**PhD Summer School**
**Formal Methods for System Analysis in Informatics**
**Druskininkai, LT, May 2007**

# Abstract Data Types


**Bernd Baumgarten**

*Fraunhofer SIT*
*Rheinstr. 75, 64295 Darmstadt, Germany*
*bernd.baumgarten@sit.fraunhofer.de*
*http://private.sit.fhg.de/~baumgart/*
*+49 6151 869263*

# What is a Data Object?

A **data object** (data item, data element)

… is a **mental** object

(take e.g. the natural number “10”).

… has various **representations**

(e.g. 10, $A_{16}$, $1010_2$, 5+5),

… which in turn may exist in various concrete instances,

e.g. 10 ($\leftarrow$ *here*) and (*there* $\rightarrow$) *10*

What ist the same, what is different?

Often, by the **same** we mean in some sense **equivalent**.

# Data objects in mathematics

In **mathematics**, data objects are often **constructed.**

E.g. in **axiomatic set theory** from atoms, e.g. natural numbers:

$$0 := \varnothing,\ n+1 := n \cup \{n\}.$$

Example: What is 3? – Recursively …

$$\begin{aligned} 3 &= 2 \cup \{2\} = (1 \cup \{1\}) \cup \{1 \cup \{1\}\} \\ &= ((0 \cup \{0\}) \cup \{0 \cup \{0\}\}) \cup \{(0 \cup \{0\}) \cup \{0 \cup \{0\}\}\} \\ &= \{\varnothing, \{\varnothing\}, \{\varnothing,\{\varnothing\}\}\} \end{aligned}$$

Note: Now $\mathbf{P}(3)$[1], or $3 \cap 2$, make sense (mathematically), even if undesired!

In **object-oriented** programming we would probably **disallow** the above non-numerical operations (power set, intersection) on natural numbers: they do not belong to the desired **catalog of operations** on this data type (class).

1) the power set (set of all subsets) of 3

# What is a concrete data type?

In programming, data types
introduce redundancy and
increase understandability and reliability.

A concrete **data type** consists of one or more homogeneous sets (domains) of data objects, and determines the **operations possible** on these domains.

The **elementary**, non-constructed domains are usually predefined by the programming language used.

The **operations** possible on a domain are determined both by its structure and by the elementary domains involved.

# Construction of concrete data types

The definition of a data type often tells how its domain is constructed from simpler domains, e.g. by

- aggregation – forming Cartesian products

  (e.g. array, with implicitly numbered components, or record, with explicitly named components)

- generalization – forming union sets (e.g. case ... of ...);
- recursion/induction – forming infinite domains whose elements are built by repetitions of the same construction(s) involving pointers (e.g. linked list);
- power set formation – (e.g. set of)

# What is/are Abstract Data Types?

**Abstract Data Types** (*plural*): a topic of theoretical computer science.

Texts on ADTs generally agree that **an abstract data type** (*singular*) ...

- deals with the **specification** and the **behaviour** of data objects, with their sorts, functions, and axioms, rather than with their **implementation**;
- does **not distinguish** among “concrete” data types with “similar” properties;
- defines a fixed **interface** to the elements of the domain(s) of the ADT, consisting of a given set of **operations** (functions, methods, ...);
- is connected with **many-sorted algebras** in mathematics.

# What is an Abstract Data Type?

But ADT texts do not agree on what exactly
**an abstract data type is**!

Do not be disappointed –
it does not matter too much here!
At any rate, We will learn what an **abstract data type specification** is.

Important is how **ADT based languages work**.

We will get to know some **typical language features** and their meanings.

We will also draw some links
between data objects in **programming** and in **algebra**.

# Redefining integers – a mini-course on single-sorted ADT's

## Integer operations and constants

Consider the integers

0, 1, -1, 2, -2, …

Data type **Integer** permits **functions** (or **operations**):

successor(–5) = –4, predecessor(7) = 6,

sum(–3,7) = 4, difference(10,3) = 7, product(3,–4) = –12.

Important **constant**: zero = 0.

Funny habit of ADT people:
They consider **constants** as values of **functions without argument**,
e.g. zero( ), then omit the ( ).

# Declaring integers

Typical **declaration** of Integer:

```
Operations:
  s,p             integer            → integer, /unary ops/
  sum,dif,prod    integer, integer   → integer, /binary ops/
  zero                               → integer; /0-ary ops/
```

(don’t mind syntactical details here)

# Signatures, algebras (1)

Un**ary** operations have 1 argument,

bin**ary** operations have 2 arguments, etc. →
number of arguments of an operation: **arity**.

