

# 1

## Abstract data types



The history of programming languages may be characterized as the genesis of increasingly powerful abstractions to aid the development of reliable programs.

### Abstract data types

8

1-1

- abstraction and types
- algebraic specification
- modules versus classes
- types as constraints

Additional keywords and phrases: *control abstractions, data abstractions, compiler support, description systems, behavioral specification, implementation specification*

Slide 1-1: Abstract data types

In this chapter we will look at the notion of *abstract data types*, which may be regarded as an essential constituent of object-oriented modeling. In particular, we will study the notion of data abstraction from a foundational perspective, that is based on a mathematical description of types. We start this chapter by discussing the notion of *types as constraints*. Then, we look at the (first order) algebraic specification of abstract data types, and we explore the trade-offs between the traditional implementation of abstract data types by employing

modules and the object-oriented approach employing classes. We conclude this chapter by exploring the distinction between classes and types, as a preparation for the treatment of (higher order) polymorphic type theories for object types and inheritance in the next chapter.

## 1.1 Abstraction and types

The concern for abstraction may be regarded as the driving force behind the development of programming languages (of which there are astoundingly many). In the following we will discuss the role of abstraction in programming, and especially the importance of types. We then briefly look at what mathematical means we have available to describe types from a foundational perspective and what we may (and may not) expect from types in object-oriented programming.

### 1.1.1 Abstraction in programming languages

In [Shaw84], an overview is given of how increasingly powerful abstraction mechanisms have shaped the programming languages we use today. See slide 1-2.

#### Abstraction – *programming methodology*

- control abstractions – *structured programming*
- data abstraction – *information hiding*

The kind of abstraction provided by ADTs can be supported by any language with a procedure call mechanism (given that appropriate *protocols* are developed and observed by the programmer). [DT88]

1-2

Slide 1-2: Abstraction and programming languages

Roughly, we may distinguish between two categories of abstractions: abstractions that aid in specifying *control* (including subroutines, procedures, *if-then-else* constructs, *while*-constructs, in short the constructs promoted by the school of *structured programming* in their battle against the *goto*); and abstractions that allow us to hide the actual representation of the data employed in a program (introduced to support the *information hiding* approach, originally advocated in [Parnas72a]).

Although there is clearly a pragmatic interest involved in developing and employing such abstractions, the concern with abstraction (and consequently types) is ultimately motivated by a concern with programming methodology and, as observed in [DT88], the need for reliable and maintainable software. However, the introduction of language features is also often motivated by programmers' desires for ease of coding and naturalness of expression.

In the same vein, although types were originally considered as a convenient means to assist the compiler in producing efficient code, types have rapidly been recognized as a way in which to capture the meaning of a program in an

implementation independent way. In particular, the notion of abstract data types (which has, so to speak, grown out of data abstraction) has become a powerful device (and guideline) to structure large software systems.

In practice, as the quotation from [DT88] in slide 1-2 indicates, we may employ the tools developed for structured programming to realize abstract data types in a program, but with the obvious disadvantage that we must rely on conventions with regard to the reliability of these realizations. Support for abstract data types (support in the sense as discussed in section ??) is offered (to some extent) by languages such as Modula-2 and Ada by means of a syntactic module or package construct, and (to a larger extent) by object-oriented languages in the form of object classes. However, both realizations are of a rather *ad hoc* and pragmatic nature, relying in the latter case on the metaphor of encapsulation and message passing. The challenge to computer science in this area is to develop a notion of types capturing the power of abstract data types in a form that is adequate both from a pragmatic point of view (in the sense of allowing efficient language support) and from a theoretical perspective (laying the foundation for a truly declarative object-oriented approach to programming).

### 1.1.2 A foundational perspective – types as constraints

Object-oriented programming may be regarded as a *declarative* method of programming, in the sense that it provides a computation model (expressed by the metaphor of encapsulation and message passing) that is independent of a particular implementation model. In particular, the inheritance subtype relation may be regarded as a pure description of the relations between the entities represented by the classes. Moreover, an object-oriented approach favors the development of an object model that bears close resemblance to the entities and their relations living in the application domain. However, the object-oriented programming model is rarely introduced with the mathematical precision characteristic of descriptions of the other declarative styles, for example the functional and logic programming model. Criticizing, [DT88] remark that *OOP is generally expressed in philosophical terms, resulting in a proliferation of opinions concerning what OOP really is.*

From a type-theoretical perspective, our interest is to identify abstract data types as elements of some *semantic* (read mathematical) domain and to characterize their properties in an unambiguous fashion. See slide 1-3.

There seems to be almost no limit to the variety and sophistication of the mathematical models proposed to characterize abstract data types and inheritance. We may make a distinction between first order approaches (based on ordinary set theory) and higher order approaches (involving typed lambda calculus and constructive logic).

The algebraic approach is a quite well-established method for the formal specification of abstract data types. A type (or sort) in an algebra corresponds to a set of elements upon which the operations of the algebra are defined. In the next section, we will look at how equations may be used to characterize the behavioral aspects of an abstract data type modeled by an algebra.

**Abstract data types – *foundational perspective***

- unambiguous values in some *semantic* domain

**Mathematical models – *types as constraints***

- algebra – *set oriented*
- second order lambda calculus – *polymorphic types*
- constructive mathematics – *formulas as types*

1-3

Slide 1-3: Mathematical models for types

Second order lambda calculus has been used to model information hiding and the polymorphism supported by inheritance and templates. In the next chapter we will study this approach in more detail.

In both approaches, the meaning of a type is (ultimately) a set of elements satisfying certain restrictions. However, in a more abstract fashion, we may regard a type as specifying a constraint. The better we specify the constraint, the more tightly the corresponding set of elements will be defined (and hence the smaller the set). A natural consequence of the idea of *types as constraints* is to characterize types by means of logical formulas. This is the approach taken by type theories based on constructive logic, in which the notion of *formulas as types* plays an important role. Although we will not study type theories based on constructive logic explicitly, our point of view is essentially to regard types as constraints, ranging from purely syntactical constraints (as expressed in a signature) to semantic constraints (as may be expressed in contracts).

From the perspective of types as constraints, a typing system may contribute to a language framework guiding a system designer's conceptualization and supporting the verification (based on the formal properties of the types employed) of the consistency of the descriptive information provided by the program. Such an approach is to be preferred (both from a pragmatic and theoretical point of view) to an *ad hoc* approach employing special annotations and support mechanisms, since these may become quite complicated and easily lead to unexpected interactions.

