Algebraic Semantics

Algebraic semantics involves the algebraic specification of data and language constructs.

Foundations based on abstract algebras.

Basic idea

- Name the sorts of objects and the operations on the objects.
- Use algebraic axioms to describe their characteristic properties.

An algebraic specification contains two parts: **signature** and **equations**.

A **signature** Σ of an algebraic specification is a pair <Sorts, Operations> where

- Sorts is a set containing names of sorts.
- Operations is a family of function symbols indexed by the functionalities of the operations represented by the function symbols.

Abstract type whose values are lists of integers:

Sorts = { Integer, Boolean, List }.

Function symbols with their signatures:

zero	: Integer
one	: Integer
plus (_ , _)	: Integer, Integer → Integer
minus (_ , _)	: Integer, Integer → Integer
true	: Boolean
false	: Boolean
emptyList	: List
cons (_ , _)	: Integer, List → List
head (_)	: List → Integer
tail (_)	: List → List
empty? (_)	: List → Boolean
length (_)	: List → Integer

Family of operations decomposes:

$Opr_{Boolean}$ = { true, false }

$Opr_{Integer,Integer \to Integer}$ = { plus, minus }

$Opr_{List \to Integer}$ = { head, length }

Equations constrain the operations to indicate the appropriate behavior for the operations.

head (cons (m, s)) = m,

empty? (emptyList) = true

empty? (cons (m, s)) = false.

Each stands for a closed assertion:

∀m:Integer, ∀s:List [head (cons (m, s)) = m].

empty? (emptyList) = true

∀m:Integer, ∀s:List [empty? (cons (m, s))= false].

Module Representation

- Decompose definitions into relatively small components.
- Import the signature and equations of one module into another.
- Define sorts and functions to be either exported or hidden.
- Modules can be parameterized to define generic abstract data types.

A Module for Truth Values

module Booleans
exports
sorts Boolean
operations
true : Boolean
false : Boolean
errorBoolean : Boolean
not (_) : Boolean → Boolean
and (_ , _) :
Boolean, Boolean → Boolean
or (_ , _) :
Boolean, Boolean → Boolean
implies (_ , _) :
Boolean,Boolean → Boolean
eq? (_ , _) :
Boolean, Boolean → Boolean
end exports
operations
xor (_ , _) : Boolean, Boolean → Boolean

variables
b, b_1, b_2 : Boolean

equations
[B1] and (true, b) = b
[B2] and (false, true) = false
[B3] and (false, false) = false
[B4] not (true)= false
[B5] not (false) = true
[B6] or (b_1, b_2) = not (and (not (b_1), not (b_2)))
[B7] implies (b_1, b_2) = or (not (b_1), b_2)
[B8] xor (b_1, b_2) =
and (or(b_1,b_2),not(and(b_1,b_2)))
[B9] eq? (b_1, b_2) = not (xor (b_1, b_2))
end Booleans

Note module syntax

A **conditional equation** has the form
lhs=rhs
when $lhs_1=rhs_1$, $lhs_2=rhs_2$, ..., $lhs_n=rhs_n$.

A Module for Natural Numbers

module Naturals
imports Booleans
exports
sorts Natural
operations
0 : Natural
1 : Natural
10 : Natural
errorNatural : Natural
succ (_) : Natural → Natural
add (_ , _) : Natural, Natural → Natural
sub (_ , _) : Natural, Natural → Natural
mul (_ , _) : Natural, Natural → Natural
div (_ , _) : Natural, Natural → Natural
eq? (_ , _) : Natural, Natural → Boolean
less? (_ , _) :
Natural, Natural → Boolean
greater?(_ , _) :
Natural, Natural → Boolean
end exports

variables
m, n : Natural

equations
[N1] 1 = succ (0)
[N2] 10 = succ (succ (succ (succ (succ (
succ (succ (succ (succ (succ (0))))))))))
[N3] add (m, 0) = m
[N4] add (m, succ (n)) = succ (add (m, n))
[N5] sub (0, succ(n)) = errorNatural
[N6] sub (m, 0) = m
[N7] sub(succ(m),succ(n)) =sub(m,n)
[N8] mul (m, 0) = 0 *when* m≠errorNatural
[N9] mul (m, 1) = m
[N10] mul (m, succ(n)) = add (m, mul (m, n))
[N11] div (m, 0) = errorNatural
[N12] div (0, succ (n)) = 0 *when* n≠errorNatural
[N13] div (m, succ (n)) =
if (less? (m, succ (n)),
0,
succ(div(sub(m,succ(n)),succ(n))))

[N14] eq? (0, 0) = true
[N15] eq? (0, succ (n)) = false
when n≠errorNatural
[N16] eq? (succ (m), 0) = false
when m≠errorNatural
[N17] eq? (succ (m), succ (n)) = eq? (m, n)
[N18] less? (0, succ (m)) = true
when m≠errorNatural
[N19] less? (m, 0) = false
when m≠errorNatural
[N20] less? (succ (m), succ (n)) = less? (m, n)
[N21] greater? (m, n) = less? (n, m)

end Naturals

All operations propagate errors

succ (errorNatural) = errorNatural,

sub (div(0,0), succ(0)) = errorNatural,

not (errorBoolean) = errorBoolean, and

eq? (0, succ (errorNatural)) = errorBoolean.

Conditions are Necessary

Use [N8] and ignore the condition:

0 = mul(succ(errorNatural),0)
= mul(errorNatural,0)
= errorNatural.

and

succ(0) = succ(errorNatural) = errorNatural,

succ(succ(0)) =
succ(errorNatural) = errorNatural,

and so on.

Conditions are needed when variable(s) on the left disappear on the right.

Constructors

- No equations for 0 and succ
- Terms 0, succ(0), succ(succ(0)), ... not equal
- These plus errorNatural can be viewed as characterizing the natural numbers, the individuals defined by the module.
- Initial algebraic semantics
- No confusion property
- No junk property

A Module for Characters

module Characters
imports Booleans, Naturals

exports
sorts Char
operations
eq? (_ , _) : Char, Char → Boolean
letter? (_) : Char → Boolean
digit? (_) : Char → Boolean
ord (_) : Char → Natural
char-0 : Char
char-1 : Char
: :
char-9 : Char
char-a : Char
: :
char-z : Char
errorChar : Char
end exports

variables
c, c_1, c_2 : Char

equations
[C1] ord (char-0) = 0
[C2] ord (char-1) = succ (ord (char-0))
[C3] ord (char-2) = succ (ord (char-1))
: : :
[C11] ord (char-a) = succ (ord (char-9))
[C12] ord (char-b) = succ (ord (char-a))
: : :
[C36] ord (char-z) = succ (ord (char-y))

[C37] eq? (c_1, c_2) = eq? (ord (c_1), ord (c_2))

[C38] letter? (c) =
and (not (greater? (ord (char-a), ord (c))),
not (greater? (ord (c), ord (char-z))))

[C39] digit? (c) =
and (not (greater? (ord (char-0), ord (c))),
not (greater? (ord (c) ord (char-9))))

end Characters

Parameterized Module and Instantiations

module Lists
imports Booleans, Naturals

parameters Items
sorts Item
operations
errorItem : Item
eq? : Item, Item → Boolean
variables
a, b, c : Item
equations
eq? (a,a) = true *when* a≠errorItem
eq? (a,b) = eq? (b,a)
implies(and(eq?(a,b),eq?(b,c)),
eq?(a,c))=true
when a≠errorItem,
b≠errorItem,
c≠errorItem
end Items

exports
sorts List
operations
null : List
errorList : List
cons (_ , _) : Item, List → List
concat (_ , _) : List, List → List
length (_) : List → Natural
equal? (_ , _): List, List → Boolean
mkList (_) : Item → List
end exports

variables
i, i_1, i_2 : Item
s, s_1, s_2 : List

equations

[S1] concat (null, s) = s

[S2] concat(cons(i,s_1),s_2) =
cons(i,concat(s_1, s_2))

[S3] equal? (null, null) = true

[S4] equal? (null, cons (i, s)) = false
when s≠errorList, i≠errorItem

[S5] equal? (cons (i, s), null) = false
when s≠errorList, i≠errorItem

[S6] equal? (cons (i_1, s_1), cons (i_2, s_2)) =
and(eq?(i_1, i_2), equal?(s_1, s_2))

[S7] length (null) = 0

[S8] length (cons (i, s)) = succ (length (s))
when i≠errorItem

[S9] mkList (i) = cons (i, null)

end Lists

Instantiations

module Files
imports Booleans, Naturals,
instantiation of Lists
bind Items
using Natural **for** Item
using errorNatural **for** errorItem
using eq? **for** eq?
rename **using** File **for** List
using emptyFile **for** null
using mkFile **for** mkList
using errorFile **for** errorList

exports
sorts File
operations
empty? (_) : File → Boolean
end exports

variables f : File

equations
[F1] empty? (f) = equal? (f, emptyFile)

end Files

module Strings

imports Booleans, Naturals, Characters,
instantiation of Lists
bind Items **using** Char **for** Item
using errorChar **for** errorItem
using eq? **for** eq?
rename **using** String **for** List
using nullString **for** null
using mkString **for** mkList
using strEqual **for** equal?
using errorString **for** errorList

exports
sorts String
operations

string-to-natural (_) :
String → Boolean, Natural

end exports

variables
c : Char b : Boolean
n : Natural s : String

equations

[Str1] string-to-natural (nullString) = <true,0>

[Str2] string-to-natural (cons (c, s)) =
if (and (digit? (c), b),
<true, add(mul(sub(ord(c),ord(char-0)),
exp(10, length(s))), n)>,
<false, 0>)
when <b,n> = string-to-natural (s)

end Strings

Expression in [Str2]:

$((\mathrm{ord}(c) - \mathrm{ord}(\text{char-0})) \cdot 10^{\mathrm{length}(s)}) + n$

A Module for Finite Mappings

module Mappings
imports Booleans

parameters Entries
sorts Domain, Range
operations
equals (_ , _) :
Domain,Domain → Boolean
errorDomain : Domain
errorRange : Range
variables
a,b,c : Domain
equations
equals (a,a) = true
when a≠errorDomain
equals (a,b) = equals (b,a)
implies (and (equals (a,b), equals (b,c)),
equals (a,c)) = true
when a, b, and c ≠ errorDomain
end Entries

exports
sorts Mapping
operations
emptyMap : Mapping
errorMapping : Mapping
update(_ , _ , _) :
Mapping,Domain,Range → Mapping
apply (_ , _) :
Mapping, Domain → Range
end exports

variables
m : Mapping
d, d_1, d_2 : Domain
r : Range

equations
[M1] apply (emptyMap, d) = errorRange

[M2] apply (update(m, d_1, r), d_2) = r
when equals(d_1,d_2) = true, m≠errorMapping

[M3] apply (update(m, d_1, r), d_2) = apply(m, d_2)
when equals(d_1,d_2)=false, r≠errorRange

end Mappings

A Store Structure

module Stores
imports Strings, Naturals,
instantiation of Mappings
bind Entries
using String **for** Domain
using Natural **for** Range
using strEqual **for** equals
using errorString **for** errorDomain
using errorNatural **for** errorRange
rename **using** Store **for** Mapping
using emptySto **for** emptyMap
using updateSto **for** update
using applySto **for** apply
end Stores

Mathematical Foundations

Simplify modules.

module Bools
exports
sorts Boolean
operations
true : Boolean
false : Boolean
not (_) : Boolean → Boolean
end exports

equations
[B1] not (true) = false
[B2] not (false) = true
end Bools

module Nats
imports Bools

exports
sorts Natural
operations
0 : Natural
succ (_) : Natural → Natural
add (_ , _) : Natural, Natural → Natural
end exports

variables
m, n : Natural

equations
[N1] add (m, 0) = m
[N2] add (m, succ (n)) = succ (add (m, n))
end Nats

Ground Terms

Function symbols used to construct terms that stand for the objects of the sorts in the signature.

Defn:
For a given signature Σ = <Sorts,Operations>, the set of **ground terms** T_S of sort S is defined inductively:

1. All constants of sort S in Operations are ground terms (in T_S).
2. For every function symbol f : $S_1,\ldots,S_n \to S$ in Operations, if $t_1,\ldots,t_n$ are ground terms of sorts $S_1,\ldots,S_n$, respectively, then $f(t_1,\ldots,t_n)$ is a ground term of sort S where $S_1,\ldots,S_n,S \in$ Sorts.

Example: Ground terms of sort Boolean in Bools

true, not(true),
not(not(true)), not(not(not(true))), ...
false, not(false), not(not(false)), …

Ground terms of sort Natural in Nats:

0, succ(0), succ(succ(0)), …
add(0,0), add(0,succ(0)),
add(succ(0),0), add(succ(0),succ(0)),
add(0,succ(succ(0))),
add(succ(succ(0)),0),
add(0,succ(succ(succ(0)))),
add(succ(succ(succ(0))),0),
add(succ(0),succ(succ(0)),
:

On the basis of the signature only (no equations), the ground terms must be mutually distinct.

Σ-Algebras

Algebraic specifications deal with syntax.

Semantics is provided by defining algebras that serve as models of the specifications.

Heterogeneous or **Many-sorted Algebras:**
A set of operations acting on a collection of sets.

Defn: For a given signature Σ, an algebra $\mathcal{A}$ is a **Σ-algebra** under the following circumstances:

- There is a one-to-one correspondence between the carrier sets of $\mathcal{A}$ and the sorts of Σ
- There is a one-to-one correspondence between the constants and functions of $\mathcal{A}$ and the operation symbols of Σ so that those constants and functions are of the appropriate sorts and functionalities.

Let Σ = <Sorts, Operations> be a signature where

- Sorts is a set of sort names and
- Operations is a set of function symbols of the form f : $S_1, \ldots, S_m \to S_{m+1}$ where each $S_i \in$ Sorts.

A Σ-algebra $\mathcal{A}$ consists of:

1. A collection of sets { $S_\mathcal{A}$ | S∈Sorts }, the **carrier sets**
2. A collection of functions { $f_\mathcal{A}$ | f∈Operations } with the functionality
$f_\mathcal{A} : (S_1)_\mathcal{A}, \ldots, (S_m)_\mathcal{A} \to S_\mathcal{A}$
for each f : $S_1, \ldots, S_m \to S$ in Operations.

Σ-algebras are called heterogeneous or many-sorted algebras because they may contain objects of more than one sort.

Defn: The **term algebra T_Σ** for a signature Σ = <Sorts, Operations> is constructed as follows. Carrier sets { S_{T_Σ} | S∈Sorts } are defined by:

1. For each constant c of sort S in Σ we have a corresponding constant "c" in S_{T_Σ}.
2. For each function symbol f : S_1,...,S_n → S in Σ and any n elements $t_1 \in (S_1)_{T_\Sigma}$, … ,$t_n \in (S_n)_{T_\Sigma}$, the term "f(t_1, ..., t_n)" belongs to the carrier set S_{T_Σ}.

For each function symbol f : S_1,...,S_n → S in Σ and any n elements $t_1 \in (S_1)_{T_\Sigma}$, … ,$t_n \in (S_n)_{T_\Sigma}$, define f_{T_Σ} by $f_{T_\Sigma}(t_1, ..., t_n)$ = "f(t_1, ..., t_n)".

The elements of the carrier sets of T_Σ consist of strings of symbols chosen from a set containing the constants and function symbols of Σ together with the special symbols "(", ")", and ",".

Example

The carrier set for the term algebra T_Σ constructed from the module Bools contains all the ground terms from the signature, including

"true", "not(true)", "not(not(true))", ...

"false", "not(false)", "not(not(false))", ...

The function not_{T_Σ} maps "true" to "not(true)", maps "not(true)" to "not(not(true))", and so forth.

The carrier set is infinite.

Also, "false" ≠ "not(true)"

We have not accounted for the equations and what properties they enforce in an algebra.

Defn: For a signature Σ and a Σ-algebra A, the **evaluation function** $eval_A : T_\Sigma \to A$ from ground terms to values in A is defined as:

$eval_A$("c") = c_A for constants c, and

$eval_A$("f(t_1,…,t_n)") = $f_A(eval_A(t_1), …, eval_A(t_n))$ where each term t_i is of sort S_i for the symbol f : S_1,…,S_m →S in Operations.

A Congruence from the Equations

The function symbols and constants create a set of ground terms.

The equations of a specification generate a congruence ≡ on the ground terms.

A congruence is an equivalence relation with an additional "substitution" property.

Definition: Let Spec = <Σ,E> be a specification with signature Σ and equations E.

The **congruence $\equiv_E$ determined by E on T_Σ** is the smallest relation satisfying the properties:

1. **Variable Assignment**: Given an equation lhs = rhs in E that contains variables v_1,..,v_n and given any ground terms t_1,..,t_n from T_Σ of the same sorts as the respective variables,

   lhs[$v_1 \mapsto t_1$,…,$v_n \mapsto t_n$] $\equiv_E$ rhs[$v_1 \mapsto t_1$,...,$v_n \mapsto t_n$]

   where $v_i \mapsto t_i$ indicates substituting the ground term t_i for the variable v_i.

   If equation is conditional, the condition must be valid after variable assignment is carried out on it.
2. **Reflexive**: For every ground term $t \in T_\Sigma$, $t \equiv_E t$.
3. **Symmetric**: For any ground terms $t_1, t_2 \in T_\Sigma$, $t_1 \equiv_E t_2$ implies $t_2 \equiv_E t_1$.

4. **Transitive**: For any terms $t_1, t_2, t_3 \in T_\Sigma$,
$(t_1 \equiv_E t_2$ and $t_2 \equiv_E t_3)$ implies $t_1 \equiv_E t_3$.

5. **Substitution Property**: If $t_1 \equiv_E t_1', \ldots, t_n \equiv_E t_n'$ and $f : S_1, \ldots, S_n \to S$ is any function symbol in Σ, then $f(t_1, \ldots, t_n) \equiv_E f(t_1', \ldots, t_n')$.

Generate an equivalence relation from equations:

- Take every ground instance of all the equations as a basis.
- Allow any derivation using properties reflexive, symmetric, and transitive and the substitution rule that each function symbol preserves equivalence when building ground terms.

Ground terms for Bools module:

true ≡ not(false) ≡ not(not(true))
≡ not(not(not(false))) ≡ ...

false ≡ not(true) ≡ not(not(false))
≡ not(not(not(true))) ≡ ...

Sample Proof

add(succ(0),succ(0))

≡ succ(add(succ(0),0)) using [N2] and [m ↦ succ(0), n ↦ 0]

≡ succ(succ(0)) using [N1] and [m ↦ succ(0)].

Defn: If Spec = <Σ,E>, a Σ-algebra $\mathbb{A}$ is a **model** of Spec if for all ground terms t_1 and t_2, $t_1 \equiv_E t_2$ implies $eval_A(t_1) = eval_A(t_2)$.

