Topic 22
Algebras

- The **prerequisite** for following this (part of the) lecture is that you understand the mathematical concepts of sets and of functions as covered in earlier lectures.
- The **aims** are
  - ⋆ to cover the mathematical concepts of algebras such as they are used in computing science and software engineering and
  - ⋆ to cover, even in this early lecture, the algebraic specification of what is known, in computing science and software engineering, as abstract data types (ADTs).

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

Fredsvej 11
DK-2840 Holte
Denmark
Tiergartenstraße 17
D 69121 Heidelberg
Germany

- The **objective** is
  - ⋆ to ensure that the course student from as early as possible can use and handle this concept of specification algebras, at ease and with determination.
- The **treatment** is systematic to semiformal.

It is a main purpose of this lecture to basically just introduce the jargon — the language, as it were — of algebras.

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

Characterisation 8.50 By an *algebra* we, loosely, mean a possibly infinite set of entities and a usually finite set of operations over these entities. ■

- In software engineering algebras play two central mathematical roles.
- The way we structure specifications and programmes (in schemes, classes, modules, objects) can perhaps best be understood with reference to algebra.
- Steps of development, from abstract specifications to concrete ones, can likewise best be understood as some algebra morphisms.

© Dines Bjørner, Fredsvej 11, DK-2840 Holte, Denmark and © Springer, Tiergartenstraße 17, D 69121 Heidelberg, Germany

Introduction

- The concept of *algebra* is a mathematical concept that allows us to abstract observations that may have their background in topics other than mathematics.
- The concept of function can be seen as one such concept, which we, in a previous lecture, “related back” to phenomena in some actual world.
- Our concept of functions, as well as the basis of the concept of mathematical logic to be covered in a next lecture can both have their presentation improved by presenting some of their structure algebraically.
- The *function algebra* thus consists of the space of all functions and a few operations such as function abstraction, function application, function composition, taking the definition set of a function, taking the range set of a function and, last, taking the fix point of a function.

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

© Dines Bjørner, Fredsvej 11, DK-2840 Holte, Denmark and © Springer, Tiergartenstraße 17, D 69121 Heidelberg, Germany

Formal Definition of the Algebra Concept

- We shall primarily take an algebraic approach when determining, i.e., when deciding upon, the form of, and developing software development descriptions.
- An *algebraic system* is a set,[12] A (finite or infinite), and a set[13], Ω, (usually finite), of operations:

$$(A, \Omega)$$

$$A = \{a_1, a_2, ..., a_m, ...\}, \Omega = \{\omega_1, \omega_2, ..., \omega_o\}$$

- Set A is the *carrier* of the algebraic system, and
- Ω is a collection of operations defined on A.

[12] We usually do not say what the elements of this set are, it is just a set.
[13] Similarly: Just a set!

Fredsvej 11, DK-2840 Holte, Denmark
Tiergartenstraße 17, D 69121 Heidelberg, Germany

- Each operation $\omega_i : \Omega$ (ω_i in Ω, i.e., ω_i of type Ω) is a function of some ***arity***, say n, taking operands, i.e., argument values in A, and yielding a result value in A:

$$\omega(a_{i_1}, a_{i_2}, \ldots, a_{i_n}) = a$$

- That is, ω_i is of type $A^n \to A$. Different functions (in Ω) may have different arities.
- Think of arity as a functional, a function that applies to functions and yields their arity:

$$\mathbf{type}\ \ \mathsf{arity}: \Omega \to \mathbf{Nat}, \quad \mathsf{arity}(\omega_i) = n$$

How Do Algebras Come About?

- Popular *software devices*, also known as *abstract data types*, such as
  - ⋆ *stacks*,
  - ⋆ *queues*,
  - ⋆ *tables*,
  - ⋆ *graphs*,
  - ⋆ etc.,

  can all be seen as algebras.