**Formal models** There is a wide variety of formal models available in the literature. These include algebraic models (to characterize the meaning of abstract data types), models based on the lambda-calculus and its extensions (which are primarily used for a type theoretical analysis of object-oriented language constructs), algebraic process calculi (which may be used to characterize the behavior of concurrent objects), operational and denotational semantic models (to capture structural and behavioral properties of programs), and various specification languages based on first or higher-order logics (which may be used to specify the desired behavior of collections of objects).

We will limit ourselves to studying algebraic models capturing the properties

of abstract data types and objects (section 1.2.4), type calculi based on typed extensions of the lambda calculus capturing the various flavors of polymorphism and subtyping (sections ??–??), and an operational semantic model characterizing the behavior of objects sending messages (section ??).

Both the algebraic and type theoretical models are primarily intended to clarify the means we have to express the desired behavior of objects and the restrictions that must be adhered to when defining objects and their relations. The operational characterization of object behavior, on the other hand, is intended to give a more precise characterization of the notion of state and state changes underlying the verification of object behavior by means of assertion logics.

Despite the numerous models introduced there are still numerous approaches not covered here. One approach worth mentioning is the work based on the *pi-calculus*. The *pi-calculus* is an extension of algebraic process calculi that allow for communication via named channels. Moreover, the *pi-calculus* allows for a notion of migration and the creation and renaming of channels. A semantics of object-based languages based on the *pi-calculus* is given in [Walker90]. However, this semantics does not cover inheritance or subtyping. A higher-order object-oriented programming language based on the *pi-calculus* is presented in [PRT93].

Another approach of interest, also based on process calculi, is the object calculus (OC) described in [Nier93]. OC allows for modeling the operational semantics of concurrent objects. It merges the notions of agents, as used in process calculi, with the notion of functions, as present in the lambda calculus.

For alternative models the reader may look in the `comp.theory` newsgroup to which information concerning formal calculi for OOP is posted by Tom Mens of the Free University, Brussels.

### 1.1.3 Objectives of typed OOP

Before losing ourselves in the details of mathematical models of types, we must reflect on what we may expect from a type system and what not (at least not currently).

From a theoretical perspective our ideal is, in the words of [DT88], to arrive at a simple type theory that provides a consistent and flexible framework for *system descriptions* (in order to provide the programmer with sufficient descriptive power and to aid the construction of useful and understandable software, while allowing the efficient utilization of the underlying hardware).

The question now is, what support does a typing system provide in this respect. In slide 1-4, a list is given of aspects in which a typing system may be of help.

One important benefit of regarding ADTs as real types is that realizations of ADTs become so-called *first class citizens*, which means that they may be treated as any other value in the language, for instance being passed as a parameter. In contrast, syntactic solutions (such as the module of Modula-2 and the package of Ada) do not allow this.

Pragmatically, the objective of a type system is (and has been) the prevention of errors. However, if the type system lacks expressivity, adequate control for errors may result in becoming over-restrictive.

**Objectives of typed OOP – *system description***

- packaging in a coherent manner
- flexible style of associating operations with objects
- inheritance of description components – *reuse, understanding*
- separation of specification and implementation
- explicit typing to guide binding decisions

1-4

Slide 1-4: Object orientation and types

In general, the more expressive the type system the better the support that the compiler may offer. In this respect, associating constructors with types may help in relieving the programmer from dealing with simple but necessary tasks such as the initialization of complex structures. Objects, in contrast to modules or packages, allow for the automatic (compiler supported) initializations of instances of (abstract) data types, providing the programmer with relief from an error-prone routine.

Another area in which a type system may make the life of a programmer easier concerns the association of operations with objects. A polymorphic type system is needed to understand the automatic dispatching for virtual functions and the opportunity of overloading functions, which are useful mechanisms to control the complexity of a program, provided they are well understood.

Reuse and understanding are promoted by allowing inheritance and refinement of description components. (As remarked earlier, inheritance and refinement may be regarded as the essential contribution of object-oriented programming to the practice of software development.) It goes without saying that such reuse needs a firm semantical basis in order to achieve the goal of reliable and maintainable software.

Another important issue for which a powerful type system can provide support is the separation of specification and implementation. Naturally, we expect our type system to support type-safe separate compilation. But in addition, we may think of allowing multiple implementations of a single (abstract type) specification. Explicit typing may then be of help in choosing the right binding when the program is actually executed. For instance in a parallel environment, behavior may be realized in a number of ways that differ in the degree to which they affect locality of access and how they affect, for example, load balancing. With an eye to the future, these are problems that may be solved with a good type system (and accompanying compiler).

One of the desiderata for a type system for OOP, laid down in [DT88], is the separation of a *behavioral hierarchy* (specifying the behavior of a type in an abstract sense) and an *implementation hierarchy* (specifying the actual realization of that behavior). Separation is needed to accommodate the need for multiple realizations and to resolve the tension between subtyping and inheritance (a

tension we have already noted in sections ?? and ??).

**Remark** In these chapters we cannot hope to do more than get acquainted with the material needed to understand the problems involved in developing a type system for object-oriented programming. For an alternative approach, see [Palsberg94].

## 1.2 Algebraic specification

Algebraic specification techniques have been developed as a means to specify the design of complex software systems in a formal way. The algebraic approach has been motivated by the notion of *information hiding* put forward in [Parnas72a] and the ideas concerning *abstraction* expressed in [Ho72]. Historically, the ADJ-group (see Goguen *et al.*, 1978) provided a significant impetus to the algebraic approach by showing that abstract data types may be interpreted as (many sorted) algebras. (In the context of algebraic specifications the notion of *sorts* has the same meaning as *types*. We will, however, generally speak of *types*.)

As an example of an algebraic specification, look at the module defining the data type *Bool*, as given in slide 1-5.

**Algebraic specification – ADT**

```

adt bool is
functions
  true : bool
  false : bool
  and, or : bool * bool -> bool
  not : bool -> bool
axioms
  [B1] and(true,x) = x
  [B2] and(false,x) = false
  [B3] not(true) = false
  [B4] not(false) = true
  [B5] or(x,y) = not(and(not(x),not(y)))
end

```

Bool

1-5

Slide 1-5: The ADT *Bool*

In this specification two constants are introduced (the zero-ary functions *true* and *false*), three functions (respectively *and*, *or* and *not*). The *or* function is defined by employing *not* and *and*, according to a well-known logical law. These functions may all be considered to be (strictly) related to the type *bool*. Equations are used to specify the desired characteristics of elements of type *bool*. Obviously,

this specification may mathematically be interpreted as (simply) a boolean algebra.