Example: $\mathbb{A}$ = <{ {off,on} }, {off, on, switch}>
where off and on are constants
and switch is defined by
switch(off) = on
switch(on) = off.

Let Σ be the signature of Bools.
A Σ-algebra $\mathbb{A}$:

$Boolean_A$ = {off,on} is the carrier set

Operation of Σ	**Functions of $\mathbb{A}$**
true : Boolean	$true_A$ = on : $Boolean_A$
false : Boolean	$false_A$ = off : $Boolean_A$
not : Boolean → Boolean	not_A = switch : $Boolean_A \to Boolean_A$

For example,
not(true) ≡ false and

$eval_A$("not(true)") = not_A($eval_A$ ("true"))
= $not_A(true_A)$ = switch(on) = off,

and $eval_A$("false") = off.

Construct a particular Σ-algebra, called the **initial algebra**, that is guaranteed to exist, and take it to *be* the meaning of the specification Spec.

Quotient Algebra

Build the **quotient algebra** $\mathbb{Q}$ from the term algebra T_Σ of a specification <S,E> by factoring out congruences.

Defn: Let <Σ,E> be a specification with
Σ = <Sorts, Operations>.

If t is a term in T_Σ, we represent its congruence class as $[t] = \{ t' \mid t \equiv_E t' \}$.

So $[t] = [t']$ if and only if $t \equiv_E t'$.

Carrier sets = $\{ (S)_{T_\Sigma} \mid S \in Sorts \}$.

A constant c becomes congruence class [c].

Functions in the term algebra define functions in the quotient algebra:

Given a function symbol $f : S_1,...,S_n \rightarrow S$ in Σ, $f_Q([t_1],\ldots,[t_n]) = [f(t_1,..,t_n)]$ for any terms $t_i : S_i$, with $1 \leq i \leq n$, from the appropriate carrier sets.

The function f_Q is well-defined:

$t_1 \equiv_E t_1', ..., t_n \equiv_E t_n'$
implies $f_Q(t_1,..,t_n) \equiv_E f_Q(t_1',..,t_n')$
by the Substitution Property for congruences.

For Bools:

$true_Q$ = [true] and $false_Q$ = [false].

The congruence class [true] contains

"true", "not(false)","not(not(true))", ...

The congruence class [false] contains
"false", "not(true)", "not(not(false))", ...

The function not_Q:

not_Q ([false]) = [not(false)] = [true], and

not_Q ([true]) = [not(true)] = [false].

This quotient algebra is an initial algebra for Bools.

Initial algebras are not necessarily unique.

For example, the algebra
$\mathcal{A}$ = <{off, on}, {off, on, switch}>
is also an initial algebra for Bools.

An initial algebra is finest-grained: It equates only those terms required to be equated, and so its carrier sets contain as many elements as possible.

Using this procedure for developing the term algebra and then the quotient algebra, we can always guarantee that at least one initial algebra exists for any specification.

Homomorphisms

Functions between Σ-algebras that preserve the operations are called Σ-homomorphisms.

Used to compare and contrast algebras that act as models of specifications.

Defn: Suppose that $\mathcal{A}$ and $\mathcal{B}$ are Σ-algebras for a given signature Σ = <Sorts, Operations>. h is a **Σ-homomorphism** if it maps carrier sets of $\mathcal{A}$ to carrier sets of $\mathcal{B}$ and constants and functions of $\mathcal{A}$ to constants and functions of $\mathcal{B}$, so that the behavior of constants and functions is preserved.

h consists of a collection { h_S | $S \in$ Sorts } of functions $h_S : S_{\mathcal{A}} \rightarrow S_{\mathcal{B}}$ for $S \in$ Sorts such that

$h_S(c_{\mathcal{A}}) = c_{\mathcal{B}}$ for each constant symbol c : S,

and

$h_S(f_{\mathcal{A}}(a_1,\ldots,a_n)) = f_{\mathcal{B}}(h_{S_1}(a_1),\ldots,h_{S_n}(a_n))$
for each function symbol $f : S_1,...,S_n \rightarrow S$ in Σ and any n elements $a_1 \in (S_1)_{\mathcal{A}},\ldots,a_n \in (S_n)_{\mathcal{A}}$.

h is an **isomorphism**

If h is a Σ-homomorphism from $\mathcal{A}$ to $\mathcal{B}$ and the inverse of h is a Σ-homomorphism from $\mathcal{B}$ to $\mathcal{A}$.

Apart from renaming carrier sets, constants, and functions, the two algebras are exactly the same.

Defn: A Σ-algebra $\mathcal{I}$ in the class of all Σ-algebras serving as models of a specification with signature Σ is called **initial** if for any Σ-algebra $\mathcal{A}$ in the class, there is a unique homomorphism $h : \mathcal{I} \rightarrow \mathcal{A}$.

The quotient algebra Q for a specification is an initial algebra.

For any Σ-algebra A that acts as a model of the specification, there is a unique Σ-homomorphism from Q to A.

The function $\mathrm{eval}_{\mathrm{A}} : T_\Sigma \to \mathrm{A}$ induces a Σ-homomorphism h from Q to A using the definition:

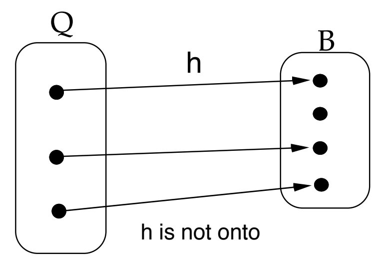
$$h([t]) = \mathrm{eval}_{\mathrm{A}}(t) \text{ for each } t \in T_\Sigma.$$

Any algebra isomorphic to Q is also an initial algebra.

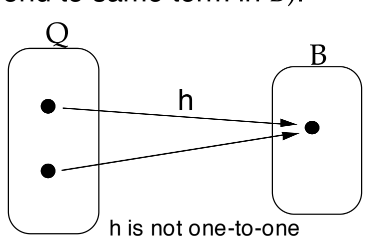
So since the quotient algebra Q and the algebra A = <{off, on}, {off, on, switch}> are isomorphic, A is also an initial algebra for Bools.

Defn: Let <Σ,E> be a specification, let Q be the quotient algebra for <Σ,E>, and let B be an arbitrary model of the specification.

1. If homomorphism from Q to a Σ-algebra B is not onto, then B contains **junk** (values that do not correspond to terms constructed from signature).



2. If homomorphism from Q to B is not one-to-one, then B exhibits **confusion** (two different values in quotient algebra correspond to same term in B).



Example

Consider the quotient algebra for Nats with the infinite carrier set

[0], [succ(0)], [succ(succ(0))], ….

Suppose that we have a 16-bit computer for which the integers consist of the following set of values:

{ -32768, -32767, ..., -1, 0, 1, 2, ..., 32766, 32767 }.

The negative integers are junk with respect to Nats since they cannot be images of any of the natural numbers.

The positive integers above 32767 must be confusion.

When mapping an infinite carrier set onto a finite machine, confusion must occur.

Consistency and Completeness

Suppose we want to add a predecessor operation to naturals by importing Naturals (original version) and defining a predecessor function pred.

module Predecessor_1
  imports Boolean, Naturals

  exports
    operations
      pred (_) : Natural → Natural
  end exports

  variables
    n : Natural

  equations
  [P1] pred (succ (n)) = n
end Predecessor_1

Naturals is a subspecification of Predecessor_1 since the signature and equations of

Predecessor_1 include the signature and equations of Naturals.

The new congruence class [pred(0)] is not congruent to 0 or any of the successors of 0.

We say that [pred(0)] is junk and that Predecessor_1 is not a **complete extension** of Naturals.

We can resolve this problem by adding the equation [P2] pred(0) = 0 (or [P2] pred(0) = errorNatural).

Suppose that we define another predecessor module in the following way:

module Predecessor_2
  imports Boolean, Naturals
  exports
    operations
      pred (_) : Natural → Natural
  end exports
  variables
    n : Natural
  equations
  [P1] pred (n) = sub (n, succ (0))
  [P2] pred (0) = 0
end Predecessor_2

The first equation specifies the predecessor by subtracting one, and the second equation is carried over from the “fix” for Predecessor_1.

In the module Naturals, we have the congruence classes:

[errorNatural], [0], [succ(0)], [succ(succ(0))], ...

With the new module Predecessor_2,

pred(0) = sub(0,succ(0))
  = errorNatural by [P1] and [N5], and

pred(0) = 0 by [P2].

So we have reduced the number of congruence classes, since [0] = [errorNatural].

Because this has introduced confusion, we say that Predecessor_2 is **not a consistent extension** of Naturals.

Defn:

Let Spec be a specification with signature Σ = <Sorts, Operations> and equations E.

Suppose SubSpec is a subspecification of Spec with sorts SubSorts (a subset of Sorts) and equations SubE (a subset of E).

Let T and SubT represent the terms of Sorts and SubSorts, respectively.

- Spec is a **complete extension** of SubSpec if for every sort S in SubSorts and every term t_1 in T, there exists a term t_2 in SubT such that t_1 and t_2 are congruent with respect to E.
- Spec is a **consistent extension** of SubSpec if for every sort subS in SubSorts and all terms t_1 and t_2 in T, t_1 and t_2 are congruent with respect to E if and only if t_1 and t_2 are congruent with respect to SubE.

