ABSTRACT DATA TYPES AND ALGEBRAIC SPECIFICATIONS

Matthieu Tixier – #6

Outline

- Algebraic specifications
  - Motivation
  - Define the relations between operations
- Abstract data type specification (« a sort of »)
  - Structure
    - Name, description
    - References
    - Preconditions
    - Operations
    - Axioms
  - Completeness and soundness (Complétude et consistance)
  - Examples
- Discussion

Motivation

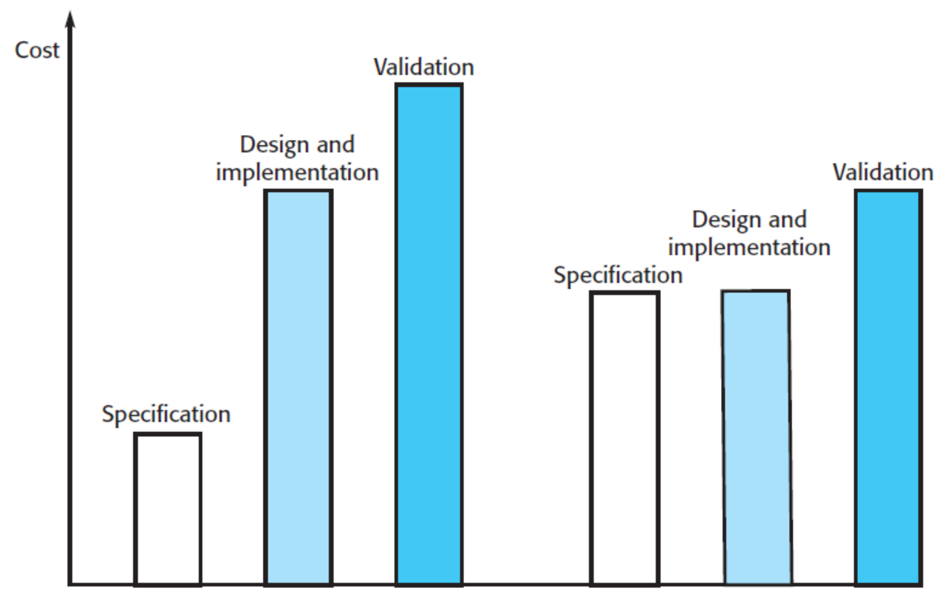
- A « complete » specification « without ambiguïty »
  - Highly detailed level
  - Mathematical formalism

➔ Critical system or critical part(s) of system

- Benefits
  - Prepare implementation and validation (tests)
- Limits

  Takes time to define

  Adoption



Amount of resources spent in software engineering activities without and with ADT [Sommerville, 2009]

Algebra

- The study of the relations between mathematical objects
  - Commutativity : $x + y = y + x$ (communtativité)
  - Identity elements : $a + e = e + a = a$ (élément neutre)
  - Inverse elements : *For all* a, *there exists* b *so that* $a + b = b + a = e$
  - Transitivity : *if* $x < y$ *and* $y < z$ *then* $x < z$ (transitivité)
    [...]

- Algebraic structures
  - Group : with an associative binary operation (ie, +, fr : loi de composition), an inverse function and an identity element.
  - Monoïd : with only a binary operation (of special relevance in formal langage with the use of the concatenation operation)

- Algebraic specification
  - « Define what you want without explaining how »

Links with formal grammar

- CFG, ABNF
  - Syntax level
    - Rules for combining the symbols of a language
    - No interpretation or semantic consideration

```
Timecode        =        2DIGIT ":" 2DIGIT ":" 2DIGIT "," 3DIGIT

; 00:03:90,000 is right from a syntax viewpoint
; but wrong from a semantic viewpoint
```

Links with formal grammar

- Algebraic specification – Abstract Data Type (ADT)
  - Semantic level
    - The mathemathical meaning of operations
    - Focus on the relations between types to define the meaning

Title : Definition of a TimeCode type to represent time span of 24h maximum

Description : As long as the related TimeCode values are positive and strictly below 24h, describe a TimeCode as a 3-tuple (hours, minutes, seconds) that matches the usual way of counting time with a 24h time scale (base 60). A TimeCode can be converted into an integer as the sum of the time span seconds (and conversely).