Example 8.29 *"Everyday" Algebras:*

1. *A Stack Algebra:* The *stack algebra* has, as *carrier*, the union
   - of the set of all stack element values
   - with the set of all stack values,

   and

   - **create** empty stack,
   - **top** of stack,
   - **push** onto stack,
   - **pop** from stack and
   - **is_empty** stack

   as *operations*.

© Dines Bjørner and © Springer
Fredsvej 11 Tiergartenstraße 17
DK-2840 Holte D 69121 Heidelberg
Denmark Germany

2. *A Queue Algebra:* The *queue algebra* has, as *carrier*, the union

- of the set of all queue element values
- with the set of all queue values

and, for example,

- create empty queue,
- enqueue,
- dequeue,
- first ("oldest"),
- last ("youngest"), and
- is_empty queue

as *operations*.

3. *A Directory Algebra:* The *directory algebra* has, as *carrier*, the union of

- the set of all directory entry values (i.e., of value triples of entry name, date and information values)
- with the set of all directory values

and, for example,

- **create** empty directory,
- **insert** entry in directory,
- directory **look-up**,
- **edit** directory entry and
- **remove** directory entry

as *operations.*

4. *A Directed, Acyclic Graph Algebra:* The *directed acyclic graph algebra* has, as *carrier*, the union of

- the set of all node labels,
- the set of all edges, and
- the set of all acyclic graphs of (these) labeled nodes and unlabeled edges,

and, for example,

- create empty graph,
- insert_node in graph,
- insert_edge in graph,
- trace edges in graph from node to node,
- depth_first_ search in graph and
- breadth_first_ search in graph,

as *operations*.

5. *Patient Medical Record Algebra:* The *patient medical record algebra* has, as *carrier*,

- all conceivable patient medical records,
  - ⋆ each consisting of one dossier.

  Each dossier consists of one or more sheets (i.e., records) that are of the following kinds:

  - ⋆ prior medical history,
  - ⋆ interview records,
  - ⋆ analysis records,
  - ⋆ diagnostics determination,
  - ⋆ treatment plans (including prescriptions),
  - ⋆ observations of effects of treatment,
  - ⋆ etc.
- In addition the *carrier* also includes these different kinds of sheets.

That is, the *carrier* is quite complex. The *patient medical record algebra* has, for example, the following *operations:*

- creation of a new medical record,
- inserting new information,
- editing previous (i.e., old) information,
- copying a sheet or a dossier and
- shredding a dossier.

■

- Algebras may have finite or infinite carriers, i.e., carriers with finite or infinite numbers of elements of possibly different types.

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

Kinds of Algebras

- There are various kinds of algebras.
- It is important to understand which kinds of algebras are of interest to software engineering and which are not.
- For that purpose we explicate the variety of algebras that you may come across.

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/˜db

© Dines Bjørner Fredsvej 11 DK-2840 Holte Denmark and © Springer Tiergartenstraße 17 D 69121 Heidelberg Germany

Concrete Algebras

- The examples above were all examples of *concrete algebras.*

Characterisation 8.51 A *concrete algebra* has sets of known, specific values as carrier, and a set of specifically given operations. ■

- That is, one knows that one has a concrete algebra when one knows the elements of the carrier and when one knows the operators and how to evaluate operation invocations.
- The Boolean algebra of a later lecture is an example of a concrete, mathematical algebra.
- Other concrete, mathematical algebras are found in Example 8.30.

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

Example 8.30 *Number Algebras:*

- *An Integer Algebra:* (Integer,$\{+,-,*\}$), an infinite carrier algebra whose operations yield all the integers.
- *A Natural Numbers Algebra:* (NatNumber,{gcd,lcm}) an infinite carrier algebra where gcd, lcm are the greatest common divisor, respectively the largest common multiple (viz.: gcd(4,6)=2, lcm(4,6)=12) operations, which yield all the natural numbers.
- *A Modulo Natural Number Algebra:* ($\Im_m = \{0,1,2,\ldots, m-1\}, \Omega = \{\oplus,\otimes\}$) is a finite carrier algebra: $\oplus$ and $\otimes$ are the addition and multiplication operations modulo m.