Using Algebraic Specifications

Data Abstraction

1. **Information Hiding**: Compiler should ensure that the user of an ADT does not have access to the representation (of values) and implementation (of operations) of an ADT.
2. **Encapsulation**: All aspects of specification and implementation of an ADT should be contain in one or two syntactic unit(s) with a well-defined interface to the users of the ADT.

   Examples: Ada package
   Modula module
   Classes in OOP
3. **Generic types** (parameterized modules): A way of defining an ADT as a template without specifying the nature of all its components.

   A generic type is instantiated when the properties of its missing component values are provided.

A Module for Unbounded Queues

Start by giving the signature of a specification of queues of natural numbers.

module Queues
  imports Booleans, Naturals
  exports
    sorts Queue
    operations
      newQ : Queue
      errorQueue : Queue
      addQ (_ , _) : Queue, Natural → Queue
      deleteQ (_) : Queue → Queue
      frontQ (_) : Queue → Natural
      isEmptyQ (_) : Queue → Boolean
  end exports
end Queues

Cannot assume any properties of the operations other than their basic syntax.

This module could be specifying stacks instead of queues.

Properties of Queues

Define the characteristic properties of the queue ADT by describing informally what each operation does, for example:

- The function isEmptyQ(q) returns true if and only if the queue q is empty.
- The function frontQ(q) returns the natural number in the queue that was added earliest without being deleted yet.
- If q is an empty queue, frontQ(q) is an error value.

The descriptions are ambiguous, depending on terms that have not been defined—for example, “empty” and “earliest”.

One may be tempted to define the meaning of the operations in terms of an implementation, but this defeats the whole intent of data abstraction, which is to separate logical properties of data objects from their concrete realization.

A more formal approach to specifying the properties of an ADT is through a set of axioms in the form of module equations that relate the operations to each other.

variables
  q : Queue
  m : Natural

equations

[Q1] isEmptyQ (newQ) = true

[Q2] isEmptyQ (addQ (q,m)) = false
  when q≠errorQueue, m≠errorNatural

[Q3] delete (newQ) = newQ

[Q4] deleteQ (addQ (q,m)) =
  if (isEmptyQ (q),
    newQ, addQ (deleteQ (q),m))
      when m≠errorNatural

[Q5] frontQ (newQ) = errorNatural

[Q6] frontQ (addQ (q,m)) =
  if (isEmptyQ (q), m, frontQ (q))
    when m≠errorNatural

Implementing Queues as Unbounded Arrays

Assuming that the axioms correctly specify the concept of a queue, use them to verify that an implementation is correct.

Realization of an abstract data type:

- a representation of the objects of the type
- implementations of the operations
- representation function Φ that maps terms in the model onto the abstract objects so that the axioms are satisfied.

Plan

Represent queues as arrays with two pointers, one to the front of the queue and one to the end.

A Module for Unbounded Arrays

module Arrays
  imports Booleans, Naturals

  exports
    sorts Array
    operations
      newArray : Array
      errorArray : Array
      assign(_,_,_) : Array,Natural,Natural→Array
      access (_ , _) : Array, Natural → Natural
  end exports

  variables
    arr: Array
    i, j, m : Natural
  equations
    [A1] access (newArray, i) = errorNatural
    [A2] access (assign (arr, i, m), j) =
        if (i = j, m, access (arr, j))
          when m≠errorNatural
end Arrays

Implementation of the ADT Queue using the ADT Array has the following set of triples as its objects:

ArrayQ =
  { <arr,f,e> | arr:Array, f,e:Natural, and f≤e }.

Operations over ArrayQ are defined as follows:

[AQ1] newAQ= <newArray,0,0>

[AQ2] addAQ (<arr,f,e>, m) =
    <assign(arr,e,m),f,e+1>

[AQ3] deleteAQ (<arr,f,e>) =
    if (f = e, <arr,f,e>, <arr,f+1,e>)

[AQ4] frontAQ (<arr,f,e>) =
    if (f = e, errorNatural,
access(arr,f))

[AQ5] isEmptyAQ (<arr,f,e>) = (f = e)
    when arr≠errorArray

Array queues are related to the abstract queues by a homomorphism

Φ : {ArrayQ,Natural,Boolean} →
    {Queue,Natural,Boolean},
defined on the objects and operations of the sorts.

Use symbolic terms “Φ(arr,f,e)” to represent abstract queue objects in Queue.

For <arr,f,e> : ArrayQ, m : Natural,
    and b : Boolean,

Φ (<arr,f,e>) = Φ(arr,f,e) *when* f≤e
Φ (<arr,f,e>) = errorQueue *when* f>e
Φ (m) = m
Φ (b) = b
Φ (newAQ) = newQ
Φ (addAQ) = addQ
Φ (deleteAQ) = deleteQ
Φ (frontAQ) = frontQ
Φ (isEmptyAQ) = isEmptyQ

Under the homomorphism, the five equations that define operations for the array queues map into five equations describing properties of abstract queues.

[D1] newQ = Φ(newArray,0,0)

[D2] addQ (Φ(arr,f,e), m) =
    Φ(assign(arr,e,m),f,e+1)

[D3] deleteQ (Φ(arr,f,e)) =
    if (f = e, Φ(arr,f,e), Φ(arr,f+1,e))

[D4] frontQ (Φ(arr,f,e)) =
    if (f = e, errorNatural, access(arr,f))

[D5] isEmptyQ (Φ(arr,f,e)) = (f = e)

Consider the image of [AQ2] under Φ.

Assume [AQ2]
addAQ (<arr,f,e>,m) =
<assign (arr,e,m),f,e+1>

Then addQ (Φ(arr,f,e),m)
= Φ(addAQ) (Φ(<arr,f,e>),Φ(m)>)
= Φ(addAQ (<arr,f,e>,m))
= Φ(assign(arr,e,m),f,e+1),
which is [D2].

The implementation is correct if its objects can be shown to satisfy the queue axioms [Q1] to [Q6] for arbitrary queues of the form q = Φ(arr,f,e) with f≤e and arbitrary elements m of Natural, given the definitions [D1] to [D5] and the equations for arrays.

Lemma: For any queue Φ(a,f,e) constructed using the operations of the implementation, f≤e.

Proof: The only operations that produce queues are newQ, addQ, and deleteQ, the constructors in the signature. The proof is by induction on the number of applications of these operations.

Basis: Since newQ = Φ(newArray,0,0), f≤e.

Induction Step: Suppose that Φ(a,f,e) has been constructed with n applications of the operations and that f≤e.

Consider a queue constructed with one more application of these functions, for a total of n+1.

Case 1: The n+1st operation is addQ.
But addQ (Φ(a,f,e),m) = Φ(assign (a,f,m),f,e+1) has f≤e+1.

Case 2: The n+1st operation is deleteQ.
But deleteQ (Φ(a,f,e)) =
if (f = e, Φ(arr,f,e), Φ(arr,f+1,e)).
If f=e, then f≤e, and if f<e, then f+1≤e.

The proof is an example of **structural induction,** induction that covers all of the ways in which the objects of the data type may be constructed.

Structural Induction: Suppose f_1, f_2, …, f_n are the operations that act as constructors for an abstract data type S, and P is a property of values of sort S.

If the truth of P for all arguments of sort S for each f_i implies the truth of P for the results of all applications of f_i that satisfy the syntactic specification of S, it follows that P is true of all values of the data type.

The basis case results from those constructors with no arguments.

For the verification of [Q4] as part of proving the validity of this queue implementation, extend Φ for the following values:

For any f : Natural and arr : Array,
Φ(arr,f,f) = newQ.
This extension is consistent with definition [D1].

Verification of Queue Axioms

Let q = Φ(a,f,e) be an arbitrary queue and let m be an arbitrary element of Natural.

[Q1] isEmptyQ (newQ)
= isEmptyQ (Φ(newArray,f,f)) by [D1]
= (f = f) = true by [D5].

[Q2] isEmptyQ (addQ (Φ(arr,f,e),m))
= isEmptyQ (Φ(assign(arr,e,m),f,e+1)
by [D2]
= (f = e+1) = false, since f≤e
by [D5] & lemma.

[Q3] deleteQ (newQ)
= deleteQ (Φ(newArray,f,f)) by [D1]
= Φ(newArray,f,f) = newQ
by [D3] and [D1].

[Q4] deleteQ (addQ (Φ(arr,f,e), m))
= deleteQ (Φ(assign(arr,e,m),f,e+1)) by [D2]
= Φ(assign(arr,e,m),f+1,e+1) by [D4].

Case 1: f = e,
that is, isEmptyQ (Φ(arr,f,e)) = true.
Then Φ(assign(a,e,m),f+1,e+1) = newQ by [D1].

Case 2: f < e,
that is, isEmptyQ (Φ(arr,f,e)) = false.
Then Φ(assign(arr,e,m),f+1,e+1)
= addQ (Φ(arr,f+1,e), m) by [D2]
= addQ (deleteQ (Φ(arr,f,e)), m) by [D3].

[Q5] frontQ (newQ)
= frontQ (Φ(newArray,f,f) by [D1]
= errorNatural since f = f by [D4].

[Q6] frontQ (addQ (Φ(arr,f,e), m))
= frontQ (Φ(assign(arr,e,m),f,e+1)) by [D2]
= access (assign(arr,e,m), f) by [D4].

Case 1: f = e,
that is, isEmptyQ (Φ(arr,f,e)) = true.
Then access (assign(arr,e,m), f)
= access (assign (arr,e,m), e) =m by [A2].