**Mathematical models** The mathematical framework of algebras allows for a direct characterization of the behavioral aspects of abstract data types by means of equations, provided the specification is consistent. Operationally, this allows for the execution of such specifications by means of term rewriting, provided that some (technical) constraints are met. The model-theoretic semantics of algebraic specifications centers around the notion of *initial algebras*, which gives us the preferred model of a specification.

To characterize the behavior of *objects* (that may modify their state) in an algebraic way, we need to extend the basic framework of initial algebra models either by allowing so-called *multiple world* semantics or by making a distinction between hidden and observable sorts (resulting in the notion of an object as an *abstract machine*). As a remark, in our treatment we obviously cannot avoid the use of some logico-mathematical formalism. If needed, the concepts introduced will be explained on the fly. Where this does not suffice, the interested reader is referred to any standard textbook on mathematical logic for further details.

### 1.2.1 Signatures – generators and observers

Abstract data types may be considered as modules specifying the values and functions belonging to the type. In [Dahl92], a type  $T$  is characterized as a tuple specifying the set of elements constituting the type  $T$  and the collection of functions related to the type  $T$ . Since constants may be regarded as zero-ary functions (having no arguments), we will speak of a *signature*  $\Sigma$  or  $\Sigma_T$  defining a particular type  $T$ . Also, in accord with common parlance, we will speak of the sorts  $s \in \Sigma$ , which are the sorts (or types) occurring in the declaration of the functions in  $\Sigma$ . See slide 1-6.

**Signature – names and profiles**  $\Sigma$

- $f : s_1 \times \dots \times s_n \rightarrow s$

**Functions – for  $T$**

- constants –  $c : \rightarrow T$   $C$
- producers –  $g : s_1 \times \dots \times s_n \rightarrow T$   $P$
- observers –  $f : T \rightarrow s_i$   $O$

**Type – generators**

- $\Sigma_T = P_T \cup O_T, C_T \subset P_T, P_T \cap O_T = \emptyset$

1-6

Slide 1-6: Algebraic specification

A *signature* specifies the names and (function) profiles of the constants and functions of a data type. In general, the profile of a function is specified as

- $f : s_1 \times \dots \times s_n \rightarrow s$

where  $s_i (i = 1..n)$  are the sorts defining the domain (that is the types of the arguments) of the function  $f$ , and  $s$  is the sort defining the codomain (or result type) of  $f$ . In the case  $n = 0$  the function  $f$  may be regarded as a constant. More generally, when  $s_1, \dots, s_n$  are all unrelated to the type  $T$  being defined, we may regard  $f$  as a relative constant. Relative constants are values that are assumed to be defined in the context where the specification is being employed.

The functions related to a data type  $T$  may be discriminated according to their role in defining  $T$ . We distinguish between *producers*  $g \in P_T$ , that have the type  $T$  under definition as their result type, and *observers*  $f \in O_T$ , that have  $T$  as their argument type and deliver a result of a type different from  $T$ . In other words, producer functions define how elements of  $T$  may be constructed. (In the literature one often speaks of *constructors*, but we avoid this term because it already has a precisely defined meaning in the object-oriented programming language C++.) In contrast, observer functions do not produce values of  $T$ , but give instead information on some particular aspect of  $T$ .

The signature  $\Sigma_T$  of a type  $T$  is uniquely defined by the union of producer functions  $P_T$  and observer functions  $O_T$ . Constants of type  $T$  are regarded as a subset of the producer functions  $P_T$  defining  $T$ . Further, we require that the collection of producers is disjoint from the collection of observers for  $T$ , that is  $P_T \cap O_T = \emptyset$ .

**Generators** The producer functions actually defining the values of a data type  $T$  are called the *generator basis* of  $T$ , or generators of  $T$ . The generators of  $T$  may be used to enumerate the elements of  $T$ , resulting in the collection of  $T$  values that is called the *generator universe* in [Dahl92]. See slide 1-7.

**Generators – values of  $T$**   $T$

- generator basis –  $G_T = \{g \in P_T\}$
- generator universe –  $GU_T = \{v_1, v_2, \dots\}$

**Examples**

- $G_{Bool} = \{t, f\}$ ,  $GU_{Bool} = \{t, f\}$
- $G_{Nat} = \{0, S\}$ ,  $GU_{Nat} = \{0, S0, SS0, \dots\}$
- $G_{SetA} = \{\emptyset, add\}$ ,  $GU_{SetA} = \{\emptyset, add(\emptyset, a), \dots\}$

1-7

Slide 1-7: Generators – basis and universe

The generator universe of a type  $T$  consists of the closed (that is variable-free) terms that may be constructed using either constants or producer functions of  $T$ . As an example, consider the data type *Bool* with generators  $t$  and  $f$ . Obviously,

the value domain of *Bool*, the generator universe  $GU_{Bool}$  consists only of the values *t* and *f*.

As another example, consider the data type *Nat* (representing the natural numbers) with generator basis  $G_{Nat} = \{0, S\}$ , consisting of the constant 0 and the successor function  $S : Nat \rightarrow Nat$  (that delivers the successor of its argument). The terms that may be constructed by  $G_{Nat}$  is the set  $GU_{Nat} = \{0, S0, SS0, \dots\}$ , which uniquely corresponds to the natural numbers  $\{0, 1, 2, \dots\}$ . (More precisely, the natural numbers are isomorphic with  $GU_{Nat}$ .)

In contrast, given a type *A* with element *a*, *b*, ..., the generators of  $Set_A$  result in a universe that contains terms such as  $add(\emptyset, a)$  and  $add(add(\emptyset, a), a)$  which we would like to identify, based on our conception of a set as containing only one exemplar of a particular value. To effect this we need additional equations imposing constraints expressing what we consider as the desired shape (or *normal form*) of the values contained in the universe of *T*. However, before we look at how to extend a signature  $\Sigma$  defining *T* with equations defining the (behavioral) properties of *T* we will look at another example illustrating how the choice of a generator basis may affect the structure of the value domain of a data type.