Several other algebras over numbers are possible. ∎

- As software engineers we shall mostly be developing concrete algebras.
- As computing scientists we shall often have occasion to explain

things in terms of abstract or universal algebras, to which we now turn.

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

Abstract Algebras

- Whereas concrete algebras are known, i.e., effectively constructed, abstract algebras are postulated, That is, they are what we shall call (and define as) 'axiomatised' in .

Characterisation 8.52 An *abstract algebra* has a sort, i.e., a presently further undefined set of entities as carrier, a set of operations, and a set of axioms that relate (i.e., constrain) properties of carrier elements and operations. ■

- The algebraic system of an abstract algebra is thus defined by a system of postulates, to be known henceforth as axioms — and to be treated in depth later.

Fredsvej 11
DK-2840 Holte
Denmark
Tiergartenstraße 17
D 69121 Heidelberg
Germany

- We shall often be using axioms
  - ⋆ to describe manifest phenomena in an actual world; and we shall likewise often be using axioms
  - ⋆ to prescribe software devices

— which will later be made "concrete", as concrete as such "phenomena" which can exist inside computers can "be".
The axiom systems should not be seen as actually "being" this or that concrete world, but "only" models of it.

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/˜db

Example 8.31 *Another Stack Algebra:* We present another version of the stack algebra of Example 8.29(1).

- There is a distinguished, unique carrier element called the empty stack: empty().
- Let s stand for any carrier stack value, i.e., stack, and let $E = \{e, e', \ldots, e'', \ldots\}$ stand for carrier stack element values. The members of E will become the elements of stacks.
- is_empty(empty()) always holds (is always true),
- whereas is_empty(push(e, s)), for any e and s, always fails to hold (is always false).
- Inquiring as to the top of a stack, s — which can be thought of as one onto which one has just pushed the element e — yields that e for any stack s: pop(push(e, s')) $= e$,
- while popping an element from the stack s' (i.e. pop(push(e, s))) yields s.

- Popping an element from, respectively inquiring as to, the **top** element of an **empty()** stack always yields the **chaotic** value, of no type, and representing the universally undefined element.

■

- We shall more explicitly use the concept of abstract algebras whenever we "lift" an example like the above by not being concrete about exactly what the elements of the stack are.
- That is, we use it when we define a *parameterised algebra*, that is, abstract, like for function abstraction, in one or more of the *sub-carriers* of the abstract algebra being defined.
- Thus we introduce the concept of *heterogeneous algebras*.

© Dines Bjørner, Fredsvej 11, DK-2840 Holte, Denmark and © Springer, Tiergartenstraße 17, D 69121 Heidelberg, Germany

Heterogeneous Algebras

Characterisation 8.53 • A *heterogeneous algebra:*

$$(\{A_1, A_2, \ldots, A_m\}, \Omega\})$$

⋆ has its carrier set A be expressible as the union of a set of disjoint sub-carriers A_i, and

⋆ associates with every operation ω in Ω a **signature**:

$$\mathsf{signature}(\omega) = A_{i_1} \times A_{i_2} \times \cdots \times A_{i_n} \to A_{i_{n+1}}$$

. ■

• Thus the kth operand of ω is of type A_{i_k}, and the result value is of type $A_{i_{n+1}}$.

Example 8.32 *Stack Algebra:* We expand on the stack algebra example, Example 8.31. Viewing that **stack** algebra as a heterogeneous algebra, the **stack** operations are (now) of the following signatures: S is the stack type, and E is the type of stack elements:

- empty: $\mathbf{Unit} \to S$,
- is_empty: $S \to \mathbf{Bool}$,
- push: $S \times E \to S$,
- top: $S \stackrel{\sim}{\to} E$, and
- pop: $S \stackrel{\sim}{\to} S$.

■

Unit is a literal. It denotes a type of one element. That element is designated by the empty parameter grouping: ().

