the sake of completeness):

[2]Usually the set X is omitted in the literature, and is only implied from the set E. I chose to explicitly specify this set.

$$
\begin{aligned}
S &= \{bool, nat\}\\
F &= \{true :\rightarrow bool,\\
&\qquad false :\rightarrow bool,\\
&\qquad zero :\rightarrow nat,\\
&\qquad succc : nat \rightarrow nat,\\
&\qquad not : bool \rightarrow bool,\\
&\qquad equal : nat, nat \rightarrow bool\}\\
X &= \{x : nat, y : nat\}\\
E &= \{not(true) = false,\\
&\qquad not(false) = true,\\
&\qquad equal(zero, zero) = true\\
&\qquad equal(zero, succ(x)) = false\\
&\qquad equal(x, y) = equal(y, x)\\
&\qquad equal(succ(x), succ(y)) = equal(x, y)\}
\end{aligned}
$$

Then $D = (\Pi, X, E)$ is a specification. ■

A signature Π admits a set of algebras, the Π-algebras. Similarly, a specification D admits a set of D-algebras. However, since a signature does not include semantic information, the only restriction imposed on Π-algebras is that they comply with the syntactic structure of the signature Π. On the other hand, the inclusion of axioms in a specification generates some semantic information to which the algebra should adhere. In other words, the equations given in the specification should be "correct" with respect to a D-algebra. This means that, given a signature Π, and a specification D based on that signature, not all Π-algebras are also D-algebras.

In order to formalise the notion of equations which are "correct" with respect to an algebra, we need to define *validity*:

Definition 3.5.3. Let A be a Π-algebra, $\nu : X \rightarrow A$ a valuation and $e = (t_1, t_2)$ a Π-equation. We say that *e is satisfied by A with respect to ν* if $\nu^*(t_1) = \nu^*(t_2)$. If e is satisfied by A for all valuations ν, then e is *valid in A*.

Example 3.5.2. Let Π be the signature defined in Example 3.5.1. Consider the Π-algebra B, defined as follows:

$$\begin{array}{rlrll}
B_{nat} & \overset{\text{def}}{=} \{0,1\} & B_{bool} & \overset{\text{def}}{=} \{T,F\} & \\
\mathit{true}^B & \overset{\text{def}}{=} T & \mathit{false}^B & \overset{\text{def}}{=} F & \\
\mathit{zero}^B & \overset{\text{def}}{=} 0 & \mathit{succ}^B(n) & \overset{\text{def}}{=} \max\{1-n,1\} & \forall n \in B_{nat} \\
\mathit{not}^B(b) & \overset{\text{def}}{=} \begin{cases} F & b = T \\ T & b = F \end{cases} & \mathit{equal}(n,m) & \overset{\text{def}}{=} \begin{cases} T & n = m \\ F & \text{otherwise} \end{cases} & \forall m,n \in B_{nat}
\end{array}$$

Let $\nu : X \to B$ be a valuation, such that $\nu(x) = 0$. Then the equation

$$\mathit{equal}(\mathit{zero}, \mathit{succ}(x)) = \mathit{false}$$

is satisfied by B w.r.t ν, since:

$$\begin{aligned}
\nu^*(\mathit{equal}(\mathit{zero}, \mathit{succ}(x))) &= \mathit{equal}^B(0, \mathit{succ}^B(\nu(x))) \\
&= \mathit{equal}^B(0, \mathit{succ}^B(0)) \\
&= \mathit{equal}^B(0, 1) \\
&= F \\
&= \mathit{false}^B \\
&= \nu^*(\mathit{false})
\end{aligned}$$

However, this equation is not valid in B. Consider a valuation $\nu' : X \to B$, such that $\nu'(x) = 1$. Then:

$$\begin{aligned}
\nu^*(\mathit{equal}(\mathit{zero}, \mathit{succ}(x))) &= \mathit{equal}^B(1, \mathit{succ}^B(\nu'(x))) \\
&= \mathit{equal}^B(1, \mathit{succ}^B(1)) \\
&= \mathit{equal}^B(1, 1) \\
&= T \\
&= \mathit{true}^B \\
&\neq \nu^*(\mathit{false})
\end{aligned}$$

On the other hand, the equation $\mathit{not}(\mathit{false}) = \mathit{true}$ is valid in B, regardless of the selected valuation ν. (In this case, we only need to check that the equation holds in B for one valuation, since the equation is ground, i.e., it equates two ground terms.) ■

Every Π-algebra has a set of equations which are valid in this algebra. This set is called the *theory* of the algebra. Theories have some nice closure properties.

Definition 3.5.4. Let $\Pi = (S, F)$ be a signature and let X be an S-sorted set of free variables. If A is a Π-algebra, then we denote by $\mathit{Th}(A)$ the set of all Π-equations over X which are valid in A.

Definition 3.5.5. Let $\Pi = (S, F)$ be a signature, and let A and B be Π-algebras. The Π-algebra C is a *product* of A and B if the following conditions holds:

1. $C_s = A_s \times B_s$ for all $s \in S$
2. $f^C = (f^A, f^B)$ for all constants $f \in F$
3. For all function names $f : s_1, \ldots, s_n \to s$ and for all carrier set elements $a_1 \in A_{s_1}, b_1 \in B_{s_1}, \ldots, a_n \in A_{s_n}, b_n \in B_{s_n}$

$$f^C((a_1, b_1), \ldots, (a_n, b_n)) = (f^A(a_1, \ldots, a_n), f^B(b_1, \ldots, b_n))$$

Theorem 3.5.1. *Let A and B and C be Π-algebras. Then*

- *If B is the Π-homomorphic image of a subalgebra of A, then $\mathit{Th}(A) \subseteq \mathit{Th}(B)$*
- *If C is the product of A and B, then $\mathit{Th}(C) = \mathit{Th}(A) \cap \mathit{Th}(B)$*

Proof. See [9] §4.12. □

We now connect the theory of an algebra with the set of axioms given by a specification. In the case that all axioms in a specification are valid in an algebra, then we can view this algebra as realising the specification:

Definition 3.5.6. Let $D = (\Pi, X, E)$ be a specification. If A is a Π-algebra such that all equations in E are valid in A, then A is a *D-algebra.*

It should be clear that A is a D-algebra if and only if $E \subseteq \mathit{Th}(A)$.

The algebra B in Example 3.5.2 is not a D-algebra for the specification of Example 3.5.1, as it was shown that there exists a valuation, with respect to which one of the equations is not satisfied by B. On the other hand, it can be proved that the algebra A of Example 3.4.1 is a D-algebra for this specification.

Thus far, we have defined algebraic specifications, and seen how a specification includes both syntactic information, through its signatures, and semantic information, by admitting a set of algebras that correspond to both the signature and the axioms. However, no connection was made between algebraic specifications and abstract data types, though it was claimed that the former are a way for describing the latter. To make this connection, we introduce the *quotient term algebra*, a special kind of algebra that is admitted by every algebraic specification.

Definition 3.5.7. Let $D = (\Pi, X, E)$ be a specification. The relation $\sim^D$ is defined on ground terms $t_1, t_2 \in T(\Pi)$ in the following way:

$$t_1 \sim^D t_2 \Leftrightarrow t_1^A = t_2^A$$

for all D-algebras A.

Theorem 3.5.2. *The relation $\sim^D$ is a congruence, i.e., it is an equivalence relation such that for all $t_1, t_1' \in T(\Pi)_{s_1}, \ldots, t_n, t_n' \in T(\Pi)_{s_n}$ and $f : s_1, \ldots, s_n \to s \in F$:*

$$t_1 \sim^D t_1', \ldots, t_n \sim^D t_n' \Rightarrow f(t_1, \ldots, t_n) \sim^D f(t_1', \ldots, t_n')$$

Proof. See [9] §2.4. □

Definition 3.5.8. Let $D = (\Pi, X, E)$ be a specification. The *quotient term algebra* $T(D)$ is a Π-algebra defined as follows:

- For all $s \in S$, the carrier set $T(D)_s \overset{\text{def}}{=} T(\Pi)_s / \sim^D$;
- If $f :\to s \in F$ is a constant, then $f^{T(D)} \overset{\text{def}}{=} [f]$;
- For all non-constant function names $f : s_1, \ldots, s_n \to s \in F$ and for all terms $[t_1] \in T(D)_{s_1}, \ldots, [t_n] \in T(D)_{s_n}$, $f^{T(D)}([t_1], \ldots, [t_n]) \overset{\text{def}}{=} [f(t_1, \ldots, t_n)]$.

Theorem 3.5.3. *For any specification $D = (\Pi, X, E)$, the quotient term algebra $T(D)$ is a D-algebra.*

Proof. See [9] §2.9. □

In the next sections we shall see how the quotient term algebra is used to generate an abstract data type out of a specification.

3.6 Categories

Given a signature Π, the collection of all Π-algebras and Π-homomorphisms, forms a special mathematical structure, known as a *category*. Category theory will be used to extend the power of equational logic in algebraic specifications, and help define abstract data structures.

Definitions in this chapter are taken from [31].

Definition 3.6.1. A *category* $\mathbf{C}$ comprises:

1. a collection of objects;
2. a collection of morphisms (or arrows);
3. operations assigning to each morphism f an object $dom(f)$, called the *domain* of f, and an object $cod(f)$, called the *co-domain* of f. If $dom(f) = A$ and $cod(f) = B$, we write $f : A \to B$;
4. A composition operator assigning to each pair of morphisms f, g with $cod(f) = dom(g)$ a morphism $g \circ f : dom(f) \to cod(g)$. This composition operator needs to be associative, i.e., if $f : A \to B$, $g : B \to C$ and $h : C \to D$ are morphisms, then
$$h \circ (g \circ f) = (h \circ g) \circ f;$$
5. For each object A, a morphism id_A, such that for any morphism $f : A \to B$
$$id_B \circ f = f \circ id_A = f.$$

Proposition 3.6.1. *Given a signature Π, the set of all Π-algebras, together with Π-homomorphisms, forms a category, denoted $\mathbf{Alg}(\Pi)$.*

Proof. The objects of the category $\mathbf{Alg}(\Pi)$ are the Π-algebras, and the morphisms are the Π-homomorphisms. If A, B, C are Π-algebras, and $h_1 : A \to B$, $h_2 : B \to C$ are Π-homomorphisms, define the composition $h_1 \circ h_2$ recursively:

- For each $s \in S$ and $a \in A_s, b \in B_s, c \in C_s$ such that $h_1(a) = b$ and $h_2(b) = c$, let
$$(h_2 \circ h_1)(a) \overset{\text{def}}{=} c$$

- For each $f : s_1, \ldots, s_n \to s \in F$ and $a_1 \in A_{s_1}, \ldots, a_n \in A_{s_n}$, define

$$(h_2 \circ h_1)(f^A(a_1, \ldots, a_n)) \overset{\text{def}}{=} f^C((h_2 \circ h_1)(a_1), \ldots, (h_2 \circ h_1)(a_n))$$

The composition $h_2 \circ h_1$ is then a Π-homomorphism from A to C.

For every object A we define a mapping id_A, such that for all $a \in A$, $id_A(a) = a$. It is easy to see that id_A is a Π-homomorphism, and is indeed the required identity morphism.

Finally, we need to show that the composition operator is associative. This is, however, an immediate result, if we observe that for any object sort $s \in S$, any element $a \in A_s$, and any Π-homomorphisms $h_1 : A \to B, h_2 : B \to C, h_3 : C \to D$, the composition $(h_3 \circ h_2 \circ h_1)(a)$ is equal, by definition, to the composition of the (set-theoretic) functions $((h_3)_s \circ (h_2)_s \circ (h_1)_s)(a)$. Since function composition is associative, so is the composition of Π-homomorphisms. □

Definition 3.6.2. A category **D** is a *subcategory* of category **C** if:

- Each object in **D** is also an object in **C**;
- Each morphism in **D** (including all identity morphisms) is also a morphism in **C**;
- The composition operator in **D** is a restriction of the operator defined in **C**.

Proposition 3.6.2. *Let $D = (\Pi, X, E)$ be a specification. Then:*

1. *The set of all D-algebras, with the Π-homomorphisms between any two of them, forms a category* $\mathbf{Alg}(D)$*;*
2. $\mathbf{Alg}(D)$ *is a subcategory of* $\mathbf{Alg}(\Pi)$.

Proof.

1. The proof is identical to that of Proposition 3.6.1, with composition and identity defined in the same way as for $\mathbf{Alg}(\Pi)$.
2. Follows immediately from the definition of a subcategory and the construction of $\mathbf{Alg}(D)$.

□

The category $\mathbf{Alg}(D)$ contains one special object, which will prove to be very important in our discussion of algebraic specifications:

Definition 3.6.3. Let $\mathbf{C}$ be a category. An object $I \in \mathbf{C}$ is called *initial*, if for every object $A \in \mathbf{C}$ there exists a unique morphism $f_A : I \to A$.

Recall from Proposition 3.4.3 that between the elements of each pair of Π-algebras there is at most one Π-homomorphism. The above definition can therefore be revised for the category $\mathbf{Alg}(D)$:

Definition 3.6.4. Let D be a specification. An algebra $I \in \mathbf{Alg}(D)$ is called an *initial algebra* if for every D-algebra A there exists a Π-homomorphism $h : I \to A$.

Theorem 3.6.1. *The quotient term algebra $T(D)$ is an initial algebra in the category $\mathbf{Alg}(D)$.*

Proof. See [9] §3.7. □

Proposition 3.6.3. *A D-algebra I is initial in the category $\mathbf{Alg}(D)$ if there exists a Π-homomorphism $h : I \to T(D)$.*

Proof. Let I be a D-algebra, and let $h : I \to T(D)$ be a Π-homomorphism. By Proposition 3.4.3, this homomorphism is unique. Let $A \in \mathbf{Alg}(D)$ be some algebra. Since $T(D)$ is an initial algebra, there exists a unique Π-homomorphism $h' : T(D) \to A$. Now consider the composition $h' \circ h : I \to A$, defined by taking the compositions of the sorted functions $\{h'_s \circ h_s\}_{s \in S}$. This composition is a Π-homomorphism. Moreover, since both h and h' are unique, so is their composition. Thus for every D-algebra A there is a unique Π-homomorphism from I, which, by definition, makes I initial. □

The algebras $T(D)$ and I are, in fact, isomorphic, as will be shown in the next section.

3.7 Abstract Data Types

It is time to answer the fundamental question behind this chapter: "what is an abstract data type?" According to the initial algebra approach to algebraic specifications, an abstract data type is the class of all initial algebras in a category $\mathbf{Alg}(D)$. This definition gains its strength from the fact that each of these algebras is isomorphic to every other.

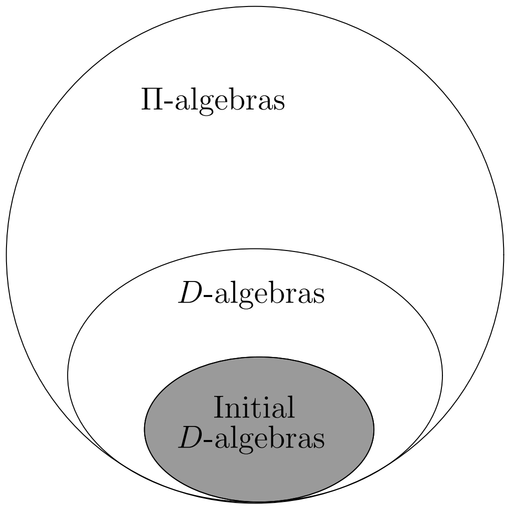


Figure 3.1: The figure depicts the relationship between the categories $\mathbf{Alg}(\Pi)$ and $\mathbf{Alg}(D)$ for some specification D over a signature Π. The shaded area denotes the abstract data type, as an isomorphism class of all initial D-algebras.

Definition 3.7.1. Let $D = (\Pi, X, E)$ be a specification. The *abstract data type* admitted by D is the set of D-algebras that are isomorphic to the quotient term algebra $T(D)$.