In the example presented in slide 1-8, the profiles are given of the functions that may occur in the signature specifying sequences. (The notation  $\_$  is used to indicate parameter positions.)

**Sequences**

Seq

- $\varepsilon : seqT$  empty
- $\_ \triangleright \_ : seqT \times T \rightarrow seqT$  right append
- $\_ \triangleleft \_ : T \times seqT \rightarrow seqT$  left append
- $\_ \cdot \_ : seqT \times seqT \rightarrow seqT$  concatenation
- $\langle \_ \rangle : T \rightarrow seqT$  lifting
- $\langle \_, \dots, \_ \rangle : T^n \rightarrow seqT$  multiple arguments

**Generator basis – preferably one-to-one**

- $G_{seqT} = \{\varepsilon, \triangleright\}$ ,  $GU_{seqT} = \{\varepsilon, \varepsilon \triangleright a, \varepsilon \triangleright b, \dots, \varepsilon \triangleright a \triangleright b, \dots\}$
- $G'_{seqT} = \{\varepsilon, \triangleleft\}$ ,  $GU'_{seqT} = \{\varepsilon, a \triangleleft \varepsilon, b \triangleleft \varepsilon, \dots, b \triangleleft a \triangleleft \varepsilon, \dots\}$
- $G''_{seqT} = \{\varepsilon, \cdot, \langle \_ \rangle\}$ ,  $GU''_{seqT} = \{\varepsilon, \langle a \rangle, \langle b \rangle, \dots, \varepsilon \cdot \varepsilon, \dots, \varepsilon \cdot \langle a \rangle, \dots\}$

**Infinite generator basis**

- $G'''_{seqT} = \{\varepsilon, \langle \_ \rangle, \langle \_, \_ \rangle, \dots\}$ ,  $GU'''_{seqT} = \{\varepsilon, \langle a \rangle, \langle b \rangle, \dots, \langle a, a \rangle, \dots\}$

1-8

Slide 1-8: The ADT *Seq*

Dependent on which producer functions are selected to generate the universe of *T*, the correspondence between the generated universe and the intended domain is either *one-to-one* (as for  $G$  and  $G'$ ) or *many-to-one* (as for  $G''$ ). Since we

require our specification to be first-order and finite, infinite generator bases (such as  $G'''$ ) must be disallowed, even if they result in a one-to-one correspondence. See [Dahl92] for further details.

### 1.2.2 Equations – specifying constraints

The specification of the signature of a type (which lists the *syntactic constraints* to which a specification must comply) is in general not sufficient to characterize the properties of the values of the type. In addition, we need to impose *semantic constraints* (in the form of equations) to define the meaning of the observer functions and (very importantly) to identify the elements of the type domain that are considered equivalent (based on the intuitions one has of that particular type).

|   |              |     |
|---|--------------|-----|
| <b>The equivalence relation – <i>congruence</i></b>           |              | 1-9 |
| • $x = x$   | reflexivity  |     |
| • $x = y \Rightarrow y = x$                                   | symmetry     |     |
| • $x = y \wedge y = z \Rightarrow x = z$                      | transitivity |     |
| • $x = y \Rightarrow f(\dots, x, \dots) = f(\dots, y, \dots)$ |              |     |
| <b>Equivalence classes – <i>representatives</i></b>           |              |     |
| • abstract elements – $GU_T / \sim$                           |              |     |

Slide 1-9: Equivalence

Mathematically, the equality predicate may be characterized by the properties listed above, including *reflexivity* (stating that an element is equal to itself), *symmetry* (stating that the orientation of the formula is not important) and *transitivity* (stating that if one element is equal to another and that element is equal to yet another, then the first element is also equal to the latter). In addition, we have the property that, given that two elements are equal, the results of the function applied to them (separately) are also equal. (Technically, the latter property makes a *congruence* of the equality relation, lifting equality between elements to the function level.) See slide 1-9.

Given a suitable set of equations, in addition to a signature, we may identify the elements that can be proved identical by applying the equality relation. In other words, given an equational theory (of which the properties stated above must be a part), we can divide the generator universe of a type  $T$  into one or more subsets, each consisting of elements that are equal according to our theory. The subsets of  $GU / \sim$ , that is  $GU$  factored with respect to equivalence, may be regarded as the abstract elements constituting the type  $T$ , and from each subset we may choose a concrete element acting as a *representative* for the subset which is the equivalence class of the element.

Operationally, equations may be regarded as *rewrite rules* (oriented from left to right), that allow us to transform a term in which a term  $t_1$  occurs as a subterm into a term in which  $t_1$  is replaced by  $t_2$  if  $t_1 = t_2$ . For this procedure to be terminating, some technical restrictions must be met, amounting (intuitively) to the requirement that the right-hand side must in some sense be simpler than the left-hand side.

Also, when defining an observer function, we must specify for each possible generator case an appropriate rewriting rule. That is, each observer must be able to give a result for each generator. The example of the natural numbers, given below, will make this clear. Identifying spurious elements by rewriting a term into a canonical form is somewhat more complex, as we will see for the example of sets.

**Equational theories** To illustrate the notions introduced above, we will look at specifications of some familiar types, namely the natural numbers and sets.

In slide 1-10, an algebraic specification is given of the natural numbers (as first axiomatized by Peano).

**Natural numbers**

functions

0 : Nat

S : Nat -> Nat

mul : Nat \* Nat -> Nat

plus : Nat \* Nat -> Nat

axioms

[1] plus(x,0) = x

[2] plus(x,Sy) = S(plus(x,y))

[3] mul(x,0) = 0

[4] mul(x,Sy) = plus(mul(x,y),x)

end

Nat

1-10

Slide 1-10: The ADT *Nat*

In addition to the constant  $0$  and successor function  $S$  we also introduce a function *mul* for multiplication and a function *plus* for addition. (The notation  $Sy$  stands for application by juxtaposition; its meaning is simply  $S(y)$ .) The reader who does not immediately accept the specification in slide 1-10 as an adequate axiomatization of the natural numbers must try to unravel the computation depicted in slide 1-11.

Admittedly, not an easy way to compute with natural numbers, but fortunately term rewriting may, to a large extent, be automated (and actual calculations may be mimicked by semantics preserving primitives).