© Dines Bjørner
Fredsvej 11
DK-2840 Holte
Denmark
and © Springer
Tiergartenstraße 17
D 69121 Heidelberg
Germany

Universal Algebras

Characterisation 8.54 • A *universal algebra* is a carrier and a set of operations with no postulates, i.e., the operations are not further constrained

. ■

The Morphism Concept

- When, in software development we transform abstract specifications to more concrete ones, then, usually, an *algebra morphism* is taking place.

- Let there be two algebras:

$$(A, \Omega), (A', \Omega')$$

- A function $\phi : A \to A'$ is said to be a *morphism* (also called a *homomorphism*) from (A, Ω) to (A', Ω') if for any $\omega \in \Omega$ and for any a_1, a_2, ..., a_n in A there is a corresponding $\omega' \in \Omega'$, such that:

$$M : \phi(\omega(a_1, a_2, \ldots, a_n)) = \omega'(\phi(a_1), \phi(a_2), \ldots, \phi(a_n))$$

- We say that the homomorphism relation M *respects* or *preserves* corresponding operations in Ω and Ω' (Fig. 8.10).

Dines Bjørner
Fredsvej 11
DK-2840 Holte
Denmark
Springer
Tiergartenstraße 17
D 69121 Heidelberg
Germany

$$
\begin{tikzcd}
A^n \arrow[r, "\omega"] \arrow[d, "\phi^n"'] & A \arrow[d, "\phi"] \\
(A')^n \arrow[r, "\omega'"'] & A'
\end{tikzcd}
$$

Figure 8.10: Morphism mapping diagram

© Dines Bjørner and © Springer
Fredsvej 11 Tiergartenstraße 17
DK-2840 Holte D 69121 Heidelberg
Denmark Germany

$$\begin{tikzcd} A^n \arrow[r, "\omega"] \arrow[d, "\phi^n"'] & A \arrow[d, "\phi"] \\ (A')^n \arrow[r, "\omega'"'] & A' \end{tikzcd}$$

Figure 8.11: Morphism Mapping Diagram (Repeat)

- ϕ^n is the n-fold Cartesian power of $\phi : A \to A'$, that is, the map $A^n \to (A')^n$, and is defined by:

$$\phi^n : (a_1, a_2, \ldots, a_n) \mapsto (\phi(a_1), \phi(a_2), \ldots, \phi(a_n))$$

- If $\phi : A \to A'$ is a homomorphism of Ω-algebras, then, by definition ϕ preserves all the operations of Ω.

Special Kinds of Morphisms

- We classify morphisms according to their properties as functions.
- If $\phi : A \to A'$ is a morphism, then we call ϕ
  - ⋆ an *isomorphism* if ϕ is *bijective*;
  - ⋆ an *epimorphism* if ϕ is *surjective*, and
  - ⋆ a *monomorphism* if ϕ is *injective*.

- Some further characterisations:
  - ⋆ The *abstract properties* of an *algebraic system* are exactly those which are *invariant* (i.e., which do not change) under *isomorphism*.
  - ⋆ For *epimorphisms*, A' is called the *homomorphic image* of A, and we regard (A', Ω') as an *abstraction* or a *model* of (A, Ω).
  - ⋆ A *monomorphism* $A \rightarrow A'$ is sometimes called an *embedding* of A into A'.

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

Dines Bjørner, Fredsvej 11, DK-2840 Holte, Denmark
Springer, Tiergartenstraße 17, D 69121 Heidelberg, Germany
DTU

- We single out morphisms that map algebras onto themselves.
- We call a morphism $\phi : A \to A'$
  - ⋆ that maps (A, Ω) into itself an *endomorphism*.
  - ⋆ If ϕ is also bijective, hence an isomorphism, $\phi : A \to A$, then we call it an *automorphism*.

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

Fredsvej 11
DK-2840 Holte
Denmark
Tiergartenstraße 17
D 69121 Heidelberg
Germany

Specification Algebras

- The mathematical concept of algebras has had a great influence on our way of presenting software designs, prescriptions for software, and, in general, any kind of documentation related also to software development.
- The whole concept of *object-orientedness* is basically an algebraic concept.
- Giving meaning, i.e., semantics, to syntactic constructs by means of presenting morphisms from syntactic algebras to semantics algebras is obviously another algebraic concept.