Theorem 3.7.1. *If A and A' are two initial algebras in a category $\mathbf{C}$, then A and A' are isomorphic. If A is an initial algebra in $\mathbf{C}$ and $A'' \in \mathbf{C}$ is isomorphic to A, then A'' is an initial algebra in $\mathbf{C}$.*

Proof. See [14]. □

Corollary 3.7.1. *An abstract data type is the set of all algebras that are initial in $\mathbf{Alg}(D)$.*

Figure 3.1 summarises the initial algebra approach to algebraic specifications. We can see how abstraction is achieved by this approach: the ADT is a set of all possible implementations of the same specification. Any initial D-algebra, by itself, is just a member of the isomorphism class, and can therefore be considered an implementation of the ADT.

Chapter 4

From Automata to Algebraic Specifications

4.1 Introduction

In this chapter we investigate one aspect of the relationship between algebraic specifications and simple state machines. Specifically, we determine how a semiautomaton can be used to derive both a specification and an algebra in such a way that the resulting algebra is an initial object in the category created by the specification. Such a construction implies that the abstract data type represented by the semiautomaton is exactly the one specified by the derived algebraic specification.

The connection made between a semiautomaton and an algebraic specification is based upon three seemingly different equivalence relations, which were defined in previous chapters. The first is $\sim^D$, defined by an algebraic specification on ground terms of the specification's signature. The second is $\equiv_\delta$, defined on words from the free monoid Σ^*, such that two words are equivalent if and only if they lead to the same state under the semiautomaton's extended transition function.

The third equivalence relation is likewise defined on words. It is derived from a set of pairs of words, and is defined as the smallest right congruence containing this set (which will be referred to as the *generating set* of the relation). We usually denote this set as G and the resulting relation as $\equiv_G$. The generating set provides us with a set of equations resulting in an algebraic specification D. Our main result shows how to construct such a set so that the following two conditions hold:

1. $\equiv_\delta = \equiv_G$ and

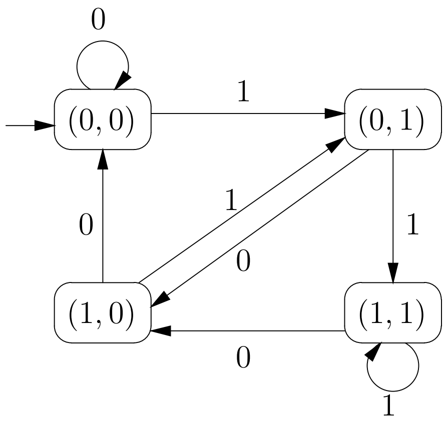


Figure 4.1: The shift register semiautomaton. The state names show the contents of the two-bit register.

2. There is a function $\varphi : \Sigma^* \to T(\Pi)_{state}$ such that for all words $u, v \in \Sigma^*$, $u \equiv_G v$ if and only if $\varphi(u) \sim^D \varphi(v)$.

Some of the results given in the following sections are an extension of similar results obtained by Brzozowski and Jürgensen in [5]. Since their work was motivated by another method for software specifications, namely trace assertions, we obtain an interesting (though, perhaps, not surprising) connection between specifications by automata, trace assertions and algebraic specifications.

To facilitate the reading of this chapter we present a semiautomaton for a two-bit shift register. We will refer to this example in the following sections.

Example 4.1.1. A two-bit shift register is an abstract data type that maintains an array of two place holders, each containing either 0 or 1, and which supports a single operation *shift*. This operation takes a bit value as an argument, copies the bit value in the right place holder to the left one, and stores the argument in the right place holder. A diagram of a semiautomaton implementing such a shift register is given in Figure 4.1. ■

4.2 Algebraic Specification of a Semiautomaton

Semiautomata were defined in Section 2.3 as complete and initialised state machines over some alphabet. We would now like to give a different definition, one that will allow us to treat semiautomata with the tools of the theory of algebraic specifications. The ultimate goal of this section is to provide an axiomatic definition of a specific semiautomaton.

First, we need to establish the syntactic view of a semiautomaton over a given alphabet:

Definition 4.2.1. Let Σ be a set. The signature $\Pi_\Sigma = (S_\Sigma, F_\Sigma)$ gives the syntactic structure of a semiautomaton over the alphabet Σ:

$$\begin{aligned}
S_\Sigma &= \{\mathit{state}, \mathit{alpha}\}\\
F_\Sigma &= \{a_\sigma :\rightarrow \mathit{alpha} \quad \forall \sigma \in \Sigma,\\
&\qquad \mathit{init} :\rightarrow \mathit{state},\\
&\qquad \mathit{trans} : \mathit{state}, \mathit{alpha} \rightarrow \mathit{state}\}
\end{aligned}$$

Any semiautomaton over the alphabet Σ can now be made into a Π_Σ-algebra:

Definition 4.2.2. Let $\mathscr{S} = (Q, \Sigma, \delta, q_0)$ be a semiautomaton. Define the semiautomaton algebra $A(\mathscr{S})$ as follows:

- The carrier set is $Q \cup \Sigma$
- $a_\sigma^{A(\mathscr{S})} = \sigma$ for all $\sigma \in \Sigma$
- $\mathit{init}^{A(\mathscr{S})} = q_0$
- $\mathit{trans}^{A(\mathscr{S})} = \delta$

Proposition 4.2.1. *The algebra $A(\mathscr{S})$ is a Π_Σ-algebra.*

Proof. Follows immediately from the definitions and the construction of Π_Σ and $A(\mathscr{S})$. □

It is important to note that the signature Π_Σ can only be used to give the syntax of semiautomata over the alphabet Σ, or any alphabet Σ' such that $|\Sigma'| \leq |\Sigma|$. A semiautomaton over an alphabet Σ', where $|\Sigma'| > |\Sigma|$ will not yield a reachable Π_Σ-algebra: some letters in Σ' will not be the image of any term of sort *alpha*. Of the remaining semiautomata, those over alphabets which are not isomorphic to Σ (namely, alphabets of size smaller than $|\Sigma|$) are of little importance. The initial algebra approach guarantees that such machines will not be part of the abstract data type defined over the signature Π_Σ (as long as there are no axioms involving terms of sort *alpha*).

One possible alternative to the signature given here is to use a parameterised specification [34], where the alphabet is omitted from the signature and instead given as a parameter. Parameterised algebraic specifications, however, require a different mathematical infrastructure, in which abstract data types are not defined by the initial algebra approach. Instead, the theory of parameterised algebraic specifications uses "loose semantics": an abstract data type is defined as the entire category of D-algebras, rather than an isomorphism class of initial algebras. Parameterised algebraic specifications are beyond the scope of this thesis.

Any initial algebra in the category $\mathbf{Alg}(\Pi_\Sigma)$ is isomorphic to the free semiautomaton over Σ. To obtain a specification for more interesting semiautomata, we need to add a few axioms. Our main method for deriving appropriate axioms will require a transformation of words over Σ^* into terms of sort *state*. Such a transformation is given by the following function:

Definition 4.2.3. The function $\pi : \Sigma^* \to T(\Pi_\Sigma)_{state}$ is defined inductively for all $w \in \Sigma^*, \sigma \in \Sigma$:

- $\pi(\epsilon) = \mathit{init}$
- $\pi(w\sigma) = \mathit{trans}(\pi(w), a_\sigma)$

It is easy to see that this transformation maps each word in Σ^* to a ground term in $T(\Pi)_{state}$. Moreover, given such a term t, the word $w \in \Sigma^*$ for which $\pi(w) = t$ can be uniquely reconstructed. We therefore obtain the following proposition:

Proposition 4.2.2. *The function π is a bijection.*

In this chapter we are only going to construct ground equations as axioms. Recall that for the ground term algebra $T(\Pi_\Sigma)$ and for any Π_Σ-algebra A, the interpretation function $\nu^* : T(\Pi_\Sigma) \to A$ is unique (Definition 3.4.5). The next lemma shows how to compute ν^* from the transformation function π:

Lemma 4.2.1. *Let $\nu^* : T(\Pi_\Sigma) \to A(\mathscr{S})$ be the unique interpretation function for the semiautomaton algebra of $\mathscr{S}$. Then for all terms $t \in T(\Pi_\Sigma)_{state}$, $\nu^*(t) = \delta(q_0, \pi^{-1}(t))$.*

Proof. By structural induction.

Base Case Let $t = \mathit{init}$. Then

$$\begin{aligned}\nu^*(\mathit{init}) &= q_0 \\ &= \delta(q_0, \epsilon) \\ &= \delta(q_0, \pi^{-1}(\mathit{init}))\end{aligned}$$

Step Let $t \in T(\Pi_\Sigma)_{\mathit{state}}, a_\sigma \in T(\Pi_\Sigma)_{\mathit{alpha}}$ be terms. Then

$$\begin{aligned}\nu^*(\mathit{trans}(t, a_\sigma)) &= \delta(\nu^*(t), \nu^*(a_\sigma)) \\ &= \delta(\delta(q_0, \pi^{-1}(t)), \sigma) && \text{(hypothesis)} \\ &= \delta(q_0, \pi^{-1}(t)\sigma) && \text{(def. of } \delta) \\ &= \delta(q_0, \pi^{-1}(\mathit{trans}(\pi(\pi^{-1}(t)), a_\sigma))) && \text{(def. of } \pi) \\ &= \delta(q_0, \pi^{-1}(\mathit{trans}(t, a_\sigma)))\end{aligned}$$

□

We are now ready to show the first method for obtaining correct axioms for a given semiautomaton. Consider two words $u, v \in \Sigma^*$. We show that if $u \equiv_\delta v$, then the transformation of u and v into Π_Σ-terms of sort *state* yields an axiom which is valid in the semiautomaton algebra. Thus, any subset of the relation $\equiv_\delta$ provides a set of axioms which are valid in this algebra.

Definition 4.2.4. Let $\mathscr{S}$ be a semiautomaton, and let $G \subseteq \equiv_\delta$ be a set. The set $E(G)$ of ground equations over Π_Σ is defined as follows:

$$E(G) = \{(\pi(w_1), \pi(w_2)) | w_1, w_2 \in \Sigma^*, (w_1, w_2) \in G\}$$

Theorem 4.2.1. *Let $\mathscr{S} = (Q, \Sigma, \delta, q_0)$ be a semiautomaton, and let $G \subseteq \equiv_\delta$ be a set. Define an algebraic specification $D = (\Pi_\Sigma, \varnothing, E(G))$. Then $A(\mathscr{S})$ is a D-algebra.*

Proof. Let $(t_1, t_2) \in E(G)$ be an equation. By definition, there exist words $w_1, w_2 \in \Sigma^*$ such that $(w_1, w_2) \in G$ and $\pi(w_1) = t_1, \pi(w_2) = t_2$. By Lemma 4.2.1, we know that $\nu^*(t_1) = \delta(q_0, w_1)$ and $\nu^*(t_2) = \delta(q_0, w_2)$. But $(w_1, w_2) \in G$ implies $w_1 \equiv_\delta w_2$, and therefore

$$\nu^*(t_1) = \delta(q_0, w_1) = \delta(q_0, w_2) = \nu^*(t_2)$$

as required. □

Recall that the set $\hat{G}(C)$ is a set of pairs of words over Σ^*, such that for every pair $(u, v) \in \hat{G}(C)$, $u \equiv_\delta v$ (Definition 2.3.10). The following corollary can therefore be obtained from Theorem 4.2.1:

Corollary 4.2.1. *The algebra $A(\mathscr{S})$ is a D-algebra for the specification $D = (\Pi_\Sigma, \varnothing, E(\hat{G}(C)))$.*

Proof. Apply Theorem 4.2.1 on the set $\hat{G}(C) \subseteq {\equiv_\delta}$. □

4.3 Properties of Generating Sets

We have seen that a set $G \subseteq {\equiv_\delta}$ gives rise to two algebraic structures that are of interest to our discussion. The first is the smallest right congruence containing G, which is defined on the free monoid Σ^*. The second structure is the specification $D = (\Pi_\Sigma, \varnothing, E(G))$, which uses a transformation of the set G into a set of equations.

It may not come as a big surprise that these two structures share a great deal in common. In fact, as we shall now prove, the term congruence that arises from the specification D is strongly related to the smallest right congruence containing G.

In the following results we assume the existence of a semiautomaton $\mathscr{S} = (Q, \Sigma, \delta, q_0)$ and of a set $G \subseteq {\equiv_\delta}$. The latter gives rise to the relation $\equiv_G$, defined as the smallest right congruence containing G, and to the specification $D = (\Pi_\Sigma, \varnothing, E(G))$.

Lemma 4.3.1. *For all words $w_1, w_2 \in \Sigma^*$, $w_1 \equiv_G w_2$ implies $\pi(w_1) \sim^D \pi(w_2)$.*

Proof. Define a relation $\equiv_D \subseteq \Sigma^* \times \Sigma^*$ such that for all $u, v \in \Sigma^*$, $u \equiv_D v$ if and only if $\pi(u) \sim^D \pi(v)$. It is easy to see that $\equiv_D$ is an equivalence relation (reflexivity, symmetry and transitivity follow from the same properties on the relation $\sim^D$).

We shall now prove that $\equiv_D$ is a right congruence. Assume that $w_1 \equiv_D w_2$ for some words $w_1, w_2 \in \Sigma^*$. Then $\pi(w_1) \sim^D \pi(w_2)$ by definition. Remember that $\sim^D$ is a Π-congruence, and therefore for all $a_\sigma \in T(\Pi_\Sigma)_{state}$ we get

$$\mathit{trans}(\pi(w_1), a_\sigma) \sim^D \mathit{trans}(\pi(w_2), a_\sigma)$$

By the definition of π this implies that $\pi(w_1\sigma) \sim^D \pi(w_2\sigma)$ for all $\sigma \in \Sigma$. We can now apply the same process on strings of any length to prove right congruence.

To complete the proof we note that $\equiv_D$ includes the pairs in G (since $\sim^D$ includes the equations in $E(G)$). But $\equiv_G$ was defined as the *smallest* right congruence containing G, and so it must follow that $\equiv_G \subseteq \equiv_D$. Thus we get that for all $w_1, w_2 \in \Sigma^*$, $w_1 \equiv_G w_2$ implies $w_1 \equiv_D w_2$ and then by definition $\pi(w_1) \sim^D \pi(w_2)$. □

Proving the converse of the above lemma is a bit harder, and requires the use of a D-algebra generated by the set G:

Definition 4.3.1. The algebra $A(G)$ is a Π_Σ-algebra, defined as follows:

- The carrier set is $\Sigma^*/{\equiv_G} \cup \Sigma$
- $a_\sigma^{A(G)} = \sigma$ for all $\sigma \in \Sigma$
- $\mathit{init}^{A(G)} = [\epsilon]_G$
- $\mathit{trans}^{A(G)} = \mathit{nat}_G$

where $\mathit{nat}_G : \Sigma^*/{\equiv_G} \times \Sigma \to \Sigma^*/{\equiv_G}$ is defined as $\mathit{nat}_G([u]_G, \sigma) = [u\sigma]_G$.

Lemma 4.3.2. *For all $t \in T(\Pi_\Sigma)_{\mathit{state}}$, $t^{A(G)} = [\pi^{-1}(t)]_G$.*

Proof. By structural induction.

Base Case $\mathit{init}^{A(G)} = [\epsilon]_G = [\pi^{-1}(\mathit{init})]_G$

Step Let $t \in T(\Pi_\Sigma)_{\mathit{state}}$ be a term, and let $w \in \Sigma^*$ be a word such that $\pi(w) = t$.

$$\begin{aligned}
(\mathit{trans}(t, a_\sigma))^{A(G)} &= \mathit{trans}^{A(G)}(t^{A(G)}, a_\sigma^{A(G)}) \\
&= \mathit{nat}_G([\pi^{-1}(t)]_G, \sigma) \\
&= [w\sigma]_G \\
&= [\pi^{-1}(\mathit{trans}(\pi(w), a_\sigma))]_G \\
&= [\pi^{-1}(\mathit{trans}(t, a_\sigma))]_G
\end{aligned}$$

□

Lemma 4.3.3. *Let $D = (\Pi_\Sigma, \varnothing, E(G))$ be a specification. Then the algebra $A(G)$ is a D-algebra.*

Proof. Let $(t_1, t_2) \in E(G)$ be an equation in D. By construction of the set $E(G)$, $(\pi^{-1}(t_1), \pi^{-1}(t_2)) \in G$, which means that $[\pi^{-1}(t_1)]_G = [\pi^{-1}(t_2)]_G$. By Lemma 4.3.2 we get $t_1^{A(G)} = t_2^{A(G)}$. □

Theorem 4.3.1. *Let $\mathscr{S} = (Q, \Sigma, \delta, q_0)$ be a semiautomaton, let $G \subseteq \equiv_\delta$ be a set, let $\equiv_G$ be the smallest right congruence containing G and let $D = (\Pi_\Sigma, \varnothing, E(G))$ be a specification. Then, for all words $w_1, w_2 \in \Sigma^*$, $w_1 \equiv_G w_2$ if and only if $\pi(w_1) \sim^D \pi(w_2)$.*

Proof. The direction $w_1 \equiv_G w_2 \Rightarrow \pi(w_1) \sim^D \pi(w_2)$ was already proved in Lemma 4.3.1.