Using the equational theory expressing the properties of natural numbers, we may eliminate the occurrences of the functions *mul* and *plus* to arrive (through

|  |      |
|--|------|
| $\begin{aligned} &\text{mul}(\text{plus}(S\ 0, S\ 0), S\ 0) \text{ -[2]-}i \\ &\text{mul}(S(\text{plus}(S\ 0, 0)), S\ 0) \text{ -[1]-}i \\ &\text{mul}(SS\ 0, S\ 0) \text{ -[4]-}i \\ &\text{plus}(\text{mul}(SS0, 0), SS0) \text{ -[3]-}i \\ &\text{plus}(0, SS0) \text{ -[2*]-}i\ SS0 \end{aligned}$ | 1-11 |
|--|------|

Slide 1-11: Symbolic evaluation

symbolic evaluation) at something of the form  $S^n0$  (where  $n$  corresponds to the magnitude of the natural number denoted by the term).

The opportunity of symbolic evaluation by term rewriting is exactly what has made the algebraic approach so popular for the specification of software, since it allows (under some restrictions) for executable specifications.

Since they do not reappear in what may be considered the *normal forms* of terms denoting the naturals (that are obtained by applying the evaluations induced by the equality theory), the functions *plus* and *mul* may be regarded as *secondary* producers. They are not part of the generator basis of the type *Nat*.

Since we may consider *mul* and *plus* as secondary producers at best, we can easily see that when we define *mul* and *plus* for the case  $0$  and  $Sx$  for arbitrary  $x$ , that we have covered all possible (generator) cases. Technically, this allows us to prove properties of these functions by using structural induction on the possible generator cases. The proof obligation (in the case of the naturals) then is to prove that the property holds for the function applied to  $0$  and assuming that the property holds for applying the function to  $x$ , it also holds for  $Sx$ .

As our next example, consider the algebraic specification of the type  $Set_A$  in slide 1-12.

|  |  |      |
|--|--|------|
| <b>Sets</b> <ul style="list-style-type: none"> <li><math>G_{Set_A} = \{\emptyset, add\}</math></li> <li><math>GU_{Set_A} = \{0, add(0, a), \dots, add(add(0, a), a), \dots\}</math></li> </ul> <b>Axioms</b> <p>[S1] <math>add(add(s, x), y) = add(add(s, y), x)</math> <i>commutativity</i></p> <p>[S2] <math>add(add(s, x), x) = add(s, x)</math> <i>idempotence</i></p> | <div style="border: 1px solid black; padding: 2px; display: inline-block;">Set</div> | 1-12 |
|--|--|------|

Slide 1-12: The ADT *Set*

In the case of sets we have the problem that we do not start with a one-to-one generator base as we had with the natural numbers. Instead, we have a many-

to-one generator base, so we need equality axioms to eliminate spurious elements from the (generator) universe of sets.

**Equivalence classes**

- $\{\emptyset\}$
- $\{add(0, a), add(add(0, a), a), \dots\}$
- $\dots$
- $\{add(add(0, a), b), add(add(0, b), a), \dots\}$

$GU_{Set_A} / \sim$

1-13

Slide 1-13: Equivalence classes for *Set*

The equivalence classes of  $GU_{Set_A} / \sim$  (which is  $GU_{Set_A}$  factored by the equivalence relation), each have multiple elements (except the class representing the empty set). To select an appropriate representative from each of these classes (representing the abstract elements of the type  $Set_A$ ) we need an ordering on terms, so that we can take the smaller term as its canonical representation. See slide 1-13.

### 1.2.3 Initial algebra semantics

In the previous section we have given a rather operational characterization of the equivalence relation induced by the equational theory and the process of term rewriting that enables us to purge the generator universe of a type, by eliminating redundant elements. However, what we actually strive for is a mathematical model that captures the meaning of an algebraic specification. Such a model is provided (or rather a class of such models) by the mathematical structures known as (not surprisingly) algebras.

A *single sorted* algebra  $\mathcal{A}$  is a structure  $(A, \Sigma)$  where  $A$  is a set of values, and  $\Sigma$  specifies the signature of the functions operating on  $A$ . A *multi-sorted* algebra is a structure  $\mathcal{A} = (\{A_s\}_{s \in S}, \Sigma)$  where  $S$  is a set of sort names and  $A_s$  the set of values belonging to the sort  $s$ . The set  $S$  may be ordered (in which case the ordering indicates the subtyping relationships between the sorts). We call the (multi-sorted) structure  $\mathcal{A}$  a  $\Sigma$ -algebra.

**Mathematical model – algebra**

- $\Sigma$ -algebra –  $\mathcal{A} = (\{A_s\}_{s \in S}, \Sigma)$
- interpretation –  $eval : T_\Sigma \rightarrow \mathcal{A}$
- adequacy –  $\mathcal{A} \models t_1 = t_2 \iff E \vdash t_1 = t_2$

1-14

Slide 1-14: Interpretations and models

Having a notion of algebras, we need to have a way in which to relate an algebraic specification to such a structure. To this end we define an interpretation  $eval : T_\Sigma \rightarrow \mathcal{A}$  which maps closed terms formed by following the rules given in the specification to elements of the structure  $\mathcal{A}$ . We may extend the interpretation  $eval$  to include variables as well (which we write as  $eval : T_\Sigma(X) \rightarrow \mathcal{A}$ ), but then we also need to assume that an assignment  $\theta : X \rightarrow T_\Sigma(X)$  is given, such that when applying  $\theta$  to a term  $t$  the result is free of variables, otherwise no interpretation in  $\mathcal{A}$  exists. See slide 1-14.

**Interpretations** As an example, consider the interpretations of the specification of *Bool* and the specification of *Nat*, given in slide 1-15.

**Booleans**

- $\mathcal{B} = (\{tt, ff\}, \{\neg, \wedge, \vee\})$
- $eval_{\mathcal{B}} : T_{Bool} \rightarrow \mathcal{B} = \{or \mapsto \vee, and \mapsto \wedge, not \mapsto \neg\}$

**Natural numbers**

- $\mathcal{N} = (\mathbb{N}, \{++, +, *\})$
- $eval_{\mathcal{N}} : T_{Nat} \rightarrow \mathcal{N} = \{S \mapsto ++, mul \mapsto *, plus \mapsto +\}$

1-15

Slide 1-15: Interpretations of *Bool* and *Nat*

The structure  $\mathcal{B}$  given above is simply a boolean algebra, with the operators  $\neg$ ,  $\wedge$  and  $\vee$ . The functions *not*, *and* and *or* naturally map to their semantic counterparts. In addition, we assume that the constants *true* and *false* map to the elements *tt* and *ff*.