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

- Thus it is that in programming as well as in specification languages we find syntactic means for presenting what amounts to heterogeneous algebras.
- In `RSL` the syntactic construct for presenting a heterogeneous algebra is called a **class** expression.
- In an `RSL` **class** expression one therefore expects to find syntactic means for defining the carriers and the operations of a heterogeneous algebra.
- We now turn to this subject.
- But we first remind the reader of an earlier lecture in which we first introduced the class concept.

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

Syntactic Means of Expressing Algebras

- To define the various carriers we define their **type**s, and
- to define the various operations over these carriers we define these as function **value**s.

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

- Schematically:

class
  type
    A, B, C, D, ...
  value
    f: A → B
    f(a) ≡ ...
    g: C → D
    g(c) ≡ ...
    ...
end

- The above class expression defines carriers A, B, C and D (etcetera), and operations f and g (etcetera).

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

© Dines Bjørner and © Springer
Fredsvej 11 Tiergartenstraße 17
DK-2840 Holte D 69121 Heidelberg
Denmark Germany

An Example Stack Algebra

Example 8.33 *Stack Algebra:* We bring a third version of the stack algebra of Examples 8.29(1) and 8.31.

- Let us define an algebra of simple stacks.
  - ⋆ E and S are the stack element type, respectively the stack types, i.e., are the *types of interest.* .
  - ⋆ The operations empty and is_empty generate empty stacks, i.e., stacks of no elements, respectively tests whether an arbitrary stack is empty;
  - ⋆ push, top and pop are the *operations of interest.*
  - ⋆ An empty stack is empty.

- ⋆ One cannot pop from an empty stack (i.e., *generate* a remaining stack), nor can one *observe* the top of an empty stack.
- ⋆ *Observing* the top of a stack which is the ["most recent"] result of having pushed the element e "onto" a ["previous"] stack s yields that element e.
- ⋆ *Generating* the stack after a pop of a stack which is the ["most recent"] result of having pushed any element e "onto" the ["previous"] stack s yields that stack s.

Fredsvej 11
DK-2840 Holte
Denmark
Tiergartenstraße 17
D 69121 Heidelberg
Germany

class
  type
    E, S
  value
    empty: **Unit** $\rightarrow$ S
    is_empty: S $\rightarrow$ **Bool**
    push: E $\rightarrow$ S $\rightarrow$ S
    top: S $\stackrel{\sim}{\rightarrow}$ E
    pop: S $\stackrel{\sim}{\rightarrow}$ S
  axiom
    is_empty(empty()),
    top(empty()) $\equiv$ **chaos**,
    pop(empty()) $\equiv$ **chaos**,
    $\forall$ e,e′:E, s:S •
      top(push(e)(s)) $\equiv$ e $\land$
      pop(push(e)(s)) $\equiv$ s
end

The above formalisation should, by now, look rather conventional! ■

© Dines Bjørner Fredsvej 11 DK-2840 Holte Denmark and © Springer Tiergartenstraße 17 D 69121 Heidelberg Germany

Informal Explanation of Some RSL Constructs

- Since this is one of the earlier examples of a full-scale use of several hitherto unexplained, but nevertheless rather simple RSL constructs, let us explain them in anticipation of material of later lectures.
  - ⋆ The RSL *keywords* **class** and **end** delineate the class expression.
  - ⋆ The class expression, in this case, contains three kinds of definitions: **type**, function **value** and **axiom.**
  - ⋆ The **type** definitions you should be familiar with.
  - ⋆ The **value** definitions name a number of values. Here, all these values are functions: one 0-ary (nullary), one 2-ary (binary, dyadic), and three 1-ary (unary, monadic). These function values are given just their type, called their signature (no function definition [body]).