Assume that $\pi(w_1) \sim^D \pi(w_2)$. Then for every D-algebra A, $\pi(w_1)^A = \pi(w_2)^A$. Specifically, this equation holds for the D-algebra $A(G)$. According to Lemma 4.3.2, $\pi(w)^{A(G)} = [w]_G$ for all words $w \in \Sigma^*$, and therefore $[w_1]_G = [w_2]_G$, or, in other words, $w_1 \equiv_G w_2$. □

4.4 Initial Algebras

We now turn our attention to the conditions under which a semiautomaton provides an initial algebra for a specification created from that machine. Such conditions are important as they provide the means through which we can create algebraic specifications that truly reflect the semiautomata from which they were generated.

Theorem 4.4.1. *Let $\mathscr{S} = (Q, \Sigma, \delta, q_0)$ be a semiautomaton, let C be a canonical set and let $D = (\Pi_\Sigma, \varnothing, E(\hat{G}(C)))$ be the resulting algebraic specification. Then for all words $w_1, w_2 \in \Sigma^*$, $w_1 \equiv_\delta w_2 \Leftrightarrow \pi(w_1) \sim^D \pi(w_2)$.*

Proof. Let $t_1 = \pi(w_1), t_2 = \pi(w_2)$, and assume that $t_1 \sim^D t_2$. Then for all D-algebras $\nu^*(t_1) = \nu^*(t_2)$ (by definition). Specifically, this equality holds for the algebra $A(\mathscr{S})$. By Lemma 4.2.1, $\delta(q_0, \pi^{-1}(t_1)) = \delta(q_0, \pi^{-1}(t_2))$, which implies $\pi^{-1}(t_1) \equiv_\delta \pi^{-1}(t_2)$. But $\pi^{-1}(t_1) = w_1$ and $\pi^{-1}(t_2) = w_2$ and thus $w_1 \equiv_\delta w_2$.

Conversely, assume that $w_1 \equiv_\delta w_2$. From Theorem 2.3.2 we know that $w_1 \equiv w_2$. By Theorem 4.3.1 this means that $\pi(w_1) \sim^D \pi(w_2)$. □

Our aim now is to prove that the algebra $A(\mathscr{S})$ is initial. To facilitate the proof, we first construct an algebra isomorphic to $A(\mathscr{S})$, prove that it is initial, and then use Theorem 3.7.1 to conclude the initiality of $A(\mathscr{S})$.

Definition 4.4.1. Let $\mathscr{S} = (Q, \Sigma, \delta, q_0)$ be a semiautomaton, and let $\mathscr{S}_\delta = (\Sigma^*/\equiv_\delta\ , \Sigma, nat_\delta, [\epsilon])$ be its isomorphic image created by the relation $\equiv_\delta$ (see Theorem 2.3.1). Define the algebra $A(\mathscr{S}_\delta)$ as follows:

- The carrier set is $\Sigma^*/\equiv_\delta \cup \Sigma$
- $a_\sigma^{A(\mathscr{S}_\delta)} = \sigma$ for all $\sigma \in \Sigma$
- $init^{A(\mathscr{S}_\delta)} = [\epsilon]_\delta$
- $trans^{A(\mathscr{S}_\delta)} = nat_\delta$

(Note that this definition is similar to Definition 4.3.1 with $\equiv_\delta$ replacing $\equiv_G$.)

Lemma 4.4.1. *The algebras $A(\mathscr{S})$ and $A(\mathscr{S}_\delta)$ are isomorphic.*

Proof. follows immediately from the isomorphism of $\mathscr{S}$ and the semiautomaton $\mathscr{S}_\delta$. □

Lemma 4.4.2. *The algebra $A(\mathscr{S}_\delta)$ is an initial D-algebra.*

Proof. Let $h : T(D) \rightarrow A(\mathscr{S}_\delta)$ be an S-sorted mapping, defined as follows:

- $h([a_\sigma]_D) = \sigma$
- $h([t]_D) = [\pi^{-1}(t)]_\delta, \forall t \in T(\Pi_\Sigma)_{state}$

We shall now prove that h is an isomorphism. The mapping h_{alpha} is clearly an isomorphism, since $[a_\sigma]$ is a singleton for all $\sigma \in \Sigma$.

Consider the mapping h_{state}. This mapping is surjective, since π is invertible: for every word $u \in \Sigma^*$, there is a term $t \in T(\Pi_\Sigma)_{state}$ such that $\pi(u) = t$ and hence $h([t]_D) = [u]_\delta$. The mapping is also injective:

$$\begin{aligned}
h([t_1]_D) = h([t_2]_D) &\Rightarrow [\pi^{-1}(t_1)]_\delta = [\pi^{-1}(t_2)]_\delta \\
&\Rightarrow \pi^{-1}(t_1) \equiv_\delta \pi^{-1}(t_2) \\
&\Rightarrow t_1 \sim^D t_2 && \text{(Theorem 4.4.1)} \\
&\Rightarrow [t_1]_D = [t_2]_D
\end{aligned}$$

□

Theorem 4.4.2. *The algebra $A(\mathscr{S})$ is an initial D-algebra.*

Proof. Combine Lemma 4.4.2 with Lemma 4.4.1. □

Example 4.4.1. Consider the shift register semiautomaton. Assume that the canonical set is selected to be $C = \{0, 1, 10, 11\}$. It can be easily verified that every state is represented in this set, and that no two words take the initial state to the same state. The set $G(C)$ is composed of the following pairs: $(00, 0)$, $(01, 1)$, $(100, 0)$, $(101, 1)$, $(110, 10)$ and $(111, 11)$. The set $\hat{G}(C)$ adds the pair $(\epsilon, 0)$. The resulting equations are

$$
\begin{aligned}
&\mathit{trans}(\mathit{trans}(\mathit{init}, a_0), a_0) = \mathit{trans}(\mathit{init}, a_0)\\
&\mathit{trans}(\mathit{trans}(\mathit{init}, a_0), a_1) = \mathit{trans}(\mathit{init}, a_1)\\
&\mathit{trans}(\mathit{trans}(\mathit{trans}(\mathit{init}, a_1), a_0), a_0) = \mathit{trans}(\mathit{init}, a_0)\\
&\mathit{trans}(\mathit{trans}(\mathit{trans}(\mathit{init}, a_1), a_0), a_1) = \mathit{trans}(\mathit{init}, a_1)\\
&\mathit{trans}(\mathit{trans}(\mathit{trans}(\mathit{init}, a_1), a_1), a_0) = \mathit{trans}(\mathit{trans}(\mathit{init}, a_1), a_0)\\
&\mathit{trans}(\mathit{trans}(\mathit{trans}(\mathit{init}, a_1), a_1), a_1) = \mathit{trans}(\mathit{trans}(\mathit{init}, a_1), a_1)\\
&\mathit{init} = \mathit{trans}(\mathit{init}, a_0)
\end{aligned}
$$

■

Note that the only property of the set $\hat{G}(C)$ we have used to prove Theorem 4.4.2 is that the resulting right congruence was equal to the right congruence generated by the transition function. We can therefore generalise Theorem 4.4.2 as follows:

Theorem 4.4.3. *Let $\mathscr{S} = (Q, \Sigma, \delta, q_0)$ be a semiautomaton, let $G \subseteq \equiv_\delta$ be a set and let $\equiv_G$ be the smallest right congruence that contains G. If $\equiv_G = \equiv_\delta$, then the algebra $A(\mathscr{S})$ is an initial D-algebra for the algebraic specification $D = (\Pi_\Sigma, \varnothing, E(G))$.*

It turns out that the converse of Theorem 4.4.3 also holds. That is, if the semiautomaton algebra is an initial object in the algebraic specification category generated by the set G, then the δ-equivalence of the semiautomaton is also the smallest right congruence containing G:

Theorem 4.4.4. *Let $\mathscr{S}$, G, $\equiv_G$ and D be as in Theorem 4.4.3. If $A(\mathscr{S})$ is an initial D-algebra, then $\equiv_G = \equiv_\delta$.*

Proof. Assume that $A(\mathscr{S})$ is an initial D-algebra. By initiality, there exists a Π_Σ-homomorphism $h : A(\mathscr{S}) \to T(D)$. We shall first prove, by induction on the length of a word, that for all $w \in \Sigma^*$, $h(\delta(q_0, w)) = [\pi(w)]_D$.

Base Case

$$\begin{aligned} h(\delta(q_0, \epsilon)) &= h(q_0) \\ &= h(init^{A(\mathscr{S})}) \\ &= init^{T(D)} \\ &= [init]_D \end{aligned}$$

Step Let $w \in \Sigma^*$ be a word and let $\sigma \in \Sigma$ be a symbol. Then

$$\begin{aligned} h(\delta(q_0, w\sigma)) &= h(\delta(\delta(q_0, w), \sigma)) \\ &= h(trans^{A(\mathscr{S})}(\delta(q_0, w), \sigma)) \\ &= trans^{T(D)}(h(\delta(q_0, w)), h(\sigma)) \\ &= trans^{T(D)}([\pi(w)]_D, [a_\sigma]_D) \\ &= [trans(\pi(w), a_\sigma)]_D \\ &= [\pi(w\sigma)]_D \end{aligned}$$

We can now prove that $\equiv_\delta \subseteq \equiv_G$. Assume $w_1 \equiv_\delta w_2$ for some words $w_1, w_2 \in \Sigma^*$. Then, by definition of $\equiv_\delta$, $\delta(q_0, w_1) = \delta(q_0, w_2)$. Applying h we get $h(\delta(q_0, w_1)) = h(\delta(q_0, w_2))$, which, by the claim proved above, implies that $[\pi(w_1)] = [\pi(w_2)]$ (or equivalently, $\pi(w_1) \sim^D \pi(w_2)$). From Theorem 4.3.1 we know that $\pi(w_1) \sim^D \pi(w_2)$ if and only if $w_1 \equiv_G w_2$.

The other inclusion is immediate: we know that $\equiv_\delta$ is a right congruence that contains G, and that $\equiv_G$ is the smallest right congruence containing G. Therefore $\equiv_G \subseteq \equiv_\delta$. □

4.5 Infinite Semiautomata

So far we have seen an example of a finite semiautomaton (the shift register), which resulted in a finite set of equations for the algebraic specifications. Many abstract data types, however, are infinite (at least in theory). Consider, for example, a very simple infinite data type: a unary counter. This ADT has two operations, increment and decrement, and an infinite number of states, one for each natural number counted. An infinite semiautomaton for such a counter appears in Figure 4.2.

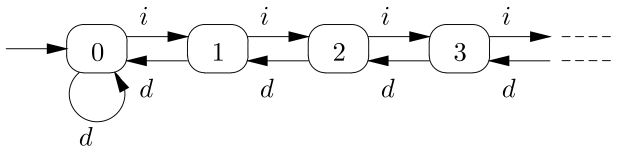


Figure 4.2: An infinite unary counter. The symbol i stands for increment, and the symbol d for decrement. A state's name reflects the net number of increments leading to that state.

Since the state set of an infinite semiautomaton is infinite, so is any chosen canonical set. This, in turn, may result in an infinite set $\hat{G}(C)$. While this imposes no problems on the mathematics used in the previous section (all theorems are valid whether the set $\hat{G}(C)$ is finite or infinite), the result, from a software specification point of view, may be impractical. Remember that the purpose of algebraic specifications is to describe software modules, and an infinite description is generally of little use.

So far we have not used one aspect of algebraic specifications in our discussion of semiautomata. Free variables give us the power to reduce an infinite number of equations to a finite one. This is possible due to the nature of such variables, which are always assumed to be universally quantified.

Consider the infinite counter, and assume the canonical set is $C = \{i^k | k = 0, 1, 2, \ldots\}$. The set $\hat{G}(C)$ is: (d, ϵ), (id, ϵ), (iid, i), $(iiid, ii)$, etc. We can observe that for each canonical word w we have the pair (wid, w) in $\hat{G}(C)$. We can therefore use a universally quantified variable c of sort *state* to substitute an infinite number of equations with the following two:

$$
\begin{aligned}
&\mathit{trans}(\mathit{trans}(c, a_i), a_d) = c \\
&\mathit{trans}(\mathit{init}, a_d) = \mathit{init}
\end{aligned}
$$

In Chapter 5 we discuss another method for finding axioms containing free variables. This method, although demonstrated for use with finite automata, may prove useful in the infinite case as well.

4.6 Output Values

In the last section of this chapter we turn our attention to Mealy machines, that is, to state machines with output values. As mentioned in Chapter 2, infinite, deterministic Mealy machines were chosen as the model for specifying abstract data types with state machines.

Since most of the work presented here concentrated on semiautomata, we do not provide a comprehensive theory on obtaining algebraic specifications from Mealy machines. Instead, we provide some notes and present an open problem that is left for further research.

First, we consider a significant difference between modules specified by Mealy machines to those described using algebraic specifications. In a general Mealy machine, an input symbol may result in both a state change and an output value. Such behaviour cannot be modelled by algebraic specifications: every function has only one sort in its range. In a specification of a stack, if a function has *stack* as its range sort, then this function is used to change the internal state of the module. Otherwise, it is a function that retrieves some information from the module. In programming terminology, functions of the first kind are "modifiers" and of the second kind are "accessors". While this property seems to be a limitation of algebraic specifications when compared with Mealy machines, it is actually a part of their strength as a formal method: it is always a good practice to differentiate between modifiers and accessors, so that the user is guaranteed that an accessor never has side effects. The user of the module can always obtain a result equivalent to the one achieved by a modifier–accessor function (such as a *pop* method in a stack, which also returns the value of the popped item) by combining two or more interface functions.

We therefore assume that the alphabet Σ of a Mealy machine can be partitioned into two sets Σ_S and Σ_V, such that the letters in Σ_S do not produce output and the letters in Σ_V do not change the state of the machine (by partition, we mean that $\Sigma = \Sigma_S \cup \Sigma_V$ and $\Sigma_S \cap \Sigma_V = \varnothing$). Note that it is easy to convert any Mealy machine to a machine that possesses this property. Let $\mathscr{M} = (Q, \Sigma, \delta, q_0, \Theta, \theta)$ be a Mealy machine. We now construct a new Mealy machine $\mathscr{M}' = (Q, \Sigma', \delta', q_0, \Theta, \theta')$ by replacing every symbol $\sigma \in \Sigma_S \cap \Sigma_V$ with two symbols σ_S, σ_V such that for all $q \in Q$

$$\begin{aligned}
\delta'(q, \sigma_S) &= \delta(q, \sigma) \\
\delta'(q, \sigma_V) &= q \\
\theta'(q, \sigma_V) &= \theta(q, \sigma)
\end{aligned}$$

and $\theta'(q, \sigma_S)$ is not defined. Let $\varphi : \Sigma \to (\Sigma')^*$ be a function defined as:

$$\varphi(\sigma') = \begin{cases} \sigma_S \sigma_V & \sigma' = \sigma \\ \sigma & \sigma' \neq \sigma \end{cases}$$

and let $\varphi^* : \Sigma^* \to (\Sigma')^*$ be the unique homomorphic extension of φ. Then for every word $w \in \Sigma^*$ and every state $q \in Q$, $\delta'(q, \varphi^*(w)) = \delta(q, w)$ and $\theta'(q, \varphi^*(w)) = \theta(q, w)$.