Definition excerpt […]

IntToTimeCode(n) = **IF** n / 3600 < 23,
    TimeCode(n / 3600, (n % 3600) / 60, (n % 3600) % 60)
    ELSE exception "n is too big"

TimeCodeToInt(Create) = 0

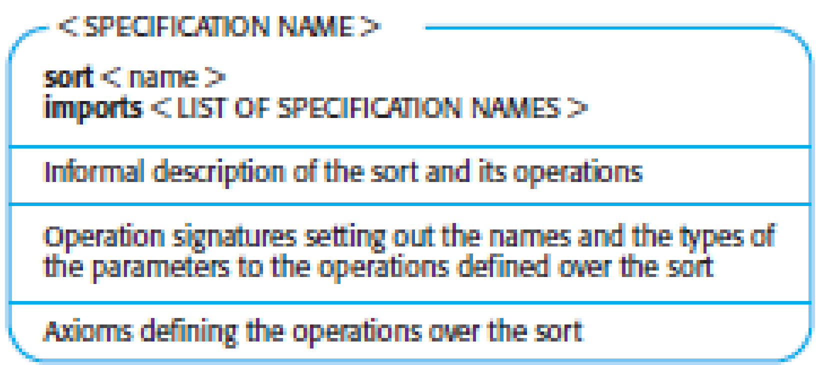
TimeCodeToInt(IntToTimeCode(n)) = n

IsBiggerThan(tc1, Create) = FAUX

IsBiggerThan(IntToTimeCode(n), IntToTimeCode(m)) = TRUE, **IF** n >= m **ELSE** FALSE

ADT Format

- Specification title
- The « sort » name (abstract type)
- References to other sort
- A full text description of the operations
- Preconditions (optional)
- Operations signatures
- Axioms



Example - List

Title : LIST(E)
Sort : List
References : Integer

- « Sort » (abstract type)
  - The type symbol
  - Propose a generic definition of a type through the relations between its operations
  - The relations hold independently from possible implementations

- References to other ADT
  - Generic or primitive types
    - Integer, Float, String, Date …

Example - List

Title : LIST(E)
Sort : List
References : Integer

Description :
Defines a list where elements are added at the end and removed from the front. The operations are Create, which brings an empty list into existence, Cons, which creates a new list with an added member, Length, which evaluates the list size, Head, which evaluates the front element of the list, and Tail, which creates a list by removing the head from its input list. Undefined represents an undefined value of type E.

- Indicate preconditions when needed
  - Boundaries or range of possible values
  - Constants or special values (ie, undefined, NULL)

Example - List

- Operations signatures
  - Define the value domain for the operation argument(s)
  - Also the value domain for the result of operation

Create : → List
Cons : List x E → List
Head : List → E
Length : List → Integer
Tail : List → List

  - Illustration : Cons(L, v)
    - Cons take 2 arguments, a list « L » of type List and an element « v » of type E.
    - Cons return a List

➔ Describe the interface of the abstract type

Example - List

- Axioms
  - Express contraints over the operations **results**
    - The « what » rather than the « how »
  - On the basis of the primitive data types that compound the sort
  - Several formalisms can be relevant:
    - Boolean : propositional logic, first order logic
    - Integer : arithmetics and algebra
    - Set theory
    - …

Example - List

- Axioms
  - A systematic description of the relations that hold between the operations
  - Constructor operation(s)
    - Create the abstract type
  - Inspection operation(s)
    - Allow to get information about the defined sort

Example - List

- Axioms
  - A systematic description of the relations that hold between the operations
  - Constructor operation(s)
    - Primary constructor : create the sort (not directly defined)
    - Secondary constructor : modify the sort and can be defined with primary constructor(s) (ie, Tail)
  - Inspection operation(s)
    - Allow to get information about the defined sort

Create : → List
Cons : List x E → List
Head : List → E
Length : List → Integer
Tail : List → List

Example - List

- Axioms (Example for LIST)

Head(Create) = Undefined (error empty list)
Head(Cons(L, v)) = **if** L = Create **then** v **else** Head(L)

Example - List

- Axioms (Example for LIST)

Head(Create) = Undefined (error empty list)
Head(Cons(L, v)) = **if** L = Create **then** v **else** Head(L)

Length(Create) = 0
Length(Cons(L, v)) = Length(L) + 1

Example - List

- Axioms (Example for LIST)

Head(Create) = Undefined (error empty list)
Head(Cons(L, v)) = **if** L = Create **then** v **else** Head(L)

Length(Create) = 0
Length(Cons(L, v)) = Length(L) + 1

Tail(Create) = Create
Tail(Cons(L, v)) = **if** L = Create **then** Create **else** Cons(Tail(L), v)

Example - List

- Axioms (Example for LIST)

Head(Create) = Undefined (error empty list)
Head(Cons(L, v)) = **if** L = Create **then** v **else** Head(L)

Length(Create) = 0
Length(Cons(L, v)) = Length(L) + 1

Tail(Create) = Create
Tail(Cons(L, v)) = **if** L = Create **then** Create **else** Cons(Tail(L), v)

➔ Show how the axioms for the Length, allows us to compute the number of element in the following list example : [3,7,8]

Example - List

- Use of recursion (e.g. Length)

Length(Create) = 0
Length