© Dines Bjørner Fredsvej 11 DK-2840 Holte Denmark and © Springer Tiergartenstraße 17 D 69121 Heidelberg Germany

DTU

- ⋆ The **axiom** definitions, that is, the axioms, constrain the function values to lie within a smaller function space than defined by their signatures.
- ⋆ We leave deciphering the specific functionality of these axioms to the reader, but close by explaining the use of the ∀ "binder".
  - ◇ The clause: $\forall$ e,e′:E, s:S $\cdot$ $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$, (where the individual $\mathcal{A}_i$ are the axioms — expressions that may or may not contain the quantifier variables e, e′, and s) expresses that these axioms' variables take values that *range* over the types E, E, and S, respectively.

An Example Queue Algebra

Example 8.34 *Queue Algebra:* We give a formal example of the queue algebra of Example 8.29(2).

- Let us define an algebra of simple queues:
  - ⋆ E and Q are the queue element type, respectively the queue type, i.e., are the *types of interest*.
  - ⋆ The operations empty and is_empty generate empty queues, i.e., queues of no elements, respectively tests whether an arbitrary queue is empty,
  - ⋆ and enq and deq are the *operations of interest.*
  - ⋆ The interesting functions are here defined in terms of the *hidden functions* dq and rq.

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

hide
  dq,rq **in**
class
  type
    E, Q
  value
    empty: **Unit** $\rightarrow$ Q, is_empty: Q $\rightarrow$ **Bool**
    enq: E $\rightarrow$ Q $\rightarrow$ Q, deq: Q $\xrightarrow{\sim}$ (Q $\times$ E)
    dq: Q $\xrightarrow{\sim}$ E, rq: Q $\xrightarrow{\sim}$ Q
  axiom
    is_empty(empty()), deq(empty()) $\equiv$ **chaos**,
    dq(empty()) $\equiv$ **chaos**, rq(empty()) $\equiv$ **chaos**,
    forall e,e′:E, q:Q •
      ∼is_empty(enq(e)(q)),
      dq(enq(e)(empty())) $\equiv$ e,
      rq(enq(e)(empty())) $\equiv$ empty(),
      dq(enq(e)(enq(e′)(q))) $\equiv$ dq(enq(e′)(q)),
      rq(enq(e)(enq(e′)(q))) $\equiv$ enq(e)(rq(enq(e′)(q))),
      deq(enq(e)(q)) $\equiv$ (rq(enq(e)(q)),dq(enq(e)(q)))
end

- Operation **dq** is called an *auxiliary operation*.
- It finds the first element enqueued, i.e., the, “oldest”, or the most distantly, in time, inserted element.
- Auxiliary operation **rq** reconstructs the queue less its currently dequeued element.

■

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

Some Notation: hide

- The functions dq and rq are defined as *hidden functions.*
  - ⋆ They are not intended to be used outside the class expression
  - ⋆ inside which they only serve as *auxiliary functions*, that is, *auxiliary operations*.
- The marker **hide** effects that it can be syntactically checked that they are not used outside the scope of the class definition.
- Hiding values (or types) enable us to reasonably simply characterise, as here, the *functions of interest* deq and enq.

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

Towards Semantic Models of "class" Expressions

- So, a **class** expression, even the little we have so far introduced about class expressions, can be seen to "cluster" the introduction of a number of identifiers, to wit: A, B, C, D, f, g, or E, S, empty, is_empty, push, pop, top, or E, Q, empty, is_empty, deq, enq, dq, and rq.
- *But what does it all mean?*
- We return now to a thread first begun in Sect. .

- Namely to informally explain the semantics of RSL constructs. The "story" applies, inter alia, here.
- As already outlined, in Sect. , the meaning of a class expression is a set of *models*.
- Each model in the set maps all identifiers defined in the class expression, whether hidden or not, into their meaning.
- The meanings of the above-mentioned identifiers, for example, E, S, empty, is_empty, push, pop, and top, are as follows:
  - ⋆ Any type identifier is mapped into the set of values as constrained by the axioms over these values, and
  - ⋆ a function identifier is mapped into a function value, as constrained by the axioms over these function values.