From now on we assume that a Mealy machine's alphabet can be partitioned as mentioned above.

In order to create a specification for a Mealy machine, we first need to modify the signature in Definition 4.2.1. This is done by splitting the sort *alpha* in two and adding a sort for output values, a constant for each output symbol and a value function.

Definition 4.6.1. Let $\mathscr{M} = (Q, \Sigma, \delta, q_0, \Theta, \theta)$ be a Mealy machine. Define the signature $\Pi_{\Sigma,\Theta} = (S, F)$ as follows:

$$\begin{aligned}
S_{\Sigma,\Theta} &= \{\textit{state}, \textit{alpha-state}, \textit{alpha-val}, \textit{output}\} \\
F_{\Sigma,\Theta} &= \{a_\sigma :\to \textit{alpha-state} \quad \forall \sigma \in \Sigma_S, \\
&\qquad a_\sigma :\to \textit{alpha-val} \quad \forall \sigma \in \Sigma_V, \\
&\qquad a_\varsigma :\to \textit{output} \quad \forall \varsigma \in \Theta, \\
&\qquad \textit{init} :\to \textit{state}, \\
&\qquad \textit{trans} : \textit{state}, \textit{alpha-state} \to \textit{state}, \\
&\qquad \textit{value} : \textit{state}, \textit{alpha-val} \to \textit{output}\}
\end{aligned}$$

The reason for separating the alphabet into two sorts is that algebraic specifications cannot define *value* as a partial function. An alternative signature can use a unary function $a_\varsigma : \textit{state} \to \textit{output}$ for every alphabet symbol producing a value.

Example 4.6.1. A Mealy machine for a two-bit shift register is depicted in Figure 4.3. The alphabet symbols l and r, when applied by the output function on a state, return the contents of the left and right register cells, respectively. Note that the output alphabet $\Theta = \{\mathbf{0}, \mathbf{1}\}$ is distinct from the input alphabet $\Sigma = \{0, 1\}$. The signature $\Pi_{\Sigma,\Theta}$ contains two constants a_0, a_1 of sort *alpha-state*, two constants a_l, a_r of sort *alpha-val* and two constants $a_{\mathbf{0}}, a_{\mathbf{1}}$ of sort *output*. ■

A significant problem that arises from this definition is that words over Σ^* can no longer be converted into terms. Consider, for example, the word

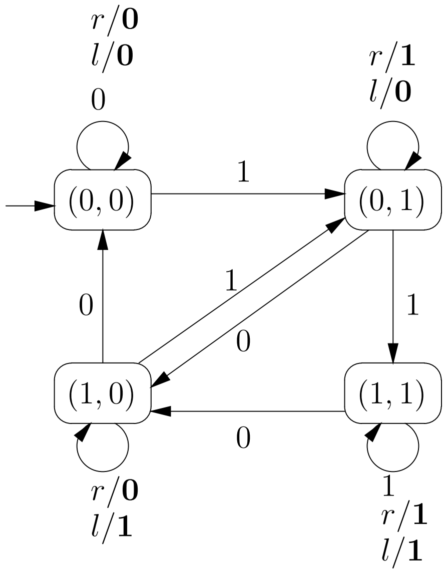


Figure 4.3: The shift register Mealy machine. The output alphabet consists of the symbols **0** and **1**.

$$w = \sigma_1\sigma_2\sigma_3\varsigma_1\sigma_4\varsigma_2$$

where $\sigma_i \in \Sigma_S$ and $\varsigma_i \in \Sigma_V$. The 4 letter prefix of this word, $\sigma_1\sigma_2\sigma_3\varsigma_1$, can be translated into the term

$$\mathit{value}(\mathit{trans}(\mathit{trans}(\mathit{trans}(\mathit{new}, a_{\sigma_1}), a_{\sigma_2}), a_{\sigma_3}), a_{\varsigma_1})$$

However, this term is of sort *output*, and cannot be a subterm of *trans*. This seems to be a shortcoming of algebraic specifications, which does not allow this method to model the concept of a "trace" (a sequence of function executions on the module). To correctly translate the word w, we first need to remove all occurrences of letters in Σ_V.

Writing axioms for a Mealy machine specification is a straightforward extension of the procedure described for semiautomata. Ostensibly, the generalised Nerode equivalence can give us further information and simplify equations generated for the underlying semiautomaton. Alternatively, we can first obtain the reduced automaton and then write the axioms for that machine, including those for the underlying semiautomaton. It is safe to assume that the second method is at least as easy as the first. We shall therefore consider all Mealy machines as having been reduced by a preprocessing stage of the specification translation procedure.

The only task that remains to be done for a reduced Mealy machine is to specify the value of every term of the sort *output*. The only way to obtain such terms is by applying the function name *value* on a term of sort *state* and a constant of sort *alpha-val* (there are no terms of sort *alpha-val* but the constants). We therefore end up with a set of equations of the form

$$\mathit{value}(t, a) = o$$

where t is a term of sort *state*, a is a constant of sort *alpha-val* and o is a term of sort *output*. Note that we need to write such an equation only once for every equivalence class of terms of sort *state*: if $t_1 \sim^D t_2$, then

$$[\mathit{value}(t_1, a)]_D = \mathit{value}^{T(D)}([t_1]_D, [a]_D) = \mathit{value}^{T(D)}([t_2]_D, [a]_D) = [\mathit{value}(t_2, a)]_D$$

which means, by the initiality of $T(D)$, that for every D-algebra $\mathit{value}(t_1, a)$ and $\mathit{value}(t_2, a)$ are interpreted the same way. Thus, it is sufficient to include one axiom for every term produced from a canonical set (see Section 4.2) and every constant of sort *alpha-val*.

Example 4.6.2. Consider the Mealy machine for a 2-bit shift register, depicted in Figure 4.3. The value producing symbols in the machine's alphabet are r, which returns the contents of the right cell, and l, which returns the contents of the left cell. The axioms for the underlying semiautomaton are given in Example 4.4.1, which assumes the canonical set to be $C = \{0, 1, 10, 11\}$. The resulting axioms for output values are

$$\begin{aligned}
&\mathit{value}(\mathit{trans}(\mathit{new}, a_0), a_r) = a_{\mathbf{0}}\\
&\mathit{value}(\mathit{trans}(\mathit{new}, a_0), a_l) = a_{\mathbf{0}}\\
&\mathit{value}(\mathit{trans}(\mathit{new}, a_1), a_r) = a_{\mathbf{1}}\\
&\mathit{value}(\mathit{trans}(\mathit{new}, a_1), a_l) = a_{\mathbf{0}}\\
&\mathit{value}(\mathit{trans}(\mathit{trans}(\mathit{new}, a_1), a_0), a_r) = a_{\mathbf{0}}\\
&\mathit{value}(\mathit{trans}(\mathit{trans}(\mathit{new}, a_1), a_0), a_l) = a_{\mathbf{1}}\\
&\mathit{value}(\mathit{trans}(\mathit{trans}(\mathit{new}, a_1), a_1), a_r) = a_{\mathbf{1}}\\
&\mathit{value}(\mathit{trans}(\mathit{trans}(\mathit{new}, a_1), a_1), a_l) = a_{\mathbf{1}}
\end{aligned}$$

■

Note that we cannot combine the second and fourth equations in Example 4.6.2 into something like

$$\mathit{value}(\mathit{trans}(\mathit{new}, x), a_l) = x$$

since x cannot be at the same time both a variable of sort *alpha-state* and a variable of sort *output*. We can, however, add an input-to-output alphabet conversion function with axioms such as

$$\mathit{convert}(x, 0) = \mathbf{0}$$

and then write

$$\mathit{value}(\mathit{trans}(\mathit{new}, x), a_l) = \mathit{convert}(x)$$

where x is a free variable of sort *alpha-state*. Generally speaking, when specifying a Mealy machine in which the output alphabet is a subset of the input alphabet one should be cautious with assigning the same sort to both alphabets: while the constants of both sorts may be the same, the set of terms produced for each sort can be different, leading to valid syntactical forms, which are semantically incorrect.

As a final note on output values, we consider the following alternative set of axioms for specifying the two-bit shift register:

Example 4.6.3. Let x be a free variable of sort *state* and let y be a free variable of sort *alpha-val*. Let $X = \{x, y\}$ be a set of free variables. The following axioms are defined over $T(\Pi_{\Sigma,\Theta}, X)$, where $\Pi_{\Sigma,\Theta}$ is the signature defined in Example 4.6.1:

$$\begin{aligned}
&\mathit{value}(\mathit{trans}(x, a_0), a_r) = a_{\mathbf{0}}\\
&\mathit{value}(\mathit{trans}(x, a_1), a_r) = a_{\mathbf{1}}\\
&\mathit{value}(\mathit{trans}(\mathit{trans}(x, a_0), y), a_l) = a_{\mathbf{0}}\\
&\mathit{value}(\mathit{trans}(\mathit{trans}(x, a_1), y), a_l) = a_{\mathbf{1}}
\end{aligned}$$

■

It can be easily verified that the set of axioms in Example 4.6.3, when added to the set of semiautomaton axioms in Example 4.4.1, completely specifies the shift register Mealy machine. Moreover, from an observational point of view, which is the only one that matters for software specification, we can omit the semiautomaton axioms. We would then have a specification of a Mealy machine with an infinite number of states (as there are no axioms equating two terms of sort *state*), which, once we add a couple of axioms for the values on *init*, behaves in exactly the same way as the 4-state machine. Whether this set of axioms can be obtained algorithmically remains an open question.

Chapter 5

Varieties

5.1 Introduction

In the previous chapter we have introduced a method for converting automaton specifications into algebraic specifications. While the method was proved to be correct in all cases, it often produces poor results, in the sense that the derived algebraic specification is lengthy and cluttered. Such a result was demonstrated in Example 4.4.1, where a shift register was specified using 5 axioms, each containing several subterms. It may be suspected (justifiably, as it turns out), that such a simple data structure can be specified in a more compact and elegant manner.

The most important reason for these rather poor results is that the aforementioned method completely ignores one of the fundamental mechanisms in algebraic specifications, namely the use of free variables. Such variables can turn axioms into universally quantified statements, thus replacing several axioms (one for each term of a given sort) with a single equation.

But how does one obtain such axioms from a semiautomaton? In Section 4.5 it was suggested that by looking at the axioms one can sometimes discover patterns that lead to the aggregation of several axioms. In this chapter we will describe another method for writing axioms containing free variables by using prior knowledge on classes of monoids.

While semiautomata and monoids are different algebraic structures, they are closely related through the concept of a transformation monoid (see Definition 2.3.6). This relation allows us to investigate the properties of a transformation monoid and apply the findings to its semiautomaton. (Such a course of action is widely used in automaton theory. See, for example, [24].) Our main tool of investigation will be *varieties*.

A variety of semigroups is a class of semigroups closed under finite direct products and under homomorphic images of subsemigroups. In one of the seminal papers in algebra, Birkhoff has shown [2] that varieties can be described by equations composed of strings of free variables. For example, the equation

$$xy = yx$$

describes all commutative semigroups. Birkhoff proved that all semigroups on which this equation is applicable form a variety, and that every variety can be described by a set of equations. Eilenberg later proved the same result [10] on classes of finite monoids, known as $\mathbf{M}$-varieties. Furthermore, Eilenberg has shown that every finite monoid gives rise to an $\mathbf{M}$-variety.

Birkhoff's work immediately suggests a resemblance between a variety and the category $\mathbf{Alg}(D)$: both are classes of algebraic structures satisfying a common set of equations. (Indeed, the resemblance in not coincidental, as this work greatly inspired the developers of algebraic specifications.) It is therefore interesting to see in what way variety equations can be translated into axioms, and of what use they are in this case. In this chapter we show that equations describing $\mathbf{M}$-varieties can be helpful in simplifying algebraic specifications of semiautomata.

The key theorem in this chapter can be summarised as follows: Let $\mathscr{S}$ be a semiautomaton, and let $T_{\mathscr{S}}$ be its transformation monoid. The monoid $T_{\mathscr{S}}$ belongs to several $\mathbf{M}$-varieties. Assume that we know the equations defining one or more of these varieties. Then a straightforward translation of these equations into Π_Σ-equations (using a suitable set of free variables) results in axioms which are valid in the semiautomaton's algebra.

An important restriction on the usefulness of this theorem is that one has to know the equations defining a certain variety in order to use it. It can be argued that finding equations for varieties is no easier than finding axioms for algebraic specifications, thus doubting any real-world application of the theorem. Nevertheless, the theorem does not require one to investigate a specific $\mathbf{M}$-variety in order to get results. Instead, it is sufficient that the transformation monoid belongs to *any* well-studied variety for the theorem to provide information that can be used to simplify the specification of the semiautomaton.

5.2 Varieties

This section contains a brief introduction to varieties, focusing on pseudo-varieties of finite monoids. Definitions and results were adapted from [10].

Definition 5.2.1. A semigroup T *divides* a semigroup S if there is a subsemigroup S' of S and a mapping $h : S' \to T$ such that h is a semigroup homomorphism. (In other words, T divides S if T is a homomorphic image of a subsemigroup of S.)

Definition 5.2.2. The *direct product* of two semigroups $(S, \cdot)$ and $(T, *)$ is a pair $(S \times T, \circ)$, where $\circ$ is a binary operation defined as

$$(s,t) \circ (s',t') = (s \cdot s', t * t')$$

for all $s, s' \in S$ and $t, t' \in T$.

Proposition 5.2.1. *The direct product of two semigroups is a semigroup.*

Definition 5.2.3. A variety $\mathbf{V}$ is a family of semigroups closed under divisions and direct products. That is, if $S, S' \in \mathbf{V}$ are semigroups, then

- If T is a semigroup that divides S, then T is in $\mathbf{V}$.
- The direct product of S and S' is in $\mathbf{V}$.

Definition 5.2.4. An $\mathbf{S}$-*variety* (or a *pseudovariety*) is a variety restricted to finite direct products.

Definition 5.2.5. An $\mathbf{M}$-*variety* is an $\mathbf{S}$-variety whose members are all monoids.

Note that the above definition implies that an $\mathbf{M}$-variety is *not* a variety *per se*, as it is not closed under semigroup divisions. The reason is that a subsemigroup of a monoid need not be a monoid. Eilenberg defines an $\mathbf{M}$-variety as the intersection of an $\mathbf{S}$-variety with the set of all monoids. We will use a different definition which, hopefully, will make things clearer:

Definition 5.2.6. An $\mathbf{M}$-*variety* is family $\mathbf{V}$ of monoids such that:

- If $M, M' \in \mathbf{V}$, then the direct product of M and M' is also in $\mathbf{V}$
- If $M \in \mathbf{V}$ and M' is a *submonoid* of M', then any homomorphic image of M' is also in $\mathbf{V}$

Definition 5.2.7. Let M be a monoid. The $\mathbf{M}$-variety *generated* by M is the least $\mathbf{M}$-variety containing M.

One of the earliest results in abstract algebra, which is due to Birkhoff [2], states that a family of semigroups is a variety if and only if it is defined by a set of equations. We will use this fact later in this chapter to establish a connection between semigroup equations and axioms in algebraic specifications.

Definition 5.2.8. Let Σ^* be the free monoid generated by a denumerable alphabet Σ, and let $u, v \in \Sigma^*$. A monoid M *satisfies* the equation $u = v$ if for every homomorphism $h : \Sigma^* \to M$, $h(u) = h(v)$.

Proposition 5.2.2. *Let* $\mathbf{V}(u, v)$ *be the family of all monoids satisfying the equation* $u = v$*. Then* $\mathbf{V}(u, v)$ *is an* $\mathbf{M}$*-variety.*

Proof. See [10], §V.2. □

The above proposition can be easily extended to a set $E \subseteq \Sigma^* \times \Sigma^*$ of equations. Thus, if E is such a set, then $\mathbf{V}(E)$ is the variety of all monoids satisfying every equation in E.