**Signature**: a **type name** + some **function names** + their **arities**

**Algebra** (a single-sorted alg.):

a set + a couple of operations (which have arities).

“Algebra is about algebras.”

# Signatures, algebras (2)

An algebra is **over a given signature** if

its operations are **named** and have their **arities** according to the signature.

Example: **integers** form an algebra over the signature

(integer, {(zero,0), (s,1), (p,1), (sum,2), (dif,2), (prod,2)}).

Algebras,

or more precisely: signatures,

allow us to form **terms**, e.g.

```
sum(p(p(zero)),s(zero))
```

# Signature syntax (general, informal)

```
def   defname
sorts  sortname;
operations:
  const1, const2                      -> sortname, /constants/
  op1,op2, op3   sortname             -> sortname, /unary ops/
  op3,op4        sortname,sortname    -> sortname, /binary ops/
  ... ;
```

“sortname” is **overkill** here,

but later we deal with **various sorts** within one signature.

# Signature examples

known example: integer

```
def         integers
sorts       integer;
operations
  zero                              -> integer; /0-ary ops/
  s,p             integer           -> integer, /unary ops/
  sum,dif,prod    integer, integer  -> integer, /binary ops/
```

new example: group

```
def         group
sorts       elem;
operations
  e:                    -> elem,   /neutral element/
  inv:      elem        -> elem,   /inverse/
  mul:      elem, elem  -> elem;   /group operation/
```

# Algebra examples

Algebras over signature *integers*:

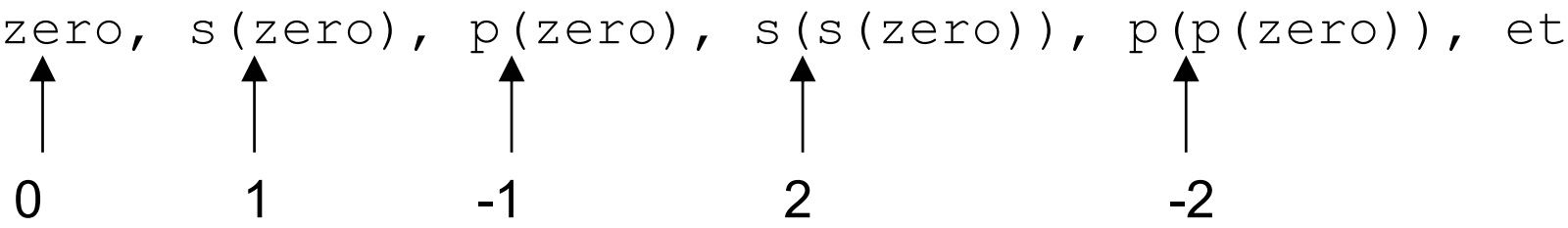
- the **integers** we know
- **vectors** in 3-dim. space $IR^3$,
  s(v) is the opposite vector –v,
  p(v) is v rotated ¼ turn left around the y-axis
  sum(v,w)=v+w, dif(v,w)=v–w, prod(v,w)=v×w
  zero=(0,0,0)


Algebras over signature *group*:

- the **integers** we know with 0, negative, addition
- the **positive rational numbers** with 1, inverse, multiplication
- **movements** (displacement + re-orientation) of a guardsman parading in a plane, chaining steps fwd/back & ¼ turns with
  no-move, reverse, concatenation

# Axioms and ADT specifications

We could use “simple terms” to write all “new integers”:




- What about the other terms, like `sum(p(p(zero)),s(zero))`?

  Of course, for this term we would expect a value of
  –2 + 1, i.e. –1, i.e. `p(zero)`

- How do we know?

  Answer: Let’s add the right **axioms** to the signature!

  A signature with axioms is called an
  **Abstract data type** (**ADT**) **specification**.