- Since the axioms do not normally constrain the function values to one specific function, but to a (possibly infinite) space of functions over suitable input argument value and result value relations, we have that the meaning of a class expression is a possibly infinite set of models: one for each combination of defined function values, etc.
- We shall later see a need for allowing these models to further map identifiers not (at all) mentioned in the class expression into arbitrary values (including set values).
- The meaning of the stack class expression is thus a set of models, with each model mapping at least the seven identifiers mentioned in the stack class expression into respective meanings: the value type of all elements, of all stacks, and specific values for empty, is-empty, push, pop and top functions.

RSL Syntax for Algebra Specifications

"class" Expressions

- We have several times illustrated the RSL syntax for presenting an algebra in the form of a class of models:

class
  type
    ... [sorts and type definitions] ...
  value
    ... [value, incl. function definitions] ...
  axiom
    ... [properties of types and values (functions) ...]
end

- The meaning of a class expression is a set,
  - ⋆ possibly empty, possibly singleton, possibly infinite,

- ⋆ of models in the form of
- ⋆ bindings
  - ◇ the type and value identifiers introduced in the class expression and
  - ◇ mathematical entities such a numbers, sets, Cartesians and functions.

- We shall only occasionally wrap our type and value definitions and our axioms into a class expression, but in a sense we really ought to so so!
- The intended meaning is of course the same.

"scheme" Declarations

- The scheme construct of RSL allows us to name classes:

scheme A =
  class
  type
    ... [sorts and type definitions] ...
  value
    ... [value, incl. function definitions] ...
  axiom
    ... [properties **of** types and values (functions) ...]
end

- Identifier *A* now names the class of all the models denoted by the class expression.

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

Fredsvej 11 Tiergartenstraße 17
DK-2840 Holte D 69121 Heidelberg
Denmark Germany

Discussion
General

- We have made a *tour de force* of covering, ever so cursorily, some concepts of mathematical algebra.
- The purpose has been twofold.
  - ⋆ First, to put names to a number of algebra concepts which can be used for later characterising a number of specification concepts, principles and techniques.
  - ⋆ Second, we showed notation and elegance of the definitions, something that we, as software engineers, can learn from and ought to copy.

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

- That is, there are so many ideas of specification and of development that can be characterised using these algebraic concepts,
- and knowing this may induce us to further study (especially the universal) algebraic notions.
- Although such a study is outside the aims of these lectures it would reveal the usefulness of the lemmas and theorems of universal algebra.
- We shall endeavour, however, to communicate, wherever relevant, the spirit of the underlying algebraic concepts.

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

- We have finally, in this section on algebra, shown how the software community has taken the *prescribed medicine:*
  - ⋆ The concept of algebra, as a mathematical structure of carriers and operations, has found its way into programming and into specification languages.
  - ⋆ We have shown the initial concepts of the `RSL` class specification construct,
  - ⋆ syntactically as well as semantically.
- In programming languages this algebra concept is usually manifested in so-called *object-orientedness.*
- In specification languages this algebra concept is usually manifested in so-called *module*, *class* or *abstract data type* constructs.

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db

Principles, Techniques and Tools

Principle 8.3 *Algebraic Semantics:* is that of capturing core notions of a domain, or of requirements, or of software designs, by expressing these as algebras. ■

Techniques 3 *Algebra construction* consists in expressing

- (i) the sorts (i.e., abstract types) of the carrier by naming them,
- (ii) the signature of the operations (functions), and
- (iii) in providing an appropriate (small) set of axioms that relate elements of the carrier and the operations

. ■

Dines Bjørner, Fredsvej 11, DK-2840 Holte, Denmark
Springer, Tiergartenstraße 17, D 69121 Heidelberg, Germany

Tool 8.2 *Algebra* tools include

- the **class** and **scheme** constructs of `RSL` (and of similar, basically model-oriented languages (for example: `B`, `event-B`, `VDM++`, and `Object-Z`)),
- `CASL`, the Common Algebraic Specification Language, and
- `CafeOBJ`

. ■

Phone: +45 4221 2141, E-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~db