Definition 5.2.9. An $\mathbf{M}$-variety $\mathbf{V}$ is *defined by a set of equations* E if $\mathbf{V} = \mathbf{V}(E)$. If there exists a set of equations E that defines $\mathbf{V}$, we say that $\mathbf{V}$ is *equational.*

Theorem 5.2.1. *Let* M *be a monoid. Then the* $\mathbf{M}$*-variety generated by* M *is equational.*

Proof. See [10], §V.2. □

5.3 The Transformation Monoid as an Algebra

Before any results on monoids can be used in the algebraic specifications of semiautomata, we need to establish a common framework to handle both structures. Fortunately, transformation monoids can be easily made into algebras for the semiautomaton signature Π_Σ. Thus we can use treat both structures as algebras over the same signature.

Definition 5.3.1. Let $\mathscr{S} = (Q, \Sigma, \delta, q_0)$ be a semiautomaton with a transformation monoid $T_{\mathscr{S}} = (\Sigma^*/{\equiv_T}, nat_T, [\epsilon])$. Make $T_{\mathscr{S}}$ into a Π_Σ-algebra $A(T_{\mathscr{S}})$ by defining $\Sigma^*/{\equiv_T}$ as the carrier set elements of sort *state*, Σ as the carrier set elements of sort *alpha* and the following functions:

- $init^{A(T_{\mathscr{S}})} = [\epsilon]$
- For all $u \in \Sigma^*, \sigma \in \Sigma$, $trans^{A(T_{\mathscr{S}})}([u], \sigma) = [u\sigma]$

Before we establish the relationship between a semiautomaton algebra and its transformation monoid algebra, we need the following general result:

Lemma 5.3.1. *Let $D = (\Pi, X, E)$ be a specification, where Π is a signature, X a set of free variables and E a set of equations over Π and X. Let I be an initial D-algebra and let A be a reachable Π-algebra. Then A is a D-algebra if and only if there exists a surjective Π-homomorphism $h : I \rightarrow A$.*

Proof. Assume A is a D-algebra. Since I is initial in the category $\mathbf{Alg}(D)$, there exists a unique Π-homomorphism $h : I \rightarrow A$. Consider the interpretation function $\nu_I : T(\Pi) \rightarrow I$. This function is a Π-homomorphism, and therefore the composition $h \circ \nu_I : T(\Pi) \rightarrow A$ is also a Π-homomorphism. The algebra $T(\Pi)$ is initial in the category $\mathbf{Alg}(\Pi)$, and therefore there is only one Π-homomorphism between $T(\Pi)$ and A, namely the interpretation function ν_A^*. The algebra A was assumed to be reachable, which means that ν_A^* is surjective. Since ν_A^* is unique, we have $\nu_A^* = h \circ \nu_I$, and hence h must also be surjective.

Conversely, assume there is a surjective Π-homomorphism $h : I \rightarrow A$. We first show that for any valuation $\nu_A : X \rightarrow A$ there is a valuation $\nu_I : X \rightarrow I$ such that $\nu_A = h \circ \nu_I$.

Let $\nu_A : X \rightarrow A$ be a valuation. Since h is surjective, we can define a function $h^{-1} : A \rightarrow I$ such that $h \circ h^{-1} = Id_A$ (where Id_A is the identity on A). Note that h^{-1} may be defined in more than one way (e.g., if both $h(i_1) = a$ and $h(i_2) = a$, we can define either $h^{-1}(a) = i_1$ or $h^{-1}(a) = i_2$), but it does not matter for our purpose. Define $\nu_I = h^{-1} \circ \nu_A$. By composing both sides with h we get $\nu_A = h \circ \nu_I$. Moreover, ν_I is clearly a valuation from X to I, and thus we obtain the required result.

Consider now the unique homomorphic extensions of ν_A and ν_I, namely $\nu_A^* : T(\Pi, X) \rightarrow A$ and $\nu_I^* : T(\Pi, X) \rightarrow I$. Since h is a Π-homomorphism, so is $h \circ \nu_I^*$. Let $x \in X$ be a free variable. Then

$$\begin{aligned} h \circ \nu_I^*(x) &= h(\nu_I^*(x)) \\ &= h(\nu_I(x)) && \text{(by homomorphic extension)} \\ &= h \circ \nu_I(x) \\ &= \nu_A(x) \end{aligned}$$

This shows that $h \circ \nu_I^*$ is a homomorphic extension of ν_A, and by the uniqueness of the extension it must follow that $\nu_A^* = h \circ \nu_I^*$. We thus conclude that, for every interpretation ν_A^*, there is an interpretation ν_I^* such that $\nu_A^* = h \circ \nu_I^*$.

Let (t_1, t_2) be an equation in E, and let $\nu_A^* : T(\Pi, X) \to A$ be an interpretation. By the above conclusion, there is an interpretation $\nu_I^* : T(\Pi, X) \to I$ such that $\nu_A^* = h \circ \nu_I^*$. Furthermore, since I is a D-algebra, $\nu_I^*(t_1) = \nu_I^*(t_2)$, and since h is a Π-homomorphism, $h(\nu_I^*(t_1)) = h(\nu_I^*(t_2))$. Hence the equation (t_1, t_2) is valid in A, which makes A a D-algebra. □

Recall that for every semiautomaton $\mathscr{S}$ there is an isomorphic semiautomaton $\mathscr{S}_\delta$ which is generated from the right-congruence $\equiv_\delta$ (see Definition 2.3.4 and Theorem 2.3.1). As in other cases, it is easier to first prove the main result of this section on the semiautomaton $\mathscr{S}_\delta$, and then use the isomorphism property to infer a similar result on $\mathscr{S}$.

Lemma 5.3.2. *Let $\mathscr{S} = (Q, \Sigma, \delta, q_0)$ be a semiautomaton and let $T_{\mathscr{S}}$ be its transformation monoid. Let $\mathscr{S}_\delta$ be the semiautomaton generated from $\mathscr{S}$ by the right-congruence $\equiv_\delta$. If D is a specification such that $A(T_{\mathscr{S}})$ is an initial D-algebra, then $A(\mathscr{S}_\delta)$ is a D-algebra.*

Proof. Define an S-sorted mapping $h : A(T_{\mathscr{S}}) \to A(\mathscr{S}_\delta)$ as follows:

- For all $\sigma \in \Sigma$, $h_{alpha}(a_\sigma) = \sigma$
- For all $w \in \Sigma^*$, $h_{state}([w]_T) = [w]_\delta$

We first note that h_{state} is a well-defined function from $\Sigma^*/\equiv_T$ to $\Sigma^*/\equiv_\delta$, since for all $u, v \in \Sigma^*$, $[u]_T = [v]_T$ implies $[u]_\delta = [v]_\delta$. Furthermore, h_{state} is surjective: every class $[u]_\delta$ is defined by at least one word $u \in \Sigma^*$, which, in turn, gives rise to a class $[u]_T$. By the definition of h_{state}, $h([u]_T) = [u]_\delta$, and thus h_{state} covers all classes in $\Sigma/\equiv_\delta$.

We now claim that h is a Π_Σ-homomorphism. The claim holds trivially for all elements of sort *alpha*. The proof on h_{state} proceeds by structural induction:

Base Case The only constant in Π_Σ of sort *state* is *init*. By definition, $init^{A(T_{\mathscr{S}})} = [\epsilon]_T$ and $init^{A(\mathscr{S}_\delta)} = [\epsilon]_\delta$, and so $h_{state}(init^{A(T_{\mathscr{S}})}) = init^{A(\mathscr{S}_\delta)}$.

Step Let $q \in T(\Pi_\Sigma)_{state}, a_\sigma \in T(\Pi_\Sigma)_{alpha}$ be terms. The algebra $A(T_\mathscr{S})$ is a reachable Π_Σ-algebra (by its initiality), and therefore there is a word $u \in \Sigma^*$ such that $[u]_T = q^{A(T_\mathscr{S})}$. Consider the term $trans(q, a_\sigma)$. By the definition of h_{state}, $h([u]_T) = [u]_\delta$, and by the induction hypothesis $[u]_\delta = q^{A(\mathscr{S}_\delta)}$. Therefore

$$\begin{aligned} h(\nu_T^*(trans(q, a_\sigma))) &= h(trans^{A(T_autoS)}([u]_T, \sigma)) \\ &= h([u\sigma]_T) \\ &= [u\sigma]_\delta \\ &= \nu_\delta^*(trans(q, a_\sigma))) \end{aligned}$$

where ν_T^*, ν_δ^* are the unique ground term interpretation functions for the algebras $A(T_\mathscr{S}), A(\mathscr{S}_\delta)$, respectively.

Since h is a surjective Π_Σ-homomorphism, we can now use Lemma 5.3.1 to conclude that $A(\mathscr{S}_\delta)$ is a D-algebra. □

Theorem 5.3.1. *Let $\mathscr{S} = (Q, \Sigma, \delta, q_0)$ be a semiautomaton and let $T_\mathscr{S}$ be its transformation monoid. If D is a specification such that $A(T_\mathscr{S})$ is an initial D-algebra, then $A(\mathscr{S})$ is a D-algebra.*

Proof. From Lemma 5.3.2 and the isomorphism of $A(\mathscr{S})$ and $A(\mathscr{S}_\delta)$. □

Corollary 5.3.1. *Any Π_Σ-equation over some set X of free variables which is valid in $A(T_\mathscr{S})$ is also valid in $A(\mathscr{S})$.*

Proof. Take all equations E over X which are valid in $A(T_\mathscr{S})$, and create a specification $D = (\Pi_\Sigma, X, E)$. Then $A(T_\mathscr{S})$ is an initial D-algebra. From Theorem 5.3.1 it follows that all equations E are valid in $A(\mathscr{S})$. □

5.4 Monoid-Derived Axioms for Semiautomata

Variety equations, as defined in Section 5.2, cannot be used in the algebraic specifications of semiautomata for the obvious reason that such equations are not composed of Π_Σ-terms. However, a simple transformation can be used to obtain proper axioms:

Definition 5.4.1. Let $\Pi_\Sigma = (S, F)$ be the semiautomaton signature over alphabet Σ, and let X be an S-sorted set such that X_{alpha} is a denumerable set and $X_{state} = \{q\}$. Let Ξ be a denumerable set, and let $\pi : \Xi \to X_{alpha}$ be a bijective function. The function $\pi^* : \Xi^* \to T(\Pi_\Sigma, X)$ is defined recursively:

- $\pi^*(\epsilon) = q$
- For all $u \in \Xi^*, x \in \Xi$, $\pi(ux) = \mathit{trans}(\pi^*(u), \pi(x))$

Recall that the monoid property of satisfying equations is defined through monoid homomorphisms. The following proposition shows that the function composition $\nu^* \circ \pi^*$ behaves very much like a monoid homomorphism, in the sense that one can always extract the homomorphic extension of $\nu \circ \pi$ from the image of $\nu^* \circ \pi^*$. We will use this behaviour in the proof of the next result.

Proposition 5.4.1. *Let $A(T_{\mathscr{S}})$ be a Π_Σ-algebra for the transformation monoid of some semiautomaton $\mathscr{S}$. For all words $u = x_1...x_n \in \Xi^*$ and for all assignments $\nu : X \to A(T_{\mathscr{S}})$,*

$$\nu^*(\pi^*(u)) = \mathit{nat}_T(\nu(q), [\nu(\pi(x_1))...\nu(\pi(x_n))])$$

where $\nu^* : T(\Pi_\Sigma, X) \to A(\mathscr{S})$ is the extension of ν to an interpretation function.

Proof. By induction on the length of u.

Base Case For $u = \epsilon$ we get

$$\begin{aligned}\nu^*(\pi^*(\epsilon)) &= \nu^*(q)\\ &= \nu(q)\\ &= \mathit{nat}_T(\nu(q), [\epsilon])\end{aligned}$$

Step Assume the proposition holds for some $u = x_1...x_n \in \Xi^*$, and let $x \in \Xi$. Then

$$\begin{aligned}
\nu^*(\pi^*(ux)) &= \nu^*(\mathit{trans}(\pi^*(u), \pi(x))) && \text{(def. of } \pi^*\text{)}\\
&= \mathit{trans}^{A(T_{\mathscr{S}})}(\nu^*(\pi^*(u)), \nu^*(\pi(x))) && \text{(def. of } \nu^*\text{)}\\
&= \mathit{trans}^{A(T_{\mathscr{S}})}(\nu^*(\pi^*(u)), \nu(\pi(x))) && \text{(def. of } \nu^*\text{)}\\
&= [\nu^*(\pi^*(u))\nu(\pi(x))] && \text{(def. of } A(T_{\mathscr{S}})\text{)}\\
&= [\mathit{nat}_T(\nu(q), [\nu(\pi(x_1))...\nu(\pi(x_n))])\nu(\pi(x))] && \text{(hypothesis)}\\
&= \mathit{nat}_T(\mathit{nat}_T(\nu(q), [\nu(\pi(x_1))...\nu(\pi(x_n))]), [\nu(\pi(x))]) && \text{(def. of } \mathit{nat}_T\text{)}\\
&= \mathit{nat}_T(\nu(q), \mathit{nat}_T([\nu(\pi(x_1))...\nu(\pi(x_n))], [\nu(\pi(x))])) && \text{(associativity)}\\
&= \mathit{nat}_T(\nu(q), [\nu(\pi(x_1))...\nu(\pi(x_n))\nu(\pi(x))]) && \text{(def. of } \mathit{nat}_T\text{)}
\end{aligned}$$

□

We are now ready to prove that the equation transformation procedure, as described in Definition 5.4.1, preserves the satisfaction property: if a monoid satisfies an equation, then the translation of this equation is a valid axiom in the Π_Σ-algebra of this monoid.

Lemma 5.4.1. *Let $\mathscr{S} = (Q, \Sigma, \delta, q_0)$ be a semiautomaton with the transformation monoid $T_{\mathscr{S}} = (\Sigma^*/{\equiv_T}, nat_T, \epsilon)$. Then for every equation (u, v) over Ξ^* which is satisfied by $T_{\mathscr{S}}$, the Π_Σ-equation $(\pi^*(u), \pi^*(v))$ is valid in $A(T_{\mathscr{S}})$.*

Proof. Let $\nu : X \to A(T_{\mathscr{S}})$ be an assignment. Define a function $\psi^* : \Xi^* \to \Sigma^*$ such that for all $u \in \Xi^*$

$$\nu^*(\pi^*(u)) = nat_T(\nu(q), [\psi^*(u)])$$

From Proposition 5.4.1 it is clear that ψ^* is the unique homomorphic extension of $\nu_{alpha} \circ \pi : \Xi \to \Sigma^*$ (while ν_{alpha} was defined with range Σ, it can certainly be treated as a function from X to Σ^*).

If $u = v$ is an equation satisfied by $T_{\mathscr{S}}$, then $\psi^*(u) = \psi^*(v)$ and hence

$$\nu^*(\pi^*(u)) = nat_T(\nu(q), [\psi^*(u)]) = nat_T(\nu(q), [\psi^*(v)]) = \nu^*(\pi^*(v))$$

Thus, $\nu^*(\pi^*(u)) = \nu^*(\pi^*(v))$ for all assignments $\nu : X \to A(T_{\mathscr{S}})$, which makes the axiom $\pi^*(u) = \pi^*(v)$ valid in $A(T_{\mathscr{S}})$. □

Finally, the results in this section are combined with the results of the previous section to obtain the main theorem of this chapter:

Theorem 5.4.1. *Let $\mathscr{S} = (Q, \Sigma, \delta, q_0)$ be a semiautomaton, with the transformation monoid $T_{\mathscr{S}} = (\Sigma^*/{\equiv_T}, nat_T, \epsilon)$. Then for every equation (u, v) over Ξ^* which is satisfied by $T_{\mathscr{S}}$, the Π_Σ-equation $(\pi^*(u), \pi^*(v))$ is valid in $A(\mathscr{S})$.*

Proof. By Corollary 5.3.1 and Lemma 5.4.1. □

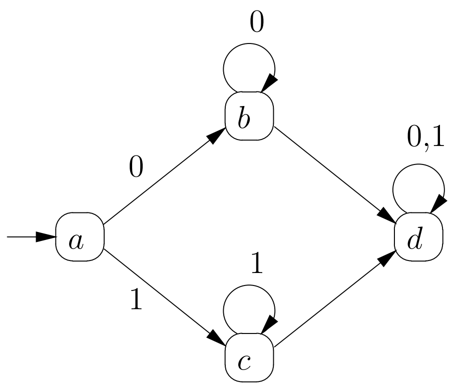


Figure 5.1: This semiautomaton realises the free commutative idempotent monoid over $\{0, 1\}$.