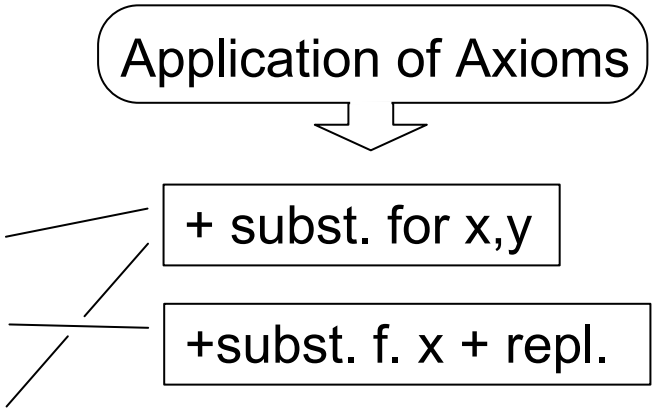
# Integer axioms and their application

```
Axioms: for all x, y in integer:
     s(p(x))=x,                          /Ax 1/
     p(s(x))=x,                          /Ax 2/
     sum(x,zero)=x,                      /Ax 3/
     sum(x,s(y))=s(sum(x,y)),            /Ax 4/
     sum(x,p(y))=p(sum(x,y));            /Ax 5/
```


(+ more for `dif` and `mul` – can you guess them?),

Now we can **calculate** (i.e. **prove** that)

```
  sum(p(p(zero)),s(zero))
= s(sum(p(p(zero)),zero))    /by Ax4/
= s(p(p(zero)))              /by Ax3/
= p(zero)                    /by Ax1/
```



# Equational Logic

**Derivation rules** for Equational Logic

**Equivalence rules**:

Idempotence, symmetry, transitivity of equality

t1 = t1 t1=t2, t2=t3 → t1=t3 t1=t2 → t2=t1

**Substitution rule**: **variables** may be replaced by terms:

If $t,u \in Terms_{\Sigma(V)}$ and $\sigma : V \to Terms_{\Sigma(W)}$, then $(\forall V : t = u) \Rightarrow (\forall W : t\sigma = u\sigma)$.

**Congruence** or **replacement rule**, if in a term **subterms** are replaced by others already proven equal, then the resulting term is equal to the first one.

Definiton of **replacement**,

*term1* —repl. *subt1* $\mapsto$ *subt2*→ *term2*, if

*term1* ←subst. *subt1*$\leftarrow\!\shortmid$*x*— *term0* —subst. *x*$\shortmid\!\rightarrow$*subt2*→ *term2*

# Integer axioms and their application

Using inductive definitions, recursive functions, and equational logic

we can **prove** special and in particular general facts that we previously either

- believed our teachers [1], or
- inferred from many examples [2],

like `sum(x,y)=sum(y,x).`

Try it.
Use induction.
Sorry, it is not totally easy.

---

[1] **Teachers can be wrong.**

[2] **Millions of examples without exception cannot replace proof.**
Take the rule “Things dropped fall down until the the end of 2010, and move up from 2011 on.”
It has been confirmed invariably millions of times! Can you prove it? Do you believe in it?

# Group specification, with axioms

```
def          group

sorts        elem;

operations
  e:                    -> elem,   /neutral element/
  inv:     elem         -> elem,   /inverse/
  mul:     elem, elem   -> elem;   /group operation/

axioms
for all x, y, z in group:
   mul(mul(x,y),z) = mul(x,mul(y,z)), /associativity/
   mul(e,x) = x,                      /left neutral/
   mul(inv(x),x) = e;                 /left inverse/
end of def
```


Exercises: Show that e is also right neutral, inv(x) is also right inverse.

Hint 2: aa'=a''a'aa'=e Hint1:ae=aa'a=a

# Interlude: equivalence relations

R is an **equivalence relation** on a set M if for all x,y,z in M:

| | |
|---|---|
| xRx | reflexive |
| xRy → yRx | symmetric |
| xRyRz → xRz | transitive |

2 Examples on the “old integers”: x=y, e.g. 5=5

x=y mod 5, e.g. 7=-3 mod 5

Example on our new Integer terms:

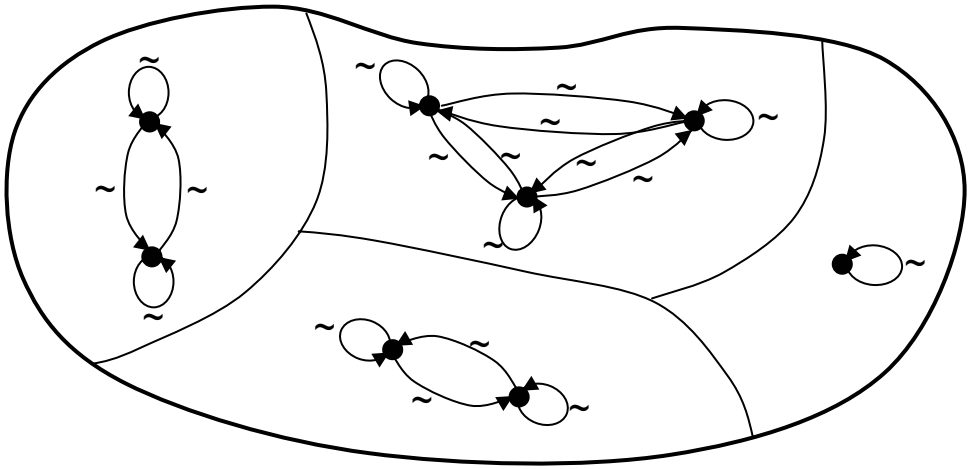
= meaning: **equal due to our axioms**

Example: `sum(p(p(zero)),s(zero)) = p(zero)`

# Equivalence relations and partitions

An equivalence relation ~ induces on M a **partition** into disjoint **equivalence classes**:

- members of **one class** are **~-equivalent**,
- members of **different classes** are **not ~-equivalent.**



Conversely, a **partition** on M into disjoint **classes** induces an equivalence relation ~ with these properties.

# Integers seen as equivalence classes

Imagine equivalence classes «**equal due to axioms**» for integer terms

(infinitely many classes, one for each “old integer”,

each one containg infinitely many terms for the same integer):

| “0” | “1” | “–1” | |
|---|---|---|---|
| zero<br>s(p(zero))<br>sum(zero,zero)<br>p(s(zero))<br>⋮ | s(zero)<br>s(s(p(zero)))<br>sum(s(zero),zero)<br>p(s(s(zero)))<br>⋮ | p(zero)<br>s(p(p(zero)))<br>sum(zero,p(zero))<br>p(s(p(zero)))<br>⋮ | etc. |

**These classes can be used as our new integers!**

# Calculating with equivalence classes

**How to add** 1 and -1:

- Pick *any member* t1 of class “1” and *any member* t2 of class “-1”,
- sum(t1,t2) will *always* be in the same (correct) class 0 **!**

Mathematicians say: the equivalence relation “=” on the Integer terms

is a **congruence relation** for sum

(and so it is for the other operations: s,p,dif,mul).

So now we are sure that 1 + (–1) = 0 !

# What did we do?

- A **signature** –
  e.g. (Integer, {(zero,0), (s,1), (p,1), (sum,2), (dif,2), (prod,2)}) –
  defines a **language of terms** over this signature.
- By **literal application** of operations,
  e.g. sum(“zero”,”s(zero)”) := “sum(zero,s(zero))”,
  this term language provides an algebra (**term algebra**)
  over this signature.
- If we add (equational) **axioms** to the signature,
  e.g. sum(x,zero)=x,
  we obtain an (equational) **ADT specification**.

# What have we achieved?

- The equivalence relation "**equal due to the axioms**" partitions the term algebra into a set of equivalence classes.
- By **application** of operations via
  - *pick any members*
  - *operate on members*
  - *find the equivalence class of the result*,

  this set of equivalence classes provides an algebra

  (**canonical term class algebra**) over this signature

  which even **conforms to the axioms!**
- This way we **re-constructed the integers**.

# Many-sorted signature syntax (general, informal)

```
def   defname
sorts  sorta, sortb, ...;
operations:
  const1, const2                    -> sorta,   /a-constants/
  const3                            -> sortb,   /b-constant/
  ... ,
  op1            sorta              -> sortb,   /a-b-op/
  op2, op3       sorta, sortb       -> sortb,   /ab-b-ops/
  ... ;
```

argument sorts

value sorts

# Many-sorted term construction

… **respects** the typing (**arities**) of operations.

Example:

```
const1, const2                -> sorta,    /a-constants/
const3                        -> sortb,    /b-constant/
op1           sorta           -> sortb,    /a-b-op/
op2, op3      sorta, sortb    -> sortb,    /ab-b-ops/
op4           sortb           -> sorta,    /b-a-ops/
```

then the following is a “valid term”:

```
op4(op2(const1,op1(const2)))
```

invalid (not a term, for several reasons):

```
op4(op4(op2(op1(const1),const2)),const1)
```

**Exercise**: define inductively the **terms over a signature**.

# A many-sorted algebra A over $\Sigma$

**interprets** a many-sorted signature $\Sigma$.

It has

- **carrier** sets $s_A$ interpreting (corresponding to) the **sort names** s,
- **distinguished elements** $c_A$ interpreting the **constants** c (of corresponding sorts),
- **functions** $op_A$ (of corresponding arity) interpreting the **operations** op:

  $\forall op \in Ops, n \in IN_0, s_1,...,s_n, s \in Sorts$ :

  $Arity(op) = (s_1,...,s_n,s) \quad \Rightarrow \quad op_A : s_{1A} \times ... \times s_{nA} \rightarrow s_A$

Note: **single-sorted** is a special case – not the opposite – of **many-sorted**.