As another example, look at the structure  $\mathcal{N}$  and the interpretation  $eval_{\mathcal{N}}$ , which maps the functions *S*, *mul* and *plus* specified in *Nat* in a natural way. However, since we have also given equations for *Nat* (specifying how to eliminate the functions *mul* and *plus*) we must take precautions such that the requirement

$$\mathcal{N} \models eval_{\mathcal{N}}(t_1) =_{\mathcal{N}} eval_{\mathcal{N}}(t_2) \iff E_{Nat} \vdash t_1 = t_2$$

is satisfied if the structure  $\mathcal{N}$  is to count as an adequate model of *Nat*. The requirement above states that whenever equality holds for two interpreted terms (in  $\mathcal{N}$ ) then these terms must also be provably equal (by using the equations given in the specification of *Nat*), and vice versa.

As we will see illustrated later, many models may exist for a single specification, all satisfying the requirement of adequacy. The question is, do we have a means to select one of these models as (in a certain sense) the best model. The answer is yes. These are the models called *initial models*.

**Initial models** A model (in a mathematical sense) represents the meaning of a specification in a precise way. A model may be regarded as stating a commitment

with respect to the interpretation of the specification. An initial model is intuitively the least committing model, least committing in the sense that it imposes only identifications made necessary by the equational theory of a specification. Technically, an initial model is a model from which every other model can be derived by an algebraic mapping which is a homomorphism.

**Initial algebra**

- $\Sigma E$ -algebra –  $\mathcal{M} = (T_\Sigma / \sim, \Sigma / \sim)$

**Properties**

- *no junk* –  $\forall a : T_\Sigma / \sim \exists t \bullet eval_{\mathcal{M}}(t) = a$
- *no confusion* –  $\mathcal{M} \models t_1 = t_2 \iff E \vdash t_1 = t_2$

1-16

Slide 1-16: Initial models

The starting point for the construction of an initial model for a given specification with signature  $\Sigma$  is to construct a term algebra  $T_\Sigma$  with the terms that may be generated from the signature  $\Sigma$  as elements. The next step is then to factor the universe of generated terms into equivalence classes, such that two terms belong to the same class if they can be proven equivalent with respect to the equational theory of the specification. We will denote the representative of the equivalence class to which a term  $t$  belongs by  $[t]$ . Hence  $t_1 = t_2$  (in the model) *iff*  $[t_1] = [t_2]$ .

So assume that we have constructed a structure  $\mathcal{M} = (T_\Sigma / \sim, \Sigma)$  then; finally, we must define an interpretation, say  $eval_{\mathcal{M}} : T_\Sigma \rightarrow \mathcal{M}$ , that assigns closed terms to appropriate terms in the term model (namely the representatives of the equivalence class of that term). Hence, the interpretation of a function  $f$  in the structure  $\mathcal{M}$  is such that

$$f_{\mathcal{M}}([t_1], \dots, [t_n]) = [f(t_1, \dots, t_n)]$$

where  $f_{\mathcal{M}}$  is the interpretation of  $f$  in  $\mathcal{M}$ . In other words, the result of applying  $f$  to terms  $t_1, \dots, t_n$  belongs to the same equivalence class as the result of applying  $f_{\mathcal{M}}$  to the representatives of the equivalence classes of  $t_1, \dots, t_n$ . See slide 1-16.

An initial algebra model has two important properties, known respectively as the *no junk* and *no confusion* properties. The *no junk* property states that for each element of the model there is some term for which the interpretation in  $\mathcal{M}$  is equal to that element. (For the  $T_\Sigma / \sim$  model this is simply a representative of the equivalence class corresponding with the element.) The *no confusion* property states that if equality of two terms can be proven in the equational theory of the specification, then the equality also holds (semantically) in the model, and vice versa. The *no confusion* property means, in other words, that sufficiently many identifications are made (namely those that may be proven to hold), but no more than that (that is, no other than those for which a proof exists). The latter property is why we may speak of an initial model as the least committing model; it simply gives no more meaning than is strictly needed.

The initial model constructed from the term algebra of a signature  $\Sigma$  is intuitively a very natural model since it corresponds directly with (a subset of) the generator universe of  $\Sigma$ . Given such a model, other models may be derived from it simply by specifying an appropriate interpretation. For example, when we construct a model for the natural numbers (as specified by *Nat*) consisting of the generator universe  $\{0, S0, SS0, \dots\}$  and the operators  $\{++, +, \star\}$  (which are defined as  $S^n ++ = S^{n+1}$ ,  $S^n \star S^m = S^{n+m}$  and  $S^n + S^m = S^{n+m}$ ) we may simply derive from this model the structure  $(\{0, 1, 2, \dots\}, \{++, +, \star\})$  for which the operations have their standard arithmetical meaning. Actually, this structure is also an initial model for *Nat*, since we may also make the inverse transformation.

More generally, when defining an initial model only the structural aspects (characterizing the behavior of the operators) are important, not the actual contents. Technically, this means that initial models are defined up to isomorphism, that is a mapping to equivalent models with perhaps different contents but an identical structure. Not in all cases is a structure derived from an initial model itself also an initial model, as shown in the example below.

**Example** Consider the specification of *Bool* as given before. For this specification we have given the structure  $\mathcal{B}$  and the interpretation  $eval_{\mathcal{B}}$  which defines an initial model for *Bool*. (Check this!)

|   |   |
|---|---|
| <b>Structure</b> $\mathcal{B} = (\{tt, ff\}, \{\neg, \wedge, \vee\})$<br><ul style="list-style-type: none"> <li><math>eval_{\mathcal{B}} : T_{\Sigma_{Bool}} \rightarrow \mathcal{B} = \{or \mapsto \vee, not \mapsto \neg\}</math></li> <li><math>eval_{\mathcal{B}} : T_{\Sigma_{Nat}} \rightarrow \mathcal{B} = \{S \mapsto \neg, mul \mapsto \wedge, plus \mapsto xor\}</math></li> </ul> | <div style="border: 1px solid black; padding: 2px; display: inline-block;"><math>\mathcal{B}</math></div> |
|---|---|