5.5 Examples

Let us consider two simple semiautomata, and observe how the method described above can assist in providing axioms for their specifications.

Example 5.5.1. Consider the semiautomaton in Figure 5.1. This semiautomaton realises a 4 element monoid over $\{0, 1\}$ with the following multiplication table

ϵ	0	1	01
0	0	01	01
1	01	1	01
01	01	01	01

This monoid is the free commutative idempotent monoid over $\{0, 1\}$. It satisfies the equations $uu = u$ (idempotence) and $uv = vu$ (commutativity). We can then add the two axioms

$$trans(trans(q, x), x) = trans(q, x)$$
$$trans(trans(q, x), y) = trans(trans(q, y), x)$$

to the algebraic specifications of the semiautomaton (with x, y as free variables of sort *alpha*). Since the semiautomaton realises the monoid, these are the only axioms required in the specification.

Example 5.5.2. The shift register semiautomaton was defined in Example 4.1.1. The transformation monoid of the shift register has 7 elements and can be described by the following multiplication table:

ϵ	0	1	00	01	10	11
0	00	01	00	01	10	11
1	10	11	00	01	10	11
00	00	01	00	01	10	11
01	10	11	00	01	10	11
10	00	01	00	01	10	11
11	10	11	00	01	10	11

The above monoid satisfies the equation $uvw = vw$, which yields the axiom

$$\mathit{trans}(\mathit{trans}(\mathit{trans}(q,x),y),z) = \mathit{trans}(\mathit{trans}(q,y),z)$$

Notice, however, that the semiautomaton in Example 5.5.2 can be described by a simpler axiom than the one achieved:

$$\mathit{trans}(\mathit{trans}(q,x),y) = \mathit{trans}(\mathit{trans}(\mathit{init},x),y)$$

The reason we were unable to get this more elegant form using the described translation method is that a significant piece of information was lost in the process. Consider the original equation

$$xyz = yz$$

We know that for all monoid homomorphisms $\varphi : \Xi^* \to \Sigma^*/{\equiv_T}$

$$\varphi(xyz) = \varphi(yz)$$

Since $\varphi(xyz)$ and $\varphi(yz)$ are both equivalence classes under $\equiv_T$, we can use the property of this relation to determine that for all monoid homomorphisms $\psi : \Xi^* \to \Sigma^*$ and for all states $q \in Q$

$$\delta(q,\psi(xyz)) = \delta(q,\psi(yz))$$

and specifically

$$\delta(q_0,\psi(xyz)) = \delta(q_0,\psi(yz))$$

We now use the homomorphism property to obtain

$$\delta(\delta(\delta(q_0,\psi(x)),\psi(y)),\psi(z)) = \delta(\delta(q_0,\psi(y))\psi(z))$$

But, since the semiautomaton is connected, and since the equation is true for all assignments of words to x, we can replace $\delta(q_0, \psi(x))$, and claim that for all states $q \in Q$

$$\delta(\delta(q, \psi(y)), \psi(z)) = \delta(\delta(q_0, \psi(y))\psi(z))$$

which suggests the simple axiom mentioned above.

The loss of information has occurred when the translation process replaced the variables in the variety equations with free variables of sort *alpha*, representing single letters in the alphabet. If x is a variable that can be set to alphabet letters only, then the universally quantified $\delta(q_0, \psi(x))$ (where ψ is the free variable) cannot be replaced by a universally quantified variable representing a state.

This limitation is a consequence of the signature chosen for representing semiautomata: the transition function takes a state and a single letter, rather than a state and a word in Σ^*. It may be interesting to see what other benefits (and perhaps disadvantages) would be to a different approach in axiomatic definitions of semiautomata.

Chapter 6

From Algebraic Specifications to Automata

6.1 Introduction

In the previous two chapters we have shown how to convert an automaton into an algebraic specification in such a way that the two specifications describe the same abstract data type. The natural question arising from this work is whether the opposite way can also be taken, that is, whether algebraic specifications can always be translated into automata.

The answer to this question, as discussed in this chapter, is somewhat equivocal. We prove that for every abstract data type described by an algebraic specification there exists an automaton describing the same, or, at least, a closely related, ADT. However, the existence of such a machine is not sufficient, as we are considering automata as a method for software specification. An automaton can provide useful information to software developers only if it can be easily described. We shall see that the automaton specification of some abstract data types is too complex to provide any meaningful description of the module. We suspect that this is an inherent problem of automata, as well as of other equivalent methods for software specification, which consider a module as a consecutive sequence of operations on the same object. Unfortunately, not all abstract data types present such behaviour.

On the other hand, the algebraic specification of such an abstract data type is clear and concise. This leads us to the belief (which is yet to be formally proved or disproved) that algebraic specifications yield a more powerful tool for software specification than automata do.

6.2 Linear Data Structures

Every well-defined algebraic specification designates a single sort in the signature as its *type of interest*. The type of interest is the sort corresponding to the abstract data type which is defined by the specification. For example, a specification of a natural number has a sort *nat* as its type of interest, a stack ADT has a **stack** sort, and so on. In this chapter we designate the type of interest in a signature by using a bold face typeface.

Definition 6.2.1. Let $\Pi = (S, F)$ be a signature with $\mathbf{s} \in S$ as the type of interest. The signature is *linear* if the following conditions hold:

1. There is exactly one constant $f :\to \mathbf{s}$;
2. For every non-constant function name $f : s_1, \ldots, s_n \to \mathbf{s} \in F$, there is exactly one $i \in \{1, \ldots, n\}$ such that $s_i = \mathbf{s}$;
3. For every non-constant function name $f : s_1, \ldots, s_n \to s \in F$ with $s \neq \mathbf{s}$, there is at most one $i \in \{1, \ldots, n\}$ such that $s_i = \mathbf{s}$.

A specification $D = (\Pi, X, E)$ is linear if the signature Π is linear.

We assume that in a linear signature the type of interest is always the first sort in the domain tuple of a function name. It should be clear that every linear signature can be converted to this form, without any change to the specification's semantics. The only reason we add this condition is to simplify the notation used in this chapter.

Example 6.2.1. The following signature, with **nat** as the type of interest, is linear:

$$\begin{aligned}
S &= \{\mathbf{nat}, \mathit{bool}\} \\
F &= \{0 :\to \mathbf{nat}, \\
&\qquad \mathit{true} :\to \mathit{bool} \\
&\qquad \mathit{false} :\to \mathit{bool} \\
&\qquad \mathit{succ} : \mathbf{nat} \to \mathbf{nat} \\
&\qquad \mathit{positive} : \mathbf{nat} \to \mathit{bool}
\end{aligned}$$

Adding a function name $1 :\rightarrow \mathbf{nat}$ or omitting the function name $0 :\rightarrow \mathbf{nat}$ would make this signature non-linear by condition 1 of the definition; The function names $\mathit{add} : \mathbf{nat}, \mathbf{nat} \rightarrow \mathbf{nat}$ and $\mathit{bool2nat} : \mathit{bool} \rightarrow \mathbf{nat}$ would make it non-linear, according to condition 2; The function name $\mathit{equal} : \mathbf{nat}, \mathbf{nat} \rightarrow \mathit{bool}$ makes it non-linear, following condition 3. On the other hand, the function name $\mathit{or} : \mathit{bool}, \mathit{bool} \rightarrow \mathit{bool}$ does not violate the linearity property, as *bool* is not the type of interest. ■

We now show how a Mealy machine can be constructed from a linear specification. We begin by setting the states to be the equivalence classes of terms of the type of interest under the congruence $\sim^D$. That is, the set of states is $T(D)_{\mathbf{s}}$, where $\mathbf{s}$ is the type of interest. Of these classes, the one holding the single constant of the type of interest is the initial state.

Next, we define the alphabet. In an automaton, an alphabet is used as part of the domain of the transition function, but the alphabet can be looked upon also as a set of functions, mapping states to states. Consider a Mealy machine $\mathscr{M} = (Q, \Sigma, q_0, \delta, \Theta, \theta)$. It is easy to see that every letter $\sigma \in \Sigma$ uniquely defines a function $f_\sigma : Q \rightarrow Q$, such that for all $q \in Q$

$$f_\sigma(q) = \delta(q, \sigma)$$

Similarly, any letter σ producing an output value defines a function $g_\sigma : Q \rightarrow \Theta$ with

$$g_\sigma(q) = \theta(q, \sigma)$$

for all $q \in Q$. Therefore, in order to construct an alphabet for the Mealy machine, we need to turn quotient term algebra functions, taking multiple parameters, into functions from $T(D)_{\mathbf{s}}$ to $T(D)_{\mathbf{s}}$ (state changing) and from $T(D)_{\mathbf{s}}$ to $T(D)_s$, where $s \neq \mathbf{s}$ (for value outputs). Recall that all function names in a linear signature contain at most one instance of the type of interest in their domain sorts. We shall ignore the ones that do not contain the type of interest as a domain sort.[1] By convention, all other functions have the following prototype:

$$f : \mathbf{s}, s_1, \ldots, s_n \rightarrow s$$

Let $t_1 \in T(\Pi)_{s_1}, \ldots, t_n \in T(\Pi)_{s_n}$ be terms. We can define the quotient term algebra function

[1]It can be argued that, in a well-defined algebraic specification, there should be no function whose domain does not include the type of interest. Such a function belongs in a different specification, from which this one can inherit.

$$f^{T(D)}_{[t_1],\ldots,[t_n]} : T(D)_{\mathbf{s}} \to T(D)_{\mathbf{s}}$$

as

$$f^{T(D)}_{[t_1],\ldots,[t_n]}([t]) = f^{T(D)}([t],[t_1],\ldots,[t_n])$$

for all ground terms $t \in T(\Pi)_{\mathbf{s}}$. Moreover, by definition of $T(D)$

$$f^{T(D)}([t],[t_1],\ldots,[t_n]) = [f(t,t_1,\ldots,t_n)]$$

Thus, every function $f : \mathbf{s}, s_1, \ldots, s_n \to s$ and every tuple $(t_1, \ldots, t_n)$ of terms of the appropriate sorts define a function

$$f_{t_1,\ldots,t_n} : T(D)_{\mathbf{s}} \to T(D)_{\mathbf{s}}$$

such that, for all ground terms $t \in T(\Pi)_{\mathbf{s}}$

$$f_{t_1,\ldots,t_n}([t]) = [f(t,t_1,\ldots,t_n)]$$

We denote the alphabet letter corresponding to the function $f_{t_1,\ldots,t_n}$ by the tuple $(f, t_1, \ldots, t_n)$.

Example 6.2.2. Suppose we are given a specification D for a table of characters. The type of interest is **table**, the stored elements are of sort *char* and indices are given by the sort *nat*. One of the function names in the specification's signature is

$$put : \mathbf{table}, char, nat \to \mathbf{table}$$

If the terms of sort *char* include $a, b, c, \ldots$ and the terms of sort *nat* are $0, succ(0), \ldots$, then alphabet letters in the constructed Mealy machine include

$$\begin{array}{c} (put, a, 0) \\ (put, b, 0) \\ (put, c, 0) \\ \vdots \\ (put, a, succ(0)) \\ (put, b, succ(0)) \\ (put, c, succ(0)) \\ \vdots \end{array}$$

■

The formal definition of the constructed Mealy automaton is given next. Note that we have divided the alphabet into two sets, Σ_S for letters resulting in a state change, and Σ_V for letters resulting in output values (see Section 4.6 for a discussion on the reasons for this distinction).

Definition 6.2.2. Let $D = (\Pi, X, E)$ be a linear specification. The *quotient term automaton* is a Mealy machine $\mathscr{M} = (Q, \Sigma, \delta, q_0, \Theta, \theta)$ where

- $Q = T(D)_{\mathbf{s}}$
- $\Sigma = \Sigma_S \cup \Sigma_V$ with
  - $\Sigma_S = \{(f, t_1, \ldots, t_n) | f : \mathbf{s}, s_1, \ldots, s_n \to \mathbf{s} \in F; t_i \in T(\Pi)_{s_i}, i = 1, \ldots, n\}$
  - $\Sigma_V = \{(g, t'_1, \ldots, t'_m) | g : \mathbf{s}, s'_1, \ldots, s'_m \to s' \in F; t'_i \in T(\Pi)_{s'_i}, i = 1, \ldots, m; s' \neq \mathbf{s}\}$
- $q_0 = [f]_D$, where $f :\to \mathbf{s} \in F$
- $\delta([t]_D, (f, t_1, \ldots, t_n)) = \begin{cases} [f(t, t_1, \ldots, t_n)]_D & (f, t_1, \ldots, t_n) \in \Sigma_S \\ [t]_D & (f, t_1, \ldots, t_n) \in \Sigma_V \end{cases}$
- $\Theta = \bigcup_{s \neq \mathbf{s}} T(D)_s$
- $\theta([t]_D, (f, t_1, \ldots, t_n)) = \begin{cases} \text{undefined} & (f, t_1, \ldots, t_n) \in \Sigma_S \\ [f(t, t_1, \ldots, t_n)]_D & (f, t_1, \ldots, t_n) \in \Sigma_V \end{cases}$

Example 6.2.3. Consider the signature Π in Example 6.2.1. We create the specification $D = (\Pi, X, E)$, where X has a single element x of sort *nat*, and E consists of the following axioms:

$$\begin{aligned} &\mathit{positive}(0) = \mathit{false} \\ &\mathit{positive}(\mathit{succ}(x)) = \mathit{true} \end{aligned}$$

A diagram for the resulting quotient term automaton is given in Figure 6.1. ■

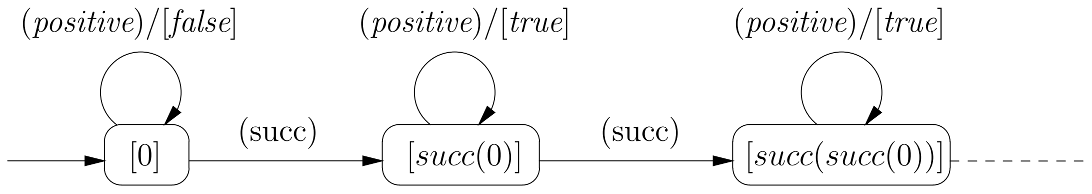


Figure 6.1: The quotient term automaton for the $nat - bool$ abstract data type given by the specification in example 6.2.3.

While the construction of an automaton from a linear specification results in a valid Mealy machine, it is not clear whether this machine truthfully represents the same abstract data structure as the specification. One option to justify this construction would be to apply the reverse construction of an algebra from an automaton, as presented in Chapter 4. However, a proof that applies only constructions may be justly dismissed. Instead, we attempt to justify the construction suggested in this section by proving that the specification congruence $\sim^D$, which lies at the core of the quotient term algebra, is closely related to the relation $\equiv_\delta$ of the generated automaton. It was argued in Chapter 2 that $\equiv_\delta$ encapsulates all the information of the state machine from which it was defined.

To obtain a result connecting these two relations, we use a method similar to the one used in Chapter 4. We define a function that converts terms into words over the alphabet of the constructed automaton, and then show that two terms are related under $\sim^D$ if and only if their corresponding words are equivalent under $\equiv_\delta$. The proofs themselves are straight-forward, and mostly involve a translation of terms into strings. Such a translation is only feasible if the signature is linear.

Definition 6.2.3. Let $\mathscr{M} = (Q, \Sigma, \delta, q_0, \Theta, \theta)$ be a quotient term automaton for the linear specification $D = (\Pi, X, E)$. The function $\varphi^* : T(\Pi)_{\mathbf{s}} \to \Sigma^*$ is defined inductively:

- $\varphi^*(f) = \epsilon$ for $f \to \mathbf{s} \in F$
- $\varphi^*(f(t, t_1 \ldots, t_n)) = \varphi^*(t)(f, t_1, \ldots, t_n)$ for all $f : \mathbf{s}, s_1, \ldots, s_n \to \mathbf{s} \in F$, $t \in T(\Pi)_{\mathbf{s}}$ and $t_i \in T(\Pi)_{s_i}$ for $i = 1, \ldots, n$.