# Many-sorted axioms

… use typed variables.

Example signature:

```
const1, const2                -> sorta,    /a-constants/
const3                        -> sortb,    /b-constant/
op1           sorta           -> sortb,    /a-b-op/
op2, op3      sorta, sortb    -> sortb,    /ab-b-ops/
op4           sortb           -> sorta,    /b-a-ops/
```

Axiom set example:

```
for all x in sorta; y in sortb:
   op4(op1(x))          = const1;
   op2(const1,const3) = op1(const2);
```

# Models

A **model** of a many-sorted ADT specification ($\Sigma$,*Ax*)
is a an algebra over the signature $\Sigma$
in which all **axioms** in the axiom system *Ax* hold **true**.

Similarly as in the single-sorted case,
for any many-sorted equational ADT-specification (signature + axioms)
a **canonical term algebra** can be constructed

It fulfils exactly the axioms and their logical consequences.

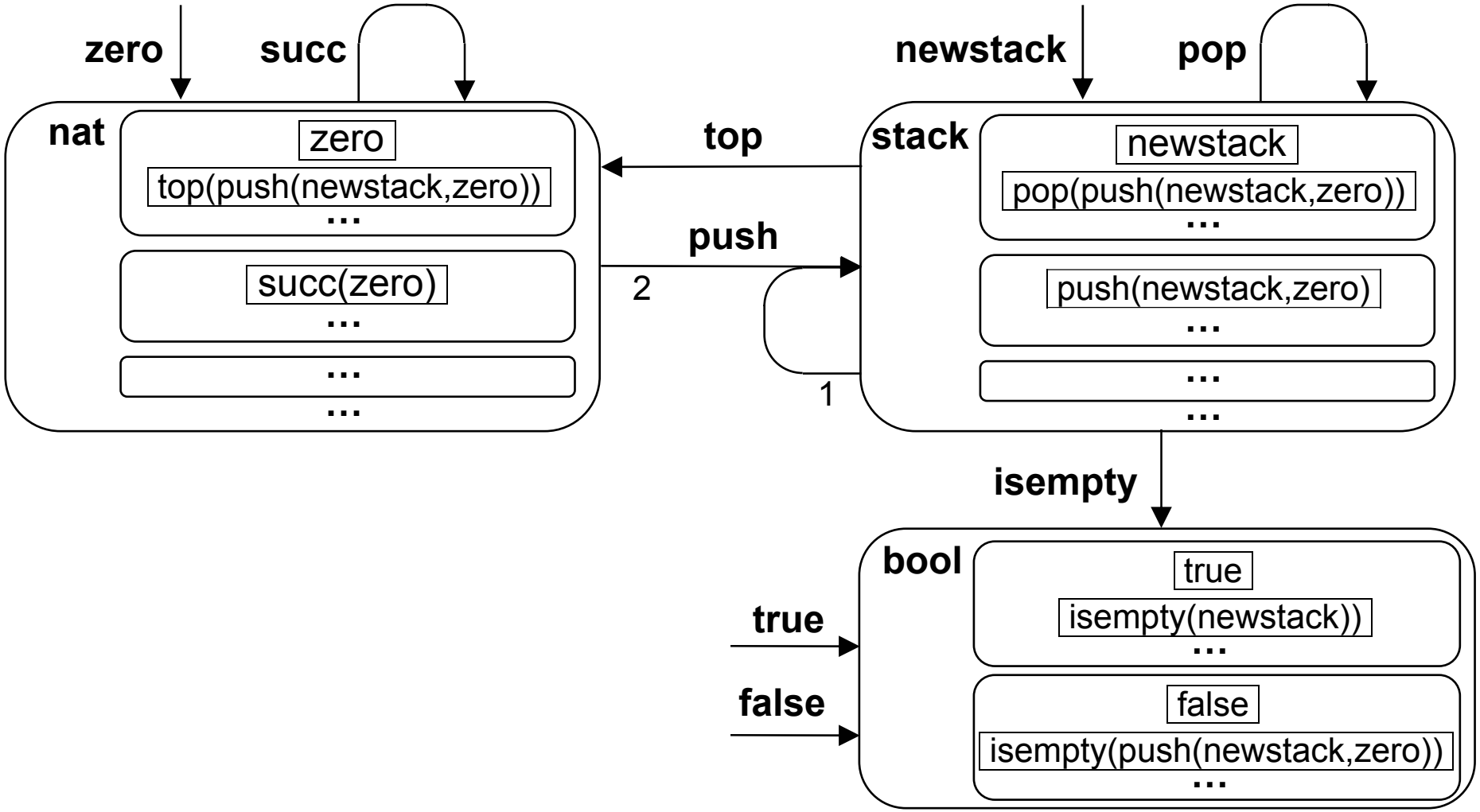
# Example: Stacks of natural numbers

```
def  NatStack
sorts     nat, stack, bool;
operations
      true, false:    -> bool;
      zero:           -> nat;
      succ:           nat -> nat;
      newstack:       -> stack;
      push:           stack, nat -> stack;
      top:        stack -> nat;
      pop:        stack -> stack;
      isempty: stack -> bool;
```

```
axioms
for all s in stack; n in nat:
      isempty(newstack) = true;
      isempty(push(s,n)) = false;
      pop(newstack) = newstack;
      pop(push(s,n)) = s;
      top(newstack) = zero;
      top(push(s,n)) = n;
end of def
```

***pop*** and ***top*** have been made **total**.
A “user” can avoid *top*ping and *pop*ping an empty stack by means of *isempty.*

# Canonical term class algebra for stacks of natural numbers



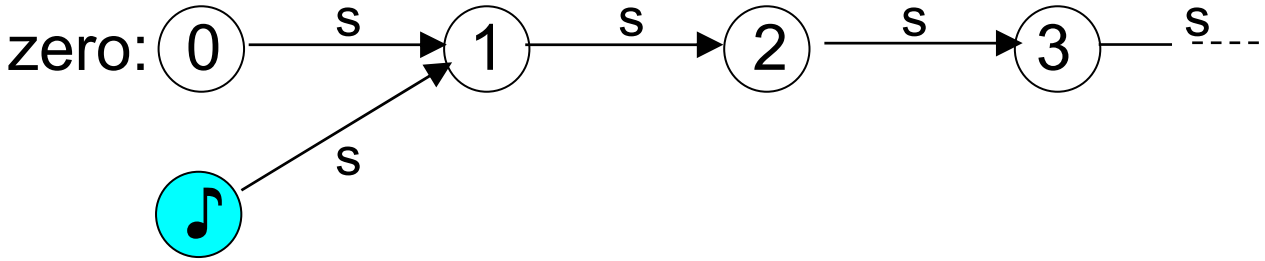
# Junk

An algebra A over a **signature** $\Sigma$

may exhibit a strange, often undesirable, phenomenon:

An element a (of sort s) may not correspond to any term (of sort s)!

Example: funny naturals




Such elements are called **junk**.

If the language for axioms is rich enough, then junk may usually be excluded by means of axioms:

Example: IF s(x)=s(y) THEN x=y.

# Confusion

A **model** of an ADT specification ($\Sigma$,*Ax*)

may exhibit another strange, often undesirable, phenomenon:

> An equation may be generally true,
> even though it is not a consequence of the axioms

Example: the trivial naturals, containing only 0.

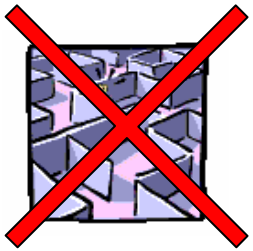
→ for all x in nat: s(x)=x !

This does not follow from the axioms – there are no axioms!

Such an equation is called **confusion**.

# Initiality

Models of ADT specifications
with **neither junk nor confusion**
are called **initial**.



The canonical term algebra quotient
of an equational ADT specification
is initial.

All initial models of an equational ADT specification
are **isomorphic** (practically identical, up to names). **!!**

Exercise:

Study the notions of **homomorphism**,
**isomorphism**, and **initiality** in universal algebra
(and perhaps in category theory, too).

# Observation of ADTs

To use an ADT specification *spec* in practice
we need to know the **behaviour permitted** by it.

A user or tester is confronted with a system *S* of unknown behaviour.

When will *S* be considered **implementing**
(realizing, **conforming to**) *spec*?

Approaches:

- visible / invisible sorts

The user can read the elements of visible sorts as outputs.
Invisible sorts are open to various implementations.

- operations the user can/cannot perform
- operations that do/do not yield visible output

Visible sorts, or ops, will usually have a predetermined fixed interpretation!

# Some other ADT technicalities

- admit properties, relations besides functions
- richness of axiom language (predicate logic?)
- modularity, inheritance, libraries
- partial functions and exception handling
- parameterization
- parallels with and differences from object-oriented programming
- familiar and/or compact notations, e.g. infix, mixfix, symbols

---

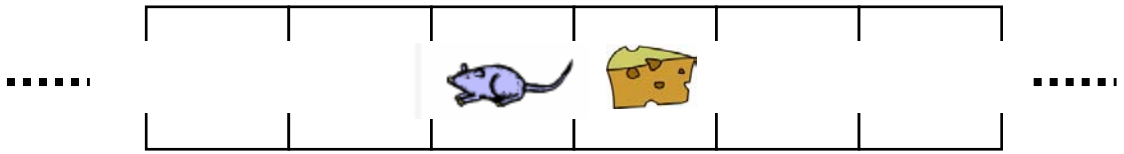
**Recommended exercise**:

Specify algebraically

FIFO queues of natural numbers .

# Another instructive exercise: Mouse and Cheese

A mouse is sitting in one of a straight row of interconnected cells, infinite in both directions. One of the cells contains a piece of cheese. The mouse is always facing either right or left. Originally, in the start configuration:



The mouse can (repeatedly) perform:

fwd: move forward to the next cell in the direction it is facing

turn: turn around by 180°

The mouse can also finally ask a question:

cheese: Am I now in the cheese cell? (Answer: yes or no)

1. Write a signature for this system, with sorts `state` and `answer`.
2. Write true equations, e.g. `cheese(start)=no` and `forall x in start: turn(turn(x))=x`.
3. Derive some of these equations from others.
4. Finite axiom system? If not: why? Adjust the signature? Experiment!

# Specific Languages for/with ADTs

**Clear** [Burstall 1981]

**ACT ONE** [EM, Classen 1988]

**LOTOS** Language of Temporal Ordering Specification [ISO 8807]
an ISO standard in the OSI series, introduced for the specification of, communication protocols, then information processing systems in general
process algebra and abstract data types
textual format, formal semantics

**SDL** Specification and Description Language [EHS,Z.100]
previously an ISO standard and CCiTT Recommendation
introduced for the specification of telecommunication systems
communicating extended finite state machines,
data structures and abstract data types
graphical and textual format, formal semantics under study

\+ **ACT TWO, ASF, ASL, ASL+, CafeOBJ, CASL, CIP-L, COLD, EML, Larch, LOOK, LSL; Maude, METAL, OBJ2/3, OBSCURE, OPAL, PLUSS, RAISE, RSL, SML, SPECTRUM, Z** etc.

# ADT Literature

E. Astesiano, H.-J. Kreowski, B. Krieg-Brückner*: Algebraic Foundations of Systems Specification*. Springer, 1999

H.-D. Ehrich, M. Gogolla, U.W. Lipeck: *Algebraische Spezifikation abstrakter Datentypen*. Teubner, 1989

J. Ellsberger, D. Hogrefe, A. Sarma: *SDL – Formal Oject-oriented Language for Communicating Systems*. Prentice Hall, 1997

H. Ehrig, B. Mahr: *Fundamentals of Algebraic Specification 1*. Springer, 1985

Th. Ihringer: *Allgemeine Algebra*. Teubner, 1993

*LOTOS – A formal description technique based on the temporal ordering of observational behaviour*. ISO International Standard 8807, 1989

J. Loeckxs, H.-D. Ehrich, M. Wolf: *Specification of Abstract Data Types*. Wiley/Teubner, 1996

W. Wechler: *Universal Algebra for Computer Scientists*. Springer, 1992

I. Van Horebeek, J. Lewi: *Algebraic Specifications in Software Engineering*. Springer, 1989

*Specification and Description Language (SDL)*. ITU-T Recomm. Z.100, 1999

**Search** the **web** for "Abstract Data Types", and find course notes, more books, exercises, short overviews, etc.