1-17

Slide 1-17: Structure and interpretation

We may, however, also use the structure  $\mathcal{B}$  to define an interpretation of *Nat*. See slide 1-17. The interpretation  $eval_{\mathcal{B}} : T_{Nat} \rightarrow \mathcal{B}$  is such that  $eval_{\mathcal{B}}(0) = ff$ ,  $eval_{\mathcal{B}}(Sx) = \neg eval_{\mathcal{B}}(x)$ ,  $eval_{\mathcal{B}}(mul(x, y)) = eval_{\mathcal{B}}(x) \wedge eval_{\mathcal{B}}(y)$  and  $eval_{\mathcal{B}}(plus(x, y)) = xor(eval_{\mathcal{B}}(x), eval_{\mathcal{B}}(y))$ , where  $xor(p, q) = (p \vee q) \wedge (\neg(p \wedge q))$ . The reader may wish to ponder on what this interpretation effects. The answer is that it interprets *Nat* as specifying the naturals modulo 2, which discriminates only between odd and even numbers. Clearly, this interpretation defines not an initial model, since it identifies all odd numbers with *ff* and all even numbers with *tt*. Even if we replace *ff* by 0 and *tt* by 1, this is not what we generally would like to commit ourselves to when we speak about the natural numbers, simply because it assigns too much meaning.

#### 1.2.4 Objects as algebras

The types for which we have thus far seen algebraic specifications (including *Bool*, *Seq*, *Set* and *Nat*) are all types of a mathematical kind, which (by virtue of being mathematical) define operations without side-effects. Dynamic state changes, that

is side-effects, are often mentioned as determining the characteristics of objects in general. In the following we will explore how we may deal with assigning meaning to dynamic state changes in an algebraic framework.

Let us look first at the abstract data type *stack*. The type *stack* may be considered as one of the ‘real life’ types in the world of programming. See slide 1-18.

|  |              |      |
|--|--------------|------|
| <b>Abstract Data Type – applicative</b>  | <i>Stack</i> | 1-18 |
| <p><i>functions</i></p> <p>new : stack;<br/> push : element * stack → stack;<br/> empty : stack → boolean;<br/> pop : stack → stack;<br/> top : stack → element;</p> <p><i>axioms</i></p> <p>empty( new ) = true<br/> empty( push(x,s) ) = false<br/> top( push(x,s) ) = x<br/> pop( push(x,s) ) = s</p> <p><i>preconditions</i></p> <p>pre: pop( s : stack ) = not empty(s)<br/> pre: top( s : stack ) = not empty(s)</p> <p><i>end</i></p> |              |      |

Slide 1-18: The ADT *Stack*

Above, a stack has been specified by giving a signature (consisting of the functions *new*, *push*, *empty*, *pop* and *top*). In addition to the axioms characterizing the behavior of the stack, we have included two pre-conditions to test whether the stack is empty in case *pop* or *top* is applied. The pre-conditions result in conditional axioms for the operations *pop* and *top*. Conditional axioms, however, do preserve the initial algebra semantics.

The specification given above is a maximally abstract description of the behavior of a stack. Adding more implementation detail would disrupt its nice applicative structure, without necessarily resulting in different behavior (from a sufficiently abstract perspective).

The behavior of elements of abstract data types and objects is characterized by state changes. State changes may affect the value delivered by observers or methods. Many state changes (such as the growing or shrinking of a set, sequence or stack) really are nothing but applicative transformations that may mathematically be described by the input-output behavior of an appropriate function.

An example in which the value of an object on some attribute is dependent on the history of the operations applied to the object, instead of the structure of

the object itself (as in the case of a stack) is the object *account*, as specified in slide 1-19. The example is taken from [Goguen].

|  |  |      |
|--|--|------|
| <b>Dynamic state changes – objects</b><br>object account is<br>functions<br>$\text{bal} : \text{account} \rightarrow \text{money}$<br>methods<br>$\text{credit} : \text{account} * \text{money} \rightarrow \text{account}$<br>$\text{debit} : \text{account} * \text{money} \rightarrow \text{account}$<br>error<br>$\text{overdraw} : \text{money} \rightarrow \text{money}$<br>axioms<br>$\text{bal}(\text{new}(A)) = 0$<br>$\text{bal}(\text{credit}(A,M)) = \text{bal}(A) + M$<br>$\text{bal}(\text{debit}(A,M)) = \text{bal}(A) - M \text{ if } \text{bal}(A) \geq M$<br>error-axioms<br>$\text{bal}(\text{debit}(A,M)) = \text{overdraw}(M) \text{ if } \text{bal}(A) < M$<br>end | <div style="border: 1px solid black; padding: 2px; display: inline-block;">account</div> | 1-19 |
|--|--|------|

Slide 1-19: The algebraic specification of an account

An *account* object has one attribute function (called *bal*) that delivers the amount of money that is (still) in the account. In addition, there are two method functions, *credit* and *debit* that may respectively be used to add or withdraw money from the account. Finally, there is one special error function, *overdraw*, that is used to define the result of *balance* when there is not enough money left to grant a *debit* request. Error axioms are needed whenever the proper axioms are stated conditionally, that is contain an *if* expression. The conditional parts of the axioms, including the error axioms, must cover all possible cases.

Now, first look at the form of the axioms. The axioms are specified as

$$fn(\text{method}(\text{Object}, \text{Args})) = \text{expr}$$

where *fn* specifies an attribute function (*bal* in the case of *account*) and *method* a method (either *new*, which is used to create new accounts, *credit* or *debit*). By convention, we assume that  $\text{method}(\text{Object}, \dots) = \text{Object}$ , that is that a method function returns its first argument. Applying a method thus results in redefining the value of the function *fn*. For example, invoking the method  $\text{credit}(\text{acc}, 10)$  for the account *acc* results in modifying the function *bal* to deliver the value  $\text{bal}(\text{acc}) + 10$  instead of simply  $\text{bal}(\text{acc})$ . In the example above, the axioms define the meaning of the function *bal* with respect to the possible method applications. It is not difficult to see that these operations are of a non-applicative nature, non-applicative in the sense that each time a method is invoked the actual definition

of *bal* is changed. The change is necessary because, in contrast to, for example, the functions employed in a boolean algebra, the actual value of the account may change in time in a completely arbitrary way. A first order framework of (multi sorted) algebras is not sufficiently strong to define the meaning of such changes. What we need may be characterized as a *multiple world semantics*, where each world corresponds to a possible state of the account. As an alternative semantics we will also discuss the interpretation of an object as an *abstract machine*, which resembles an (initial) algebra with hidden sorts.