Lemma 6.2.1. *For all ground terms $t \in T(\Pi)$ of sort $\mathbf{s}$, $\delta(q_0, \varphi^*(t)) = [t]_D$.*

Proof. By structural induction on t. Since D is linear, there is only one constant f of sort $\mathbf{s}$. Furthermore, by definition, $[f]_D = q_0$. Therefore

$$\delta(q_0, \varphi^*(f)) = \delta(q_0, \epsilon) = q_0 = [f]_D$$

Assume the lemma holds for some term t of sort $\mathbf{s}$. Let $f : \mathbf{s}, s_1, \ldots, s_n \to \mathbf{s}$ be a function name and let $t_1 \in T(\Pi)_{s_1}, \ldots, t_n \in T(\Pi)_{s_n}$ be terms. Then

$$\begin{aligned}\delta(q_0, \varphi^*(f(t, t_1, \ldots, t_n))) &= \delta(q_0, \varphi^*(t)(f, t_1, \ldots, t_n)) \\ &= \delta(\delta(q_0, \varphi^*(t)), (f, t_1, \ldots, t_n)) \\ &= \delta([t]_D, (f, t_1, \ldots, t_n)) \\ &= [f(t, t_1, \ldots, t_n)]_D\end{aligned}$$

as required. □

Theorem 6.2.1. *For all terms $t_1, t_2 \in T(\Pi)$ of sort $\mathbf{s}$, $t_1 \sim^D t_2$ if and only if $\varphi^*(t_1) \equiv_\delta \varphi^*(t_2)$.*

Proof. By definition, $\varphi^*(t_1) \equiv_\delta \varphi^*(t_2)$ if and only if $\delta(q_0, \varphi^*(t_1)) = \delta(q_0, \varphi^*(t_2))$. By Lemma 6.2.1, this equation holds if and only if $[t_1]_D = [t_2]_D$, or, in other words, if and only if $t_1 \sim^D t_2$. □

6.3 Handling Errors

Algebraic specifications, in the form presented in this thesis, do not have an integral mechanism for specifying error conditions. This has led to several misconceptions in the literature regarding error handling. As we shall see, the initial algebra approach allows us considerable freedom of choice as to how error conditions should be specified, while maintaining a correct specification of the abstract data type.

Consider, for example, a stack abstract data type. The stack contains items that can be inserted and removed, following a Last-In-First-Out policy. We shall consider three different models for handling errors in a stack, and present the resulting algebraic specifications and automata for each model.

An error in the stack ADT occurs when the user tries to remove an item from an empty stack, or ask for the top item of such a stack. Our first model ignores errors altogether, assuming the user is careful enough not to perform an illegal operation (for example, the user may test whether the stack is empty before attempting to pop an item). Such a behaviour is analogous to dereferencing an uninitialised pointer in C/C++: the result of this illegal operation is undefined, and it is the responsibility of the user to avoid such an error.

Example 6.3.1. A stack ADT is specified by the signature $\Pi = (S, F)$ and the specification $D = (\Pi, X, E)$, defined as follows:

$$\begin{aligned}
S &= \{\textbf{stack}, \mathit{item}\} \\
F &= \{\mathit{new} :\to \textbf{stack} \\
&\qquad 0, 1 :\to \mathit{item} \\
&\qquad \mathit{push} : \textbf{stack}, \mathit{item} \to \textbf{stack} \\
&\qquad \mathit{pop} : \textbf{stack} \to \textbf{stack} \\
&\qquad \mathit{top} : \textbf{stack} \to \mathit{item} \\
X &= \{s : \textbf{stack}, i : \mathit{item}\} \\
E &= \{\mathit{pop}(\mathit{push}(s, i)) = s, \\
&\qquad \mathit{top}(\mathit{new}) = 0, \\
&\qquad \mathit{top}(\mathit{push}(s, i)) = i\}
\end{aligned}$$

The axioms in this specification show two separate options for handling errors within the first approach. For the *top*(*new*), we have set the result to some arbitrary value. Such a value cannot be distinguished from the result of a legal operation. For the second option, consider the term *pop*(*new*), which represents an illegal sequence. Note that there is no axiom in the specification that allows us to infer an equivalence between *pop*(*new*) and any other term. In this case we have used the important property of initial algebras, which states that two terms are considered distinct unless proved to be equal. Therefore, *pop*(*new*) produces an equivalence class under $\sim^D$ for which it is the only member. The same is true for any term that includes *pop*(*new*) as a sub-term. ■

The second approach, referred to as *error recovery* [9], distinguishes a special term as the result of an illegal operation. The user can then immediately observe when an error has occurred by checking the returned value. However, the user is not restricted from using this value as a parameter to other operations. That is, unless the user checks the returned item, a trace of operations that includes an illegal one is fully executed, with possibly undesired results.

Example 6.3.2. The following specification demonstrates the use of a dedicated error value for the result of an illegal *top* operation:

$$
\begin{aligned}
S &= \{\textbf{stack}, \mathit{item}\}\\
F &= \{\mathit{new} :\rightarrow \textbf{stack}\\
&\qquad 0, 1, \mathit{undef} :\rightarrow \mathit{item}\\
&\qquad \mathit{push} : \textbf{stack}, \mathit{item} \rightarrow \textbf{stack}\\
&\qquad \mathit{pop} : \textbf{stack} \rightarrow \textbf{stack}\\
&\qquad \mathit{top} : \textbf{stack} \rightarrow \mathit{item}\\
X &= \{s : \textbf{stack}, i : \mathit{item}\}\\
E &= \{\mathit{pop}(\mathit{new}) = \mathit{new},\\
&\qquad \mathit{pop}(\mathit{push}(s, i)) = s,\\
&\qquad \mathit{top}(\mathit{new}) = \mathit{undef},\\
&\qquad \mathit{top}(\mathit{push}(s, i)) = i\}
\end{aligned}
$$

In this example, the term *push*(*new*, *undef*) is syntactically legal, even though this term clearly does not represent a valid sequence of operations on a stack. ■

The last model uses a method called *error propagation* [14]. This approach resembles exceptions in languages like C++ and Java, in that an illegal operation causes the ADT to enter a trap state in which it remains until the user explicitly performs a recovery operation (usually creating a new instance of the ADT). In algebraic specifications, a trap can be modelled by a term t that is equivalent to all terms that contain t as a sub-term. Formally:

Definition 6.3.1. Let $\Pi = (S, F)$ be a signature and let $D = (\Pi, X, E)$ be a specification. A term t of sort s is called a *trap term* if for all function names $f : s_1, \ldots, s, \ldots, s_n \rightarrow s$ and for all terms $t_1 \in T(\Pi)_{s_1}, \ldots, t_n \in T(\Pi)_{s_n}$,

$$f(t_1, \ldots, t, \ldots, t_n) \sim^D t$$

Example 6.3.3. The specification below defines *pop*(*new*) as a trap term:

$$
\begin{aligned}
S &= \{\mathbf{stack}, \mathit{item}\}\\
F &= \{\mathit{new} :\rightarrow \mathbf{stack}\\
&\qquad 0, 1, \mathit{undef} :\rightarrow \mathit{item}\\
&\qquad \mathit{push} : \mathbf{stack}, \mathit{item} \rightarrow \mathbf{stack}\\
&\qquad \mathit{pop} : \mathbf{stack} \rightarrow \mathbf{stack}\\
&\qquad \mathit{top} : \mathbf{stack} \rightarrow \mathit{item}\\
X &= \{s : \mathbf{stack}, i : \mathit{item}\}\\
E &= \{\mathit{pop}(\mathit{pop}(\mathit{new})) = \mathit{pop}(\mathit{new}),\\
&\qquad \mathit{push}(\mathit{pop}(\mathit{new}), i) = \mathit{pop}(\mathit{new}),\\
&\qquad \mathit{push}(s, \mathit{undef}) = \mathit{pop}(\mathit{new}),\\
&\qquad \mathit{pop}(\mathit{push}(s, i)) = s,\\
&\qquad \mathit{top}(\mathit{new}) = \mathit{undef},\\
&\qquad \mathit{top}(\mathit{push}(s, i)) = i\}
\end{aligned}
$$

While the trap term could have been defined as a new constant of sort *stack*, we have used here the term *pop*(*new*) in order to keep the specification linear (see Definition 6.2.1).

Note that we have added an axiom which states that pushing the undefined value is an error, by equating any such term to the trap term.

There is, however, a semantic flaw in the specification in Example 6.3.3: the choice of axioms collapses all values of sort *item* to one equivalence class. Consider the following equalities:

$$
\begin{aligned}
[0]_D &= [\mathit{top}(\mathit{push}(\mathit{pop}(\mathit{new}), 0))]_D\\
&= \mathit{top}^{T(D)}([\mathit{push}(\mathit{pop}(\mathit{new}), 0)]_D)\\
&= \mathit{top}^{T(D)}([\mathit{pop}(\mathit{new})]_D)\\
&= [\mathit{top}(\mathit{pop}(\mathit{new}))]_D\\
&= [\mathit{undef}]_D
\end{aligned}
$$

Similarly, it can be shown that $[1]_D = [\mathit{undef}]_D$. This example proves an important point about algebraic specifications: one cannot write a “wrong” specification, in the sense that any specification that is syntactically correct leads to a valid abstract data type. The only question, in this case, is how does the specified ADT compares with the one intended by the specification's author.

To correct the problem with the specification in Example 6.3.3, we can use a function that determines whether a state is the trap state, and then use it to break $\mathit{top}(\mathit{push}(s, i))$ into two cases. Such a scheme can be implemented by adding the function prototype

$$\mathit{select} : \mathbf{stack}, \mathit{item}, \mathit{item} \rightarrow \mathit{item}$$

to the stack signature, and the axioms

$$\begin{aligned}
&\mathit{select}(\mathit{pop}(\mathit{new}), i_1, i_2) = i_1 \\
&\mathit{select}(\mathit{new}, i_1, i_2) = i_2 \\
&\mathit{select}(\mathit{push}(s, i), i_1, i_2) = \mathit{select}(s, i_1, i_2)
\end{aligned}$$

to the specification. What *select* does is to check whether a given term of sort **stack** is the trap term. If so, it selects the first item (i_1), and if not it selects the second item (i_2). We can now write a new axiom for $\mathit{top}(\mathit{push}(s, i))$ that reads

$$\mathit{top}(\mathit{push}(s, i)) = \mathit{select}(s, \mathit{undef}, i)$$

The resulting quotient term automaton is depicted in Figure 6.2. The trap term $\mathit{pop}(\mathit{new})$ defines a trap state in the automaton, that is, a state which has only incoming edges and self loops. This state is reached either when the letter (*pop*) is applied to the state new, or the letter (*push*, *undef*) is applied to any state.

6.4 Non-Linear Data Structures

We conclude this chapter with a discussion of non-linear data structures. Recall from Definition 6.2.1 that a specification is linear if it adheres to three conditions. Consequently, a specification is non-linear if it violates at least one of these conditions. We shall investigate what happens to the quotient term automaton in any of these cases.

First, assume that a signature contains more than one constant of the type of interest. The result is a machine with more than one initial state, which is therefore

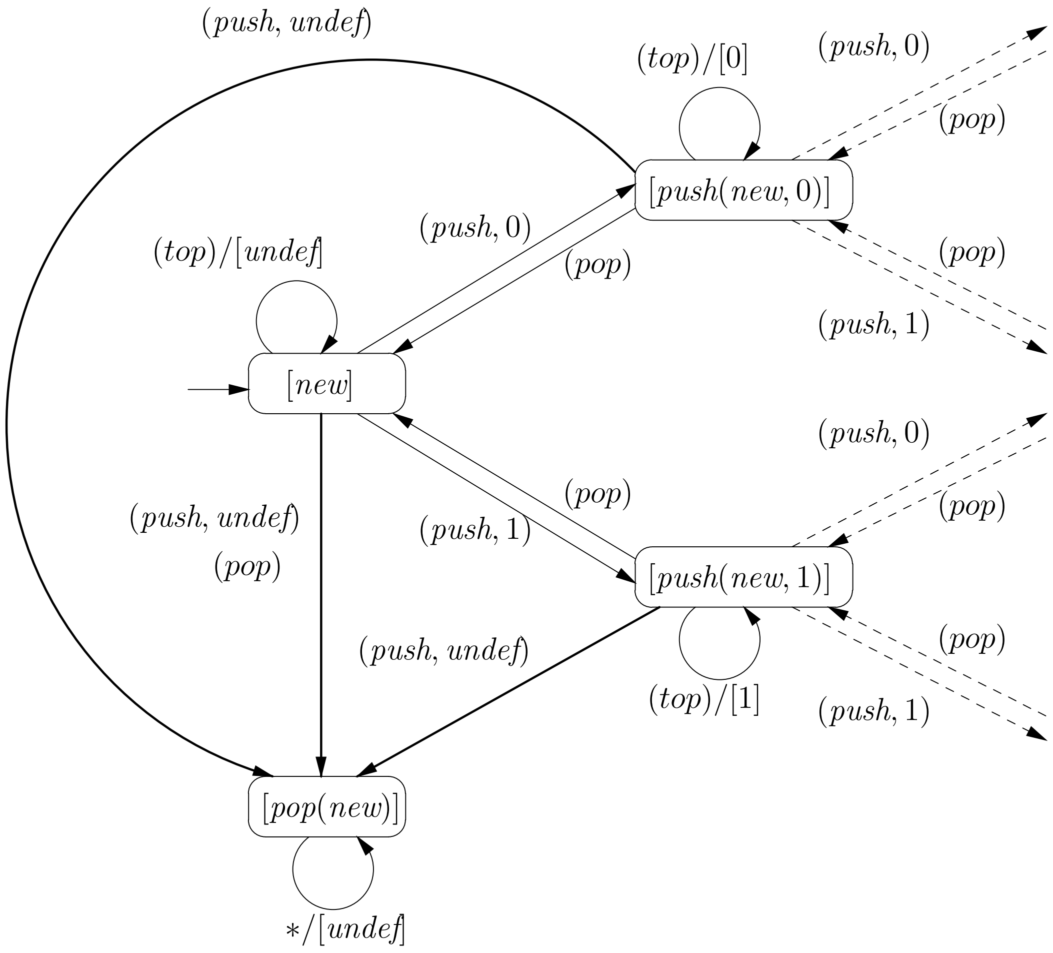


Figure 6.2: The stack quotient term automaton with error propagation. The term *pop*(*new*) is a trap term and hence the automaton state [*pop*(*new*)] is a trap state. The bold edges represent transitions to the trap state. Dashed edges indicate that there are more states that are not shown in this diagram. See Example 6.3.3 for the stack's algebraic specification.