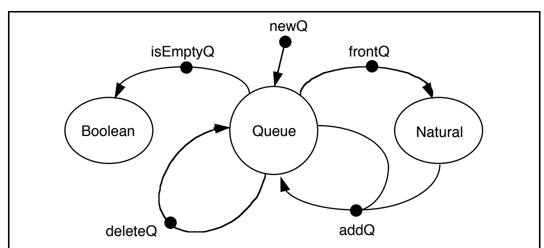
Case 2: f < e,
that is, isEmptyQ (Φ(arr,f,e)) = false.
Then access (assign (arr,e,m), f)
= access (arr,f)
= frontQ (Φ(arr,f,e)) by [A2] and [D4].

Since the six axioms for the unbounded queue ADT have been verified, the implementation via the unbounded arrays is correct.

ADTs As Algebras

Recall that any signature Σ defines a Σ-algebra T_Σ of all the terms over the signature, and that by taking the quotient algebra Q defined by the congruence based on the equations E of a specification, we get an initial algebra that serves as the finest-grained model of a specification <Σ,E>.

Example: An instance of the Queue ADT has operations involving three sorts of objects—namely, Natural, Boolean, and the type being defined, Queue. Some authors designate the type being defined as the **type of interest**. In this context, a graphical notation has been suggested to define the **signature** of the operations of the algebra.



Signature of Queues

The signature of the Queue ADT defines a term algebra T_Σ, sometimes called a **free word algebra**, formed by taking all legal combinations of operations that produce Queue objects.

The values in the sort Queue are those produced by the constructor operations.

Example of terms in T_Σ:
newQ,
addQ (newQ,5), and
deleteQ (addQ (addQ (deleteQ (newQ),9),15)).

The term **free** for such an algebra means that the operations are combined in any way satisfying the syntactic constraints, and that all such terms are distinct objects in the algebra.

The properties of an ADT are given by a set E of equations or axioms that define identities among the terms of T_Σ.

So the Queue ADT is not a free algebra, since the axioms recognize certain terms as being equal.

For example:
deleteQ (newQ) = newQ and
deleteQ(addQ(addQ(deleteQ(newQ),9),15)) = addQ (newQ, 15).

The equations define a congruence $\equiv_E$ on the free algebra of terms as described in section 12.2. That equivalence relation defines a set of equivalence classes that partitions T_Σ.

$$[\ t\ ]_E = \{\ u \in T_\Sigma \mid u \equiv_E t\ \}$$

For example, $[\text{ newQ }]_E = \{$ newQ, deleteQ(newQ), deleteQ(deleteQ(newQ)), … }.

The operations of the ADT can be defined on these equivalence classes before:

For an n-ary operation f∈S
and $t_1,t_2,\ldots,t_n \in T_\Sigma$,
let $f_Q([t_1],[t_2],\ldots,[t_n]) = [f(t_1,t_2,\ldots,t_n)]$.

The resulting (quotient) algebra, also called $T_{\Sigma,E}$, *is* the abstract data type being defined. When manipulating the objects of the (quotient) algebra $T_{\Sigma,E}$ the normal practice is to use representatives from the equivalence classes.

Definition: A **canonical** or **normal form** for the terms in a quotient algebra is a set of distinct representatives, one from each equivalence class.

Lemma: For the Queue ADT $T_{\Sigma,E}$ each term is equivalent to the value newQ or a term of the form
$addQ(addQ(\ldots addQ(addQ(newQ,m_1),m_2),\ldots), m_{n-1}),m_n)$ for some $n \geq 1$
where $m_1,m_2,\ldots,m_n$: Natural.

Proof: The proof is by structural induction.

Basis: The only constant in T_Σ is newQ, which is in normal form.

Induction Step: Consider a queue term t with more than one application of the constructors (newQ, addQ, deleteQ), and assume that any term with fewer applications of the constructors can be put into normal form.

Case 1: t = addQ(q,m) will be in normal form when q, which has fewer constructors than t, is in normal form.

Case 2: Consider t = deleteQ(q) where q is in normal form.

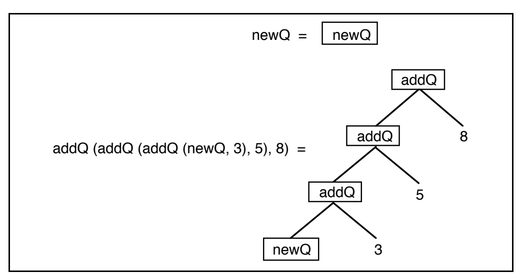
Subcase a: q = newQ. Then deleteQ(q) = newQ is in normal form.

Subcase b: q = addQ(p,m) where p is in normal form.
Then deleteQ(addQ(p,m)) = *if* (
isEmptyQ(p),
newQ,
addQ(deleteQ(p),m))
If p is empty, deleteQ(q) = newQ is in normal form.

If p is not empty, deleteQ(q) = addQ(deleteQ(p),m). Since deleteQ(p) has fewer constructors than t, it can be put into normal form, so that deleteQ(q) is in normal form. ∎

A canonical form for a ADT can be thought of as an “abstract implementation” of the type.

John Guttag [Guttag78b] calls this a **direct implementation** and represents it graphically as shown below.



The canonical form for an ADT provides an effective tool for proving properties about the type.

Lemma: The representation function Φ that implements queues as arrays is an onto function.

Proof: Since any queue can be written as newQ or as addQ(q,m), we need to handle only these two forms.

By [D1], Φ(newArray,0,0) = newQ.
Assume as an induction hypothesis that q = Φ(arr,f,e) for some array.

Then by [D2], Φ(assign(arr,e,m),f,e+1) = addQ (Φ(arr,f,e),m).

Therefore, any queue is the image of some triple under the representation function Φ. ∎

Given an ADT with signature S, operations in S that produce element of the type of interest have already been called **constructors**. Those operations in S whose range is an already defined type of “basic” values are called **selectors**. The operations of S are partitioned into two disjoint sets, Con the set of constructors and Sel the set of selectors. The selectors for Queues are frontQ and isEmptyQ.

Definition: A set of equations for an ADT is **sufficiently complete** if for each ground term $f(t_1,t_2,\ldots,t_n)$ where $f \in Sel$, the set of selectors, there is an element u of a predefined type such that $f(t_1,t_2,\ldots,t_n) \equiv_E u$. This condition means there are sufficient axioms to make the derivation to u.

Theorem: The equations in the module Queues are sufficiently complete.

Proof:

1. Every queue can be written in normal form as newQ or as addQ(q,m).
2. isEmptyQ(newQ) = true,
   isEmptyQ(addQ(q,m)) = false, frontQ(newQ)
   = errorNatural, and frontQ(addQ(q,m))
   = m or frontQ(q) (use induction). ∎

Abstract Syntax and Algebraic Specifications

Points about abstract syntax:

- Only need to specify the meaning of the syntactic forms given by the abstract syntax, since this formalism furnishes all the essential syntactic constructs in the language.
- No harm arises from an ambiguous abstract syntax since its purpose is not syntactic analysis .
- The abstract syntax of a programming language may take many different forms, depending on the semantic techniques that are applied to it.

These points raise questions concerning the nature of abstract syntax and its relation to the language defined by the concrete syntax.

Example: Expressions

Concrete Syntax:

<expr> ::= <term>

<expr> ::= <expr> + <term>

<expr> ::= <expr> - <term>

<term> ::= <element>

<term> ::= <term> * <element>

<element> ::= <identifier>

<element> ::= (<expr>)

Define a signature Σ that corresponds exactly to the BNF definition.

Each nonterminal becomes a sort in Σ, and each production becomes a function symbol whose syntax captures the essence of the production.

The signature of the concrete syntax is given in the module Expressions.

module Expressions
  exports
    sorts Expression, Term, Element, Identifier
    operations
      expr (_) : Term → Expression
      add (_ , _) :
            Expression, Term → Expression
      sub (_ , _) :
            Expression, Term → Expression
      term (_) : Element → Term
      mul (_ , _) : Term, Element → Term
      elem (_) : Identifier → Element
      paren (_) : Expression → Element
  end exports
end Expressions

The terminal symbols in the grammar are "forgotten" in the signature since they are embodied in unique names of the function symbols.

Consider the collection of Σ-algebras following this signature.

The term algebra T_Σ is initial in the collection of all Σ-algebras, meaning that for any Σ-algebra A, there is a unique homomorphism h : $T_\Sigma \rightarrow A$.

The elements of T_Σ are terms constructed using the function symbols in Σ.

Since this signature has no constants, assume a set of constants of sort Identifier and represent them as structures of the form ide(x) containing atoms as the identifiers.

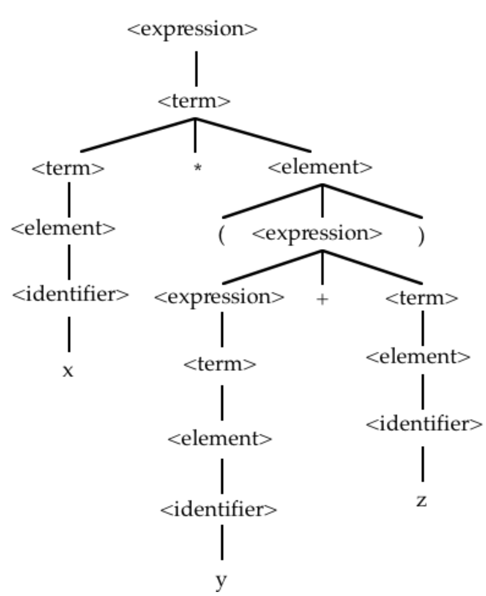
Think of these structures as the tokens produced by a scanner.

The expression "x * (y + z)" corresponds to the following term in T_Σ:

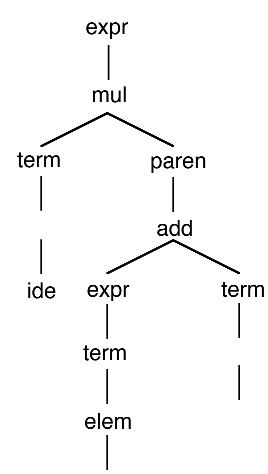
t = expr (mul (term (elem (ide(x))),
      paren (add (expr (term (elem (ide(y)))),
            term (elem (ide(z))))))).

Constructing such a term corresponds to parsing the expression.

Concrete Syntax



Abstract Syntax



The concrete syntax of a programming language coincides with the initial term algebra of a specification with signature Σ.

What does its abstract syntax correspond to?

Consider the following algebraic specification of abstract syntax for the expression language.

module AbstractExpressions
  exports
    sorts AbsExpr, Symbol
    operations
      plus (_ , _) :
        AbsExpr, AbsExpr → AbsExpr
      minus (_ , _) :
        AbsExpr, AbsExpr → AbsExpr
      times (_ , _) :
        AbsExpr, AbsExpr → AbsExpr
      ide (_) : Symbol → AbsExpr
  end exports
end AbstractExpressions

Use set Symbol of symbolic atoms as identifiers.

Construct terms with the constructor function symbols in the AbstractExpressions module to represent the abstract syntax trees.

These freely constructed terms form term algebra A according to signature of AbstractExpressions.

A also serves as a model of the specification in the Expressions module; that is, A is a Σ-algebra:

$\text{Expression}_{\mathrm{A}}$ = Term_{A} = $\text{Element}_{\mathrm{A}}$ = AbsExpr

$\text{Identifier}_{\mathrm{A}}$ = { ide(x) | x : Symbol }.

Operations:

expr_{A} : AbsExpr → AbsExpr
  defined by expr_{A} (e) = e

add_{A} : AbsExpr, AbsExpr → AbsExpr
  defined by add_{A} (e_1,e_2) = plus(e_1,e_2)

sub_{A} : AbsExpr, AbsExpr → AbsExpr
  defined by sub_{A} (e_1,e_2) = minus(e_1,e_2)

term_{A} : AbsExpr → AbsExpr
  defined by term_{A} (e) = e

mul_{A} : AbsExpr, AbsExpr → AbsExpr
  defined by mul_{A} (e_1,e_2) = times(e_1,e_2)

elem_{A} : Identifier → AbsExpr
  defined by elem_{A} (e) = e

$\text{paren}_{\mathrm{A}}$: AbsExpr → AbsExpr
  defined by $\text{paren}_{\mathrm{A}}$ (e) = e

Under this interpretation of the symbols in Σ, this term t becomes a value in the Σ-algebra A:

t_{A} = (expr (mul (term (elem (ide(x))),
  paren (add (expr (term(elem (ide(y)))),
    term (elem (ide(z)))))))))$_{\mathrm{A}}$

= expr_{A} (mul_{A} (term_{A} (elem_{A} (ide(x))),
  $\text{paren}_{\mathrm{A}}$ (add_{A}
    (expr_{A} (term_{A} (elem_{A} (ide(y)))),
      term_{A}(elem_{A} (ide(z)))))))

= expr_{A} (mul_{A} (term_{A} (ide(x)),
  $\text{paren}_{\mathrm{A}}$ (add_{A} (expr_{A} (term_{A} (ide(y))),
    term_{A} (ide(z))))))

= expr_{A} (mul_{A} (ide(x), $\text{paren}_{\mathrm{A}}$
  (add_{A} (expr_{A} (ide(y)),
ide(z)))))

= mul_{A} (ide(x), add_{A} (ide(y), ide(z)))

= times (ide(x), plus (ide(y), ide(z))),

which represents the abstract syntax tree in A that corresponds to the original expression "x * (y + z)".

Each version of abstract syntax is a Σ-algebra for the signature associated with the grammar that forms the concrete syntax of the language.

Any Σ-algebra serving as an abstract syntax is a homomorphic image of T_Σ, the initial algebra for the specification with signature Σ.

Confusion

Generally, Σ-algebras acting as abstract syntax will contain confusion; the homomorphism from T_Σ will not be one-to-one.

This confusion reflects the abstracting process:

By confusing elements in the algebra, we are suppressing details in the syntax.

The expressions "x+y" and "(x+y)", although distinct in the concrete syntax and in T_Σ, are the same when mapped to plus(ide(x),ide(y)) in $\mathbb{A}$.

Any Σ-algebra for the signature resulting from the concrete syntax can serve as the abstract syntax for some semantic specification of the language, but many such algebras will be so confused that the associated semantics will be trivial or absurd.

The task of the semanticist is to choose an appropriate Σ-algebra that captures the organization of the language in such a way that appropriate semantics can be attributed to it.

Algebraic Semantics for Wren

module WrenTypes
  imports Booleans
  exports
    sorts WrenType
    operations
      naturalType, booleanType : WrenType
      programType, errorType : WrenType
      eq?(_ , _) :
          WrenType,WrenType → Boolean
  end exports
  variables
    t_1, t_2 : WrenType
  equations
  [Wt1] eq? (t_1,t_1) = true *when* t_1≠errorType
  [Wt2] eq? (t_1, t_2) = eq? (t_2,t_1)
  [Wt3] eq? (naturalType, booleanType) = false
  [Wt4] eq? (naturalType, programType) = false
  [Wt5] eq? (naturalType, errorType) = false
  [Wt6] eq? (booleanType, programType) = false
  [Wt7] eq? (booleanType, errorType) = false
  [Wt8] eq? (programType, errorType) = false
end WrenTypes

module WrenValues
  imports Booleans, Naturals
  exports
    sorts WrenValue
    operations
      wrenValue (_) : Natural → WrenValue
      wrenValue (_) : Boolean → WrenValue
      errorValue : WrenValue
      eq?(_ , _) :
          WrenValue,WrenValue→Boolean
  end exports
  variables
    x, y : WrenValue
    m, n : Natural
    b, b_1, b_2 : Boolean

  equations
  [Wv1] eq? (x, x) = true *when* x≠errorValue
  [Wv2] eq? (x, y) = eq? (y,x)
  [Wv3] eq? (wrenValue(m), wrenValue(n))
          = eq? (m,n)
  [Wv4] eq? (wrenValue(b_1), wrenValue(b_2))
          = eq? (b_1,b_2)
  [Wv5] eq? (wrenValue(m), wrenValue(b)) = false
          when m≠errorNatural, b≠errorBoolean
  [Wv6] eq? (wrenValue(m), errorValue) = false
          when m ≠ errorNatural
  [Wv7] eq? (wrenValue(b), errorValue) = false
          when b ≠ errorBoolean
end WrenValues

Abstract Syntax for Wren

module WrenASTs
imports Naturals, Strings, WrenTypes

exports
sorts WrenProgram, Block, DecSeq, Declaration, CmdSeq, Cmd, Expr, Ident

operations
astWrenProg (_ , _) : Ident, Block → WrenProg
astBlock (_ , _) : DecSeq, CmdSeq → Block
astDecs (_ , _) : Declaration, DecSeq → DecSeq
astEmptyDecs : DecSeq
astDec (_ , _) : Ident, WrenType → Declaration
astCmds (_ , _) : Cmd, CmdSeq → CmdSeq
astOneCmd (_) : Command → CmdSeq
astRead (_) : Ident → Command
astWrite (_) : Expr → Command
astAssign (_ , _) : Ident, Expr → Command
astSkip : Command
astWhile (_ , _) : Expr, CmdSeq → Command

astIfThen (_ , _) : Expr, CmdSeq → Command
astIfElse(_ , _ , _) : Expr,CmdS,CmdS → Cmd
astAddition (_ , _) : Expr, Expr → Expr
astSubtraction (_ , _) : Expr, Expr → Expr
astMultiplication (_ , _) : Expr, Expr → Expr
astDivision (_ , _) : Expr, Expr → Expr
astEqual (_ , _) : Expr, Expr → Expr
astNotEqual (_ , _) : Expr, Expr → Expr
astLessThan (_ , _) : Expr, Expr → Expr
astLessThanEqual (_ , _) : Expr, Expr → Expr
astGreaterThan (_ , _) : Expr, Expr → Expr
astGreaterThanEqual (_ , _) : Expr, Expr → Expr
astVariable (_) : Ident → Expr
astNaturalConstant (_) : Natural → Expr
astIdent (_) : String → Ident
end exports
end WrenASTs

A Type Checker for Wren

module WrenTypeChecker
imports Booleans, WrenTypes, WrenASTs,
instantiation of Mappings
bind Entries
using String **for** Domain
using WrenType **for** Range
using eq? **for** equals
using errorString **for** errorDomain
using errorType **for** errorRange
rename **using** SymbolTable **for** Mapping
using nullSymTab **for** emptyMap

exports
operations
check (_) : WrenProgram → Bool
check (_ , _) : Block, SymTab → Bool
check (_ , _) :
DecSeq, SymTab → Bool,SymTab
check(_ , _) :
Declaration,SymTab → Bool,SymTab
check (_ , _) : CmdSeq, SymTab → Bool
check (_ , _) : Command, SymTab → Bool
end exports

operations
typeExpr : Expr, SymTab → WrenType

variables
block : Block
decs : DecSeq
dec : Declaration
cmds, $cmds_1$, $cmds_2$: CmdSeq
cmd : Command
expr, $expr_1$, $expr_2$: Expr
type:WrenType
symtab, $symtab_1$: SymbolTable
m : Natural
name : String
b, b_1, b_2 : Boolean

equations

[Tc1]
check (astWrenProgram (astIdent (name), block))
= check(block,
update(nullSymTab,name,progType)

[Tc2]
check (astBlock (decs, cmds), symtab)
= and (b_1,b_2)
when <b_1,$symtab_1$>=check (decs, symtab)
b_2 = check (cmds, $symtab_1$)

[Tc3]
check (astDecs (dec, decs), symtab)
= <and (b_1,b_2), $symtab_2$>
when <b_1,$symtab_1$> = check (dec, symtab)
<b_2,$symtab_2$>=check(decs, $symtab_1$)

[Tc4]
check (astEmptyDecs, symtab)
= <true, symtab>

[Tc5]
check (astDec (astIdent (name), type), symtab)
= *if* (apply (symtab, name) = errorType,
<true, update(symtab, name, type)>,
<false, symtab>)

[Tc6]
check (astCmds (cmd, cmds), symtab)
= and (check (cmd, symtab),
check (cmds, symtab))

[Tc7]
check (astOneCmd (cmd), symtab)
= check (cmd, symtab)

[Tc8]
check (astRead (astIdent (name)), symtab)
= eq?(apply (symtab, name), naturalType)

[Tc9]
check (astWrite (expr, symtab)
= eq? (typeExpr (expr, symtab),
naturalType)

[Tc10]
check(astAssign (astIdent (name), expr), symtab)
= eq? (apply(symtab, name),
typeExpr (expr, symtab))

[Tc11]
check (astSkip, symtab)
= true

[Tc12]
check (astWhile (expr, cmds), symtab)
= *if* (eq? (typeExpr (expr, symtab),
booleanType),
check (cmds, symtab),
false)

[Tc13]
check (astIfThen (expr, cmds), symtab)
= *if* (eq?(typeExpr(expr, symtab), booleanType),
check (cmds, symtab),
false)

[Tc14]
check (astIfElse (expr, $cmds_1$, $cmds_2$), symtab)
= *if* (eq? (typeExpr (expr, symtab), booleanType),
and (check ($cmds_1$, symtab), check ($cmds_2$,
symtab)),
false)

[Tc15]
typeExpr (astAddition ($expr_1$, $expr_2$), symtab)
= *if* (and(eq?(typeExpr($expr_1$,symtab),natType),
eq?(typeExpr ($expr_2$, symtab), natType)),
naturalType,
errorType)

[Tc19]
typeExpr (astEqual ($expr_1$, $expr_2$), symtab)
= *if* (and(eq?(typeExpr($expr_1$,symtab),natType),
eq?(typeExpr($expr_2$,symtab),natType)),
booleanType,
errorType)

[Tc21]
typeExpr (astLessThan ($expr_1$, $expr_2$), symtab)
= *if* (and(eq?(typeExpr($expr_1$,symtab),natType),
eq?(typeExpr($expr_2$,symtab),natType)),
booleanType,
errorType)

[Tc25]
typeExpr (astNaturalConstant (m), symtab)
= naturalType

[Tc26]
typeExpr (astVariable (astIdent(name)), symtab)
= apply (symtab, name)

end WrenTypeChecker

The following equations perform the actual type checking:

[Tc8] The variable in a **read** command has naturalType

[Tc9] The expression in a **write** command has naturalType

[Tc10] The assignment target variable and expression have the same type

[Tc15-18] Arithmetic operations involve expressions of naturalType

[Tc19-24] Comparisons involve expressions of naturalType.

An Interpreter for Wren

module WrenEvaluator
  imports Booleans, Naturals, Strings, Files,
      WrenValues, WrenASTs,
    instantiation of Mappings
      bind Entries
        using String **for** Domain
        using Wren-Value **for** Range
        using eq? **for** equals
        using errorString **for** errDomain
        using errorValue **for** errorRange
      rename
        using Store **for** Mapping
        using emptySto **for** emptyMap
        using updateSto **for** update
        using applySto **for** apply
  exports
  operations
    meaning (_ , _) : WrenProgram, File → File
    perform (_ , _) : Block, File → File
    elaborate (_ , _) : DecSeq, Store → Store
    elaborate (_ , _) : Declaration, Store → Store

    execute (_ , _ , _ , _) :
      CmdSeq, Store, File, File → Store, File, File
    execute (_ , _ , _ , _) :
      Cmd, Store, File, File → Store, File, File
    evaluate (_ , _) : Relation, Store → Boolean
    evaluate (_ , _) : Expr, Store → WrenValue
  end exports

variables
  input, $input_1$, $input_2$: File
  output, $output_1$, $output_2$: File
  block : Block
  decs : DecSeq
  cmds, $cmds_1$, $cmds_2$: CmdSeq
  cmd : Command
  expr, $expr_1$, $expr_2$: Expr
  sto, sto_1, sto_2 : Store
  value : WrenValue
  m,n : Natural
  name : String
  b : Boolean

equations

[Ev1]
meaning(astWrenProgram(astIdent(name),block),input)
    = perform (block, input)

[Ev2]
  perform (astBlock (decs,cmds), input)
    = execute (cmds,
        elaborate(decs,emptySto),
          input, emptyFile)

[Ev3]
  elaborate (astDecs (dec, decs), sto)
    = elaborate (decs,elaborate(dec, sto))

[Ev4]
  elaborate (astEmptyDecs, sto)
    = sto

[Ev5]
elaborate(astDec(astIdent(name),natType), sto)
    = updateSto(sto, name, wrenValue(0))

[Ev6]
elaborate(astDec(astIdent(name),booleanType),sto)
    = updateSto(sto, name, wrenValue(false))

[Ev7]
  elaborate (astEmptyDecs, sto)
    = sto

[Ev8]
execute(astCmds(cmd,cmds),sto$_1$, input$_1$, output$_1$)
= execute (cmds, sto$_2$, input$_2$, output$_2$)
when <sto$_2$, input$_2$, output$_2$> =
execute (cmd, sto$_1$, input$_1$, output$_1$)

[Ev9]
execute (astOneCmd (cmd), sto, input, output)
= execute (cmd, sto, input, output)

[Ev10]
execute (astSkip, sto, input, output)
= <sto, input, output>

[Ev11]
execute(astRead(astIdent(name)),sto,input,output)
= *if* (empty? (input),
need error case here
<updateSto(sto,name,first), rest, output>)
when cons(first,rest) = input

[Ev12]
execute (astWrite (expr), sto, input, output)
= <sto,input,
concat(output,mkFile(evaluate(expr,sto)))>

[Ev13]
execute(astAssign(astIdent(name),expr),
sto,input,output)
= <updateSto(sto,name,evaluate(expr,sto)),
input,output>

[Ev14]
execute(astWhile(expr,cmds), sto$_1$, input$_1$, output$_1$)
= *if* (eq? (evaluate (expr, sto$_1$), wrenVal(true))
execute(astWhile(expr,cmds), sto$_2$, input$_2$, output$_2$)
when <sto$_2$, input$_2$, output$_2$> =
execute (cmds, sto$_1$, input$_1$, output$_1$),
<sto, input, output>)

[Ev15]
execute (astIfThen (expr, cmds), sto, input, output)
= *if* (eq? (evaluate (expr, sto), wrenVal(true))
execute (cmds, sto, input, output),
<sto, input, output>)

[Ev16]
execute(astIfElse(expr,cmds$_1$,cmds$_2$),sto,input,output)
= *if* (eq? (evaluate (expr, sto), wrenVal(true))
execute (cmds$_1$, sto, input, output)
execute (cmds$_2$, sto, input, output))

[Ev17]
evaluate (astAddition (expr$_1$, expr$_2$), sto)
= wrenValue(add (m,n))
when wrenValue(m) = evaluate (expr$_1$, sto),
wrenValue(n) = evaluate (expr$_2$, sto

[Ev21]
evaluate (astEqual (expr$_1$, expr$_2$), sto)
= wrenValue(eq? (m,n))
when wrenValue(m) = evaluate (expr$_1$, sto),
wrenValue(n) = evaluate (expr$_2$, sto)

[Ev23]
evaluate (astLessThan (expr$_1$, expr$_2$), sto)
= wrenValue(less? (m,n))
when wrenValue(m) = evaluate (expr$_1$, sto),
wrenValue(n) = evaluate (expr$_2$, sto)

[Ev27]
evaluate (astNaturalConstant (m), sto)
= wrenValue(m)

[Ev28]
evaluate (astVariable (astIdent (name)), sto)
= applySto (sto, name)

end WrenEvaluator

A Wren System

module WrenSystem
imports WrenTypeChecker, WrenEvaluator

exports
operations
runWren : WrenProgram, File → File
end exports

variables
input : File
program : WrenProgram

equations
[Ws1] runWren (program, input)
= *if* (check (program),
eval (program, input),
emptyFile)

-- return an empty file if context violation,
otherwise run program
end WrenSystem

Implementing Algebraic Semantics

We show the implementation of three modules: Booleans, Naturals, and WrenEvaluator.

Expected behavior of the system:

```
>>> Interpreting Wren via Algebraic Semantics <<<
Enter name of source file: frombinary.wren
    program frombinary is
      var sum,n : integer;
    begin
      sum := 0; read n;
      while n<2 do
        sum := 2*sum+n; read n
      end while;
      write sum
    end
Scan successful
Parse successful
Enter an input list followed by a period:
                                        [1,0,1,0,1,1,2].
Output = [43]
yes
```


Module Booleans

```
boolean(true).
boolean(false).

bnot(true, false).
bnot(false, true).

and(true, P, P).
and(false, true, false).
and(false, false, false).

or(false,P,P).
or(true,P,true) :- boolean(P).

xor(P, Q, R) :- or(P,Q,PorQ), and(P,Q,PandQ),
                bnot(PandQ,NotPandQ),
                and(PorQ,NotPandQ, R).

beq(P, Q, R) :- xor(P,Q,PxorQ), bnot(PxorQ,R).
```

Module Naturals

The predicate natural succeeds with arguments of the form

zero, succ(zero), succ(succ(zero)), ….

Calling this predicate with a variable, such as natural(M), generates the natural numbers in this form if repeated solutions are requested by entering semicolons.


```
natural(zero).
natural(succ(M)) :- natural(M).
```

The arithmetic functions follow the algebraic specification closely.

Rather than return an error value for subtraction of a larger number from a smaller number or for division by zero, we print an appropriate error message and abort the program execution.

The comparison operations follow directly from their definitions.

```
add(M, zero, M) :- natural(M).
add(M, succ(N), succ(R)) :- add(M,N,R).

sub(zero, succ(N), R) :-
    write('Fatal Error: Result of subtraction is negative'),
    nl, abort.
sub(M, zero, M) :- natural(M).
sub(succ(M), succ(N), R) :- sub(M,N,R).

mul(M, zero, zero) :- natural(M).
mul(M, succ(zero), M) :- natural(M).
mul(M, succ(succ(N)), R) :-
            mul(M,succ(N),R1), add(M,R1,R).
```



```
div(M, zero, R) :-
    write('Fatal Error: Division by zero'),
    nl, nl, abort.
div(M, succ(N), zero) :- less(M,succ(N),true).
div(M,succ(N),succ(Quotient)) :-
    less(M,succ(N),false),
    sub(M,succ(N),Dividend),
    div(Dividend,succ(N),Quotient).

exp(M, zero, succ(zero)) :- natural(M).
exp(M, succ(N), R) :- exp(M,N,MexpN),
                      mul(M, MexpN, R).

eq(zero,zero,true).
eq(zero,succ(N),false) :- natural(N).
eq(succ(M),zero,false) :- natural(M).
eq(succ(M),succ(N),BoolValue) :-
                eq(M,N,BoolValue).

less(zero,succ(N),true) :- natural(N).
less(M,zero,false) :- natural(M).
less(succ(M),succ(N),BoolValue) :-
                less(M,N,BoolValue).

greater(M,N,BoolValue) :- less(N,M,BoolValue).
```

```
lesseq(M,N,BoolValue) :-
      less(M,N,B1), eq(M,N,B2),
      or(B1,B2,BoolValue).

greatereq(M,N,BoolValue) :-
      greater(M,N,B1), eq(M,N,B2),
      or(B1,B2,BoolValue).
```

Two operations not specified in Naturals module.

toNat converts a numeral to natural notation

toNum converts a natural number to a base-ten numeral.

```
toNat(4,Num) returns
          Num = succ(succ(succ(succ(zero)))).

toNat(0,zero).
toNat(Num, succ(M)) :-
      Num>0, NumMinus1 is Num-1,
      toNat(NumMinus1, M).

toNum(zero,0).
toNum(succ(M),Num) :-
      toNum(M,Num1), Num is Num1+1.
```


Declarations

The clauses for elaborate are used to build a store with numeric variables initialized to zero and Boolean variables initialized to false.

```
elaborate([Dec|Decs],StoIn,StoOut) :-   % Ev3
          elaborate(Dec,StoIn,Sto),
          elaborate(Decs,Sto,StoOut).

elaborate([],Sto,Sto).                  % Ev4

elaborate(dec(integer,[Var]),StoIn,StoOut) :-
    updateSto(StoIn,Var,zero,StoOut).   % Ev5

elaborate(dec(boolean,[Var]),StoIn,StoOut) :-
    updateSto(StoIn,Var,false,StoOut).  % Ev6
```

Commands

For a sequence of commands, the commands following the first command are evaluated with the store produced by the first command

```
execute([Cmd|Cmds],StoIn,InputIn,OutputIn,
        StoOut,InputOut,OutputOut) :-   % Ev8
```



```
execute(Cmd,StoIn,InputIn,OutputIn,
                          Sto,Input,Output),
      execute(Cmds,Sto,Input,Output,
                StoOut,InputOut,OutputOut).

execute([],Sto,Input,Output,Sto,Input,Output).
                                        % Ev9
```

The **read** command removes the first item from the input file, converts it to the natural number notation, and places the result in the store.

```
execute(read(Var),StoIn,emptyFile,Output,
      StoOut,_,Output) :-               % Ev11
      write('Fatal Error: Reading an empty file'),
      nl, abort.
execute(read(Var),[FirstIn|RestIn],Output,
      StoOut,RestIn,Output) :-          % Ev11
      toNat(FirstIn,Value),
          updateSto(StoIn,Var,Value,StoOut).
```


The **write** command evaluates the expression, converts the resulting value from natural number notation to a numeric value, and appends the result to the end of the output file.

```
execute(write(Expr),Sto,Input,OutputIn,
            Sto,Input,OutputOut) :-  % Ev2
        evaluate(Expr,StoIn,ExprValue),
        toNum(ExprValue,Value),
        mkFile(Value,ValueOut),
        concat(OutputIn,ValueOut,OutputOut).
```

Assignment evaluates the expression using the current store and then updates that store to reflect the new binding. The **skip** command makes no changes to the store or to the files.

```
execute(assign(Var,Expr),StoIn,Input,Output,
            StoOut,Input,Output) :-  % Ev13
        evaluate(Expr,StoIn,Value).
        updateSto(StoIn,Var,Value,StoOut).

execute(skip,Sto,Input,Output,Sto,Input,Output).
                                     % Ev10
```

Two forms of **if** test Boolean expressions and let a predicate "select" perform actions.

```
execute(if(Expr,Cmds),StoIn,InputIn,OutputIn,
        StoOut,InputOut,OutputOut) :-
  evaluate(Expr,StoIn,BoolVal),      % Ev15
  select(BoolVal,Cmds, [ ],
        StoIn,InputIn,OutputIn,
        StoOut,InputOut,OutputOut).

execute(if(Expr,Cmds1,Cmds2),StoIn,InputIn,
        OutputIn,StoOut,InputOut,OutputOut) :-
  evaluate(Expr,StoIn,BoolVal),      % Ev16
  select(BoolVal,Cmds1,Cmds2,
        StoIn,InputIn,OutputIn,
        StoOut,InputOut,OutputOut).

select(true,Cmds1,Cmds2,
        StoIn,InputIn,OutputIn,
        StoOut,InputOut,OutputOut) :-
  execute(Cmds1,StoIn,InputIn,OutputIn,
        StoOut,InputOut,OutputOut).

select(false,Cmds1,Cmds2,
        StoIn,InputIn,OutputIn,
        StoOut,InputOut,OutputOut) :-
  execute(Cmds2,StoIn,InputIn,OutputIn,
        StoOut,InputOut,OutputOut).
```


If the comparison in the **while** command is false, the store and files are returned unchanged.

If the comparison is true, the **while** command is reevaluated with the store and files resulting from the execution of the while loop body.

```
execute(while(Expr,Cmds),
        StoIn,InputIn,OutputIn,
        StoOut,InputOut,OutputOut) :-
  evaluate(Expr,StoIn,BoolVal),      % Ev14
  iterate(BoolVal,Expr,Cmds,
        StoIn,InputIn,OutputIn,
        StoOut,InputOut,OutputOut).

iterate(false,Expr,Cmds,
        Sto,Input,Output,Sto,Input,Output).

iterate(true,Expr,Cmds,
        StoIn,InputIn,OutputIn,
        StoOut,InputOut,OutputOut) :-
  execute(Cmds,StoIn,InputIn,OutputIn,
        Sto,Input,Output),
  execute(while(Expr,Cmds),
        Sto,Input,Output,
        StoOut,InputOut,OutputOut).
```


Expressions

The evaluation of arithmetic expressions is straightforward.

Evaluating a variable involves looking up the value in the store.

A numeric constant is converted to natural number notation and returned.

```
evaluate(exp(plus,Expr1,Expr2),Sto,Result) :-
        evaluate(Expr1,Sto,Val1),  % Ev17
        evaluate(Expr2,Sto,Val2),
        add(Val1,Val2,Result).

evaluate(num(Constant),Sto,Value) :-
        toNat(Constant,Value).     %Ev27

evaluate(ide(Var),Sto,Value) :-
        applySto(Sto,Var,Value).   % Ev28
```


Evaluation of comparisons is similar to arithmetic expressions; the equal comparison is given below, and the five others are left as an exercise.

```
evaluate(exp(equal,Expr1,Expr2),Sto,Bool) :-
        evaluate(Expr1,Sto,Val1),      % Ev21
        evaluate(Expr2,Sto,Val2),
        eq(Val1,Val2,Bool).
```

Prolog implementation of algebraic semantics is similar to the denotational interpreter with respect to command and expression evaluation.

Biggest difference:

- Ignore native arithmetic in Prolog
- Naturals module performs arithmetic based solely on a number system derived from applying a successor operation to an initial value zero.