**Multiple world semantics** From a semantic perspective, an object that changes its state may be regarded as moving from one world to another, when we see a world as representing a particular state of affairs. Take for example an arbitrary (say John's) account, which has a balance of 500. We may express this as  $balance(account.John) = 500$ . Now, when we invoke the method *credit*, as in  $credit(account.John, 200)$ , then we expect the balance of the account to be raised to 700. In the language of the specification, this is expressed as

$$bal(credit(account.John, 200)) = bal(account.John) + 200$$

Semantically, the result is a state of affairs in which  $bal(account.John) = 700$ .

In [Goguen] an operational interpretation is given of a multiple world semantics by introducing a database  $D$  (that stores the values of the attribute functions of objects as first order terms) which is transformed as the result of invoking a method, into a new database  $D'$  (that has an updated value for the attribute function modified by the method). The meaning of each database (or world) may be characterized by an algebra and an interpretation as before.

The rules according to which transformations on a database take place may be formulated as in slide 1-20.

**Multiple world semantics – inference rules**

- $\langle f(t_1, \dots, t_n), D \rangle \rightarrow \langle v, D \rangle$  attribute
- $\langle m(t_1, \dots, t_n), D \rangle \rightarrow \langle t_1, D' \rangle$  method
- $\langle t, D \rangle \rightarrow \langle t', D' \rangle \Rightarrow \langle e(\dots, t, \dots), D \rangle \rightarrow \langle e(\dots, t', \dots), D' \rangle$

1-20

Slide 1-20: The interpretation of change

The first rule (*attribute*) describes how attribute functions are evaluated. Whenever a function  $f$  with arguments  $t_1, \dots, t_n$  evaluates to a value (or expression)  $v$ , then the term  $f(t_1, \dots, t_n)$  may be replaced by  $v$  without affecting the database  $D$ . (We have simplified the treatment by omitting all aspects having to do with matching and substitutions, since such details are not needed to understand the process of symbolic evaluation in a multiple world context.) The next rule (*method*) describes the result of evaluating a method. We assume that invoking the method changes the database  $D$  into  $D'$ . Recall that, by convention, a method

returns its first argument. Finally, the last rule (*composition*) describes how we may glue all this together.

No doubt, the reader needs an example to get a picture of how this machinery actually works.

**Example - a counter object**

```

object ctr is
  function n : ctr -> nat
  method incr : ctr -> ctr
  axioms
    n(new(C)) = 0
    n(incr(C)) = n(C) + 1
end

```

ctr

1-21

Slide 1-21: The object *ctr*

In slide 1-21, we have specified a simple object *ctr* with an attribute function *value* (delivering the value of the counter) and a method function *incr* (that may be used to increment the value of the counter).

**Abstract evaluation**

```

i n(incr(incr(new(C)))) , { C } i -[new]- i
i n(incr(incr(C))) , { C[n:=0] } i -[incr]- i
i n(incr(C)) , { C[n:=1] } i -[incr]- i
i n(C) , { C[n:=2] } i -[n]- i
i 2 , { C[n:=2] } i

```

1-22

Slide 1-22: An example of abstract evaluation

The end result of the evaluation depicted in slide 1-22 is the value 2 and a context (or database) in which the value of the counter *C* is (also) 2. The database is modified in each step in which the method *incr* is applied. When the attribute function *value* is evaluated the database remains unchanged, since it is merely consulted.

**Objects as abstract machines** Multiple world semantics provide a very powerful framework in which to define the meaning of object specifications. Yet, as illustrated above, the reasoning involved has a very operational flavor and lacks the appealing simplicity of the initial algebra semantics given for abstract data types. As an alternative, [Goguen] propose an interpretation of objects (with dynamic state changes) as *abstract machines*.

Recall that an initial algebra semantics defines a model in which the elements are equivalence classes representing the abstract values of the data type. In effect, initial models are defined only up to isomorphism (that is, structural equivalence with similar models). In essence, the framework of initial algebra semantics allows us to abstract from the particular representation of a data type, when assigning meaning to a specification. From this perspective it does not matter, for example, whether integers are represented in binary or decimal notation.

The notion of *abstract machines* generalizes the notion of initial algebras in that it loosens the requirement of (structural) isomorphism, to allow for what we may call *behavioral equivalence*. The idea underlying the notion of behavioral equivalence is to make a distinction between *visible* sorts and *hidden* sorts and to look only at the visible sorts to determine whether two algebras  $\mathcal{A}$  and  $\mathcal{B}$  are behaviorally equivalent. According to [Goguen], two algebras  $\mathcal{A}$  and  $\mathcal{B}$  are behaviorally equivalent if and only if the result of evaluating any expression of a visible sort in  $\mathcal{A}$  is the same as the result of evaluating that expression in  $\mathcal{B}$ .

Now, an *abstract machine* (in the sense of Goguen and Meseguer, 1986) is simply the equivalence class of behaviorally equivalent algebras, or in other words the maximally abstract characterization of the visible behavior of an abstract data type with (hidden) states.

The notion of abstract machines is of particular relevance as a formal framework to characterize the (implementation) refinement relation between objects. For example, it is easy to determine that the behavior of a stack implemented as a list is equivalent to the behavior of a stack implemented by a pointer array, whereas these objects are clearly not equivalent from a structural point of view. Moreover, the behavior of both conform (in an abstract sense) with the behavior specified in an algebraic way. Together, the notions of abstract machine and behavioral equivalence provide a formalization of the notion of *information hiding* in an algebraic setting. In the chapters that follow we will look at alternative formalisms to explain information hiding, polymorphism and behavioral refinement.

### 1.3 Decomposition – modules versus objects

Abstract data types allow the programmer to define a complex data structure and an associated collection of functions, operating on that structure, in a consistent way. Historically, the idea of data abstraction was originally not type-oriented but arose from a more pragmatic concern with information hiding and representation abstraction, see [Parnas72b]. The first realization of the idea of data abstraction was in the form of modules grouping a collection of functions and allowing the actual representation of the data structures underlying the values of the (abstract) type domain to be hidden, see also [Parnas72a].

In [Cook90], a comparison is made between the way in which abstract data types are realized traditionally (as modules) and the way abstract data types may be realized using object-oriented programming techniques. According to [Cook90], these approaches must be regarded as being orthogonal to one another and, being to some extent complementary, deserve to be integrated in a common framework.