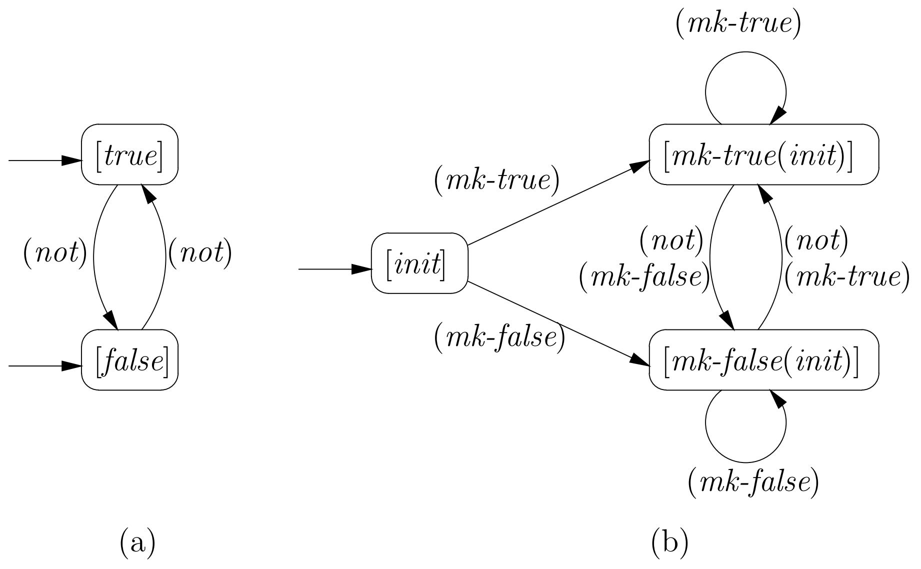


Figure 6.3: A quotient term automaton for a Boolean ADT specification with multiple initial states (a) and a possible deterministic automaton for the same data type (b).

a non-deterministic machine. Consider, for example, the signature $\Pi = (S, F)$ and specification $D = (\Pi, \varnothing, E)$ for a Boolean abstract data type:

$$\begin{aligned} S &= \{\textbf{bool}\} \\ F &= \{\mathit{true} :\rightarrow \textbf{bool}, \\ &\quad\ \mathit{false} :\rightarrow \textbf{bool}, \\ &\quad\ \mathit{not} : \textbf{bool} \rightarrow \textbf{bool}\} \\ E &= \{\mathit{not}(\mathit{true}) = \mathit{false}, \\ &\quad\ \mathit{not}(\mathit{false}) = \mathit{true}\} \end{aligned}$$

The non-deterministic quotient term automaton for this specification is given in Figure 6.3(a).

This non-determinism, however, is quite superficial, as the transition function remains deterministic. In fact, we can make the automaton deterministic quite easily by using one of these methods:

- Modify the automaton by adding a state and an alphabet letter for each initial state. The new state becomes the unique initial state, from which the new alphabet letters lead to the old initial states. The transition function becomes a partial function in which the new alphabet letters are ignored for all but the new initial state.

- Modify the specification by changing every constant $c :\to \mathbf{s}$ of the type of interest into a unary function (e.g., *make-c* : $\mathbf{s} \to \mathbf{s}$), and adding a single constant (e.g. *init*). The old constants are now represented by the application of their corresponding unary functions to the new constant term.

The main problem with the first approach is that the abstract data type specified by the automaton is no longer the same as the one described by the algebraic specification. However, the new automaton $\mathscr{M}'$ closely resembles the original automaton $\mathscr{M}$, in the sense that for any initial state in $\mathscr{M}$ and any word in the alphabet of $\mathscr{M}$, both transition functions take their respective machines to the same state.

In the second approach, the machine is the quotient term automaton, which means that it follows the specification exactly. On the other hand, the modified specification results in a new abstract data type. In this type, we first create an uninitialised instance of the ADT, and then use the new unary functions to give it some initial value. The new specification needs to describe the semantics of applying the unary functions to terms other than the new constant, as well as of applying any other function on the uninitialised instance. The first can be done by treating the unary functions as reset operations, and the second by using any of the error handling methods described in Section 6.3.

While a non-linear specification violating the first condition results in mostly technical difficulties, a violation of the second condition raises more profound problems. Recall that the second condition states that a function name whose range sort is the type of interest must have exactly one instance of this sort in its domain tuple. Following this assumption, we were able to treat terms as functions taking one term of the type of interest as an argument and returning another as the result. We shall now consider specifications containing a function name with more than one instance of the type of interest in its domain tuple. Such functions will be referred to as *non-linear*.

From a purely mathematical point of view, there seems to be nothing wrong with the construction introduced in Section 6.2 when applied to specifications containing non-linear function names. In some cases, the result is even reasonable, as in the following specification of a Boolean ADT with a logical-and function:

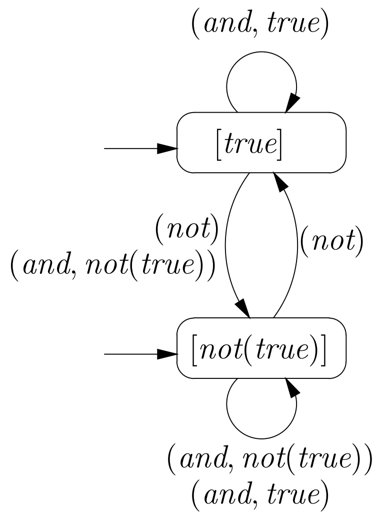


Figure 6.4: A quotient term automaton for a Boolean ADT specification containing a non-linear function name.

$$\begin{aligned}
S &= \{\mathbf{bool}\}\\
F &= \{\mathit{true} :\to \mathbf{bool},\\
&\qquad \mathit{not} : \mathbf{bool} \to \mathbf{bool},\\
&\qquad \mathit{and} : \mathbf{bool}, \mathbf{bool} \to \mathbf{bool}\}\\
X &= \{x, y : \mathbf{bool}\}\\
E &= \{\mathit{not}(\mathit{not}(x)) = x,\\
&\qquad \mathit{and}(\mathit{true}, \mathit{true}) = \mathit{true},\\
&\qquad \mathit{and}(\mathit{not}(\mathit{true}), x) = \mathit{not}(\mathit{true}),\\
&\qquad \mathit{and}(x, y) = \mathit{and}(y, x)\}
\end{aligned}$$

The quotient term automaton for this specification is depicted in Figure 6.4.

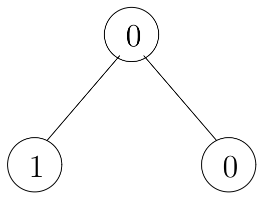
On the other hand, the construction of the quotient term automaton leads to strange results for some abstract data types defined with non-linear specifications. To illustrate this point, we shall use the following specification of a binary tree:

$$
\begin{aligned}
S = \{&\mathbf{tree}, \mathit{item}\} \\
F = \{&\mathit{empty} :\to \mathbf{tree}, \\
&0, 1 : \mathit{item}, \\
&\mathit{make} : \mathbf{tree}, \mathit{item}, \mathbf{tree} \to \mathbf{tree} \\
&\mathit{left} : \mathbf{tree} \to \mathbf{tree} \\
&\mathit{right} : \mathbf{tree} \to \mathbf{tree} \\
&\mathit{root} : \mathbf{tree} \to \mathit{item}\} \\
X = \{&x, y : \mathbf{tree}, r : \mathbf{item}\} \\
E = \{&\mathit{left}(\mathit{empty}) = \mathit{empty}, \\
&\mathit{left}(\mathit{make}(x, r, y)) = x, \\
&\mathit{right}(\mathit{empty}) = \mathit{empty}, \\
&\mathit{right}(\mathit{make}(x, r, y)) = y, \\
&\mathit{root}(\mathit{make}(x, r, y)) = r\}
\end{aligned}
$$

The binary tree specification contains the non-linear function name *make*, which takes two sub-trees and a root item, and creates a new tree. Mathematically, the construction of the quotient term automaton results in a valid machine. Moreover, there is nothing in the proof of Theorem 6.2.1 which prevents this result from being applicable to the tree specification.

Nevertheless, we are not interested in algebraic specifications and automata as abstract mathematical structures. The ultimate goal of the construction of the quotient term automaton is to come up with a useable specification of a software module. In this respect, the resulting machine for the binary tree fails in this test, as it can hardly be considered as a reasonable specification. This claim is supported by two important observations.

First, the automaton fails to capture the process of a tree's construction. Consider, for example, the binary tree



which is specified by the term

$$\mathit{make}(\mathit{make}(\mathit{empty}, 1, \mathit{empty}), 0, \mathit{make}(\mathit{empty}, 0, \mathit{empty}))$$

If we use the construction described in Section 6.2, we get that the state corresponding to this tree is the result of applying the string

$$(\mathit{make}, 1, \mathit{empty})(\mathit{make}, 0, \mathit{make}(\mathit{empty}, 0, \mathit{empty}))$$

on the initial state $[\mathit{empty}]$. It can be seen how describing the construction of a tree as a sequence of operations misrepresents the actual process, where each sub-tree is constructed independently and then the two are merged into a new object. Also, the sequence completely ignores the symmetric nature of the construction, by assuming that the tree is created from its leftmost node, with a new tree obtained by adding a root and a right sub-tree to the current object.

The second obstacle that renders this specification impractical is the connection between the alphabet and the state set in the automaton. By construction, if $f : \mathbf{s}, s_1, \ldots, s_n \rightarrow \mathbf{s}$ is a function name, then the alphabet contains a symbol for every combination of terms $t_1 \in T(\Pi)_{s_1}, \ldots, t_n \in T(\Pi)_{s_n}$. We have already seen that we need to consider only one representative of each equivalence class under $\sim^D$. Therefore, the set of alphabet symbols generated by the function name f is in one-to-one correspondence with the set

$$T(\Pi)_{s_1}/\sim^D \times \ldots \times T(\Pi)_{s_n}/\sim^D$$

In the case of the binary tree, the alphabet symbols generated by the *make* function are in one-to-one correspondence with the set

$$T(\Pi)_{\mathit{item}}/\sim^D \times T(\Pi)_{\mathbf{tree}}/\sim^D$$

However, it can be inferred from the axioms that every application of *make* results in a new class in $T(\Pi)_{\mathbf{tree}}/\sim^D$. A representative of this new class then generates more alphabet symbols, which, when applied by the transition function, create more states that need to be specified, and so on.

Consider once again the binary tree. The initial state is $[\mathit{empty}]$. When the transition function applies the alphabet symbol $(\mathit{make}, 0, \mathit{empty})$ to this state we get the new state $[\mathit{make}(\mathit{empty}, 0, \mathit{empty})]$. This state represents a class of terms of sort **tree**, which is not equal to $[\mathit{empty}]$. We therefore take a representative of this class and create the alphabet symbols

$$(make, 0, make(empty, 0, empty))$$

and

$$(make, 0, make(empty, 0, empty))$$

Applying these new symbols to current states results in new states and new alphabet symbols, all of which need to become a part of the specification.

It appears that this feedback process of generating new states from alphabet symbols and new symbols from states makes it impossible to provide a finite definition of the transition function. It is unlikely that such a definition exists for the binary tree, and, consequently, for any other specification of a tree. Moreover, we can deduce the same conclusion for many other ADTs whose algebraic specification is clear and concise. The reason is that every algebra of terms can be looked upon as a set of trees with constants and variables as leaves and with non-constant function names as internal nodes [14].

Unfortunately, we have been unable to characterise the set of ADTs for which a practical automaton specification can be obtained from the algebraic specifications. We know that the procedure works for linear ADTs as well as for some non-linear ones (for example, the specification of a queue in [9] §2.14 contains the non-linear selection function IF-THEN-ELSE which can be easily modelled in the transition function). A better description of the set of ADTs that cannot be specified by automata is left for future research.

There are two more options for non-linear specifications we have not yet considered, and will now describe briefly. A function name with the type of interest as the range sort and without any occurrence of the type of interest in the domain creates new instances of the object without any prior state. Therefore, we can consider such functions as defining a set of constants. For example, if $f : s, s \rightarrow \mathbf{s}$ is a function name, such that $s \neq \mathbf{s}$, and if t_1, t_2 are terms of sort s, then $f(t_1, t_1)$, $f(t_1, t_2)$, $f(t_2, t_1)$ and $f(t_2, t_2)$ should all be treated as constants of sort $\mathbf{s}$.

A function name whose range sort is not the type of interest and whose domain contains more than one instance of the type of interest may not lead to problems, since it does not suffer from the same feedback property described above. Whether such function can be described efficiently as part of the automaton's output function seems to depend on the structure of the function names that affect the state set.

Bibliography

[1] W. Bartussek and D. L. Parnas. Using assertions about traces to write abstract specifications for software modules. In *Proceedings of the 2nd Conference of the European Cooperation on Informatics*, pages 211–236. Springer-Verlag, 1978.

[2] G. Birkhoff. On the structure of abstract algebras. *Proceedings of the Cambridge Philosophical Society*, 31:433–454, 1935.

[3] J. P. Bowen and M. G. Hinchey, editors. *Applications of Formal Methods.* Prentice Hall International Series in Computer Science. Prentice Hall, 1995.

[4] J. P. Bowen and M. G. Hinchey. Ten commandments revisited: a ten-year perspective on the industrial application of formal methods. In *FMICS '05: Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems*, pages 8–16. ACM Press, 2005.

[5] J. Brzozowski and H. Jürgensen. Representation of semiautomata canonical words and equivalences. *International Journal of Foundations of Computer Science*, 16(5):831–850, 2005.

[6] J. Brzozowski and H. Jürgensen. Theory of deterministic trace-assertion specifications. Technical Report CS-2004-30, School of Computer Science, University of Waterloo, 2005.

[7] R. M. Burstall and J. A. Goguen. The semantics of CLEAR, a specification language. In *Proceedings of the Abstract Software Specifications, 1979 Copenhagen Winter School*, pages 292–332. Springer-Verlag, 1980.

[8] E. W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[9] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I - Equations and Initial Semantics.* Springer-Verlag Berlin, 1985.

[10] S. Eilenberg. *Automata, Languages and Machines*, volume B. Academic Press, 1976.

[11] S. J. Garland and J. V. Guttag. A guide to LP, the Larch prover. Technical Report 82, Digital Equipment Corporation Systems Research Center, 1991.

[12] A. Ginzburg. *Algebraic Theory of Automata.* Academic Press, 1968.

[13] J. A. Goguen and J. Meseguer. Completeness of many-sorted equational logic. *SIGPLAN Notices*, 16(7):24–32, 1981.

[14] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. T. Yeh, editor, *Current Trends in Programming Methodology*, volume IV: Data Structuring, pages 80–149. Prentice Hall, 1978.

[15] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24(1):68–95, 1977.

[16] J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J. P. Jouannaud. Introducing OBJ. In J. A. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action.* Kluwer, 2000.

[17] R. M. Gott, J. R. Baumgartner, P. Roessler, and S. I. Joe. Functional formal verification on designs of pseries microprocessors and communication subsystems. *IBM Journal of Research and Development*, 49(4/5):565–580, 2005.

[18] J. V. Guttag. *The specification and application to programming of abstract data types.* PhD thesis, University of Toronto, 1975.

[19] J. V. Guttag. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):396–404, 1977.

[20] J. V. Guttag and J. J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[21] J. V. Guttag, E. Horowitz, and D. R. Musser. The design of data type specifications. In *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering*, pages 414–420. IEEE Computer Society Press, 1976.

[22] J. V. Guttag, E. Horowitz, and D. R. Musser. The design of data type specifications. In R. T. Yeh, editor, *Current Trends in Programming Methodology*, volume IV: Data Structuring, pages 60–79. Prentice Hall, 1978.

[23] J. Henkel and A. Diwan. Discovering algebraic specifications from java classes. In *17th European Conference on Object-Oriented Programming (ECOOP 2003)*. Springer, 2003.

[24] W. M. L. Holcombe. *Algebraic Automata Theory*. Cambridge University Press, 1982.

[25] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[26] C. B. Jones. *Systematic software development using VDM, 2nd Edition*. Prentice Hall, 1990.

[27] B. Liskov and S. Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 50–59, 1974.

[28] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34:1045–1079, 1955.

[29] B. Meyer. On formalism in specifications. *IEEE Software*, 2(1):6–26, 1985.

[30] D. L. Parnas and Y. Wang. The trace assertion method of module interface specification. Technical Report 89-261, Queen's University, 1989.

[31] B. C. Pierce. *Basic category theory for computer scientists*. MIT Press, 1991.

[32] N. Narasimhan R. Kaivola and. Formal verification of the Pentium® 4 floating-point multiplier. In *Design, Automation and Test in Europe Conference and Exhibition (DATE'02)*, pages 20–27, 2002.

[33] J. M. Spivey. *The Z Notation: a Reference Manual*. Prentice Hall, 1992.

[34] J. W. Thatcher, E. G. Wagner, and J. B. Wright. Data type specification: Parameterization and the power of specification techniques. *ACM Transactions on Programming Languages and Systems*, 4(4):711–732, 1982.

[35] M. Wand. Final algebra semantics and data type extensions. *Journal of Computer and System Sciences*, 19(1):27–44, 1979.

[36] J. M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–23, 1990.

[37] M. Wirsing. Algebraic specifications. In *Handbook of Theoretical Computer Science (vol. B): Formal Models and Semantics*, pages 675–788. MIT Press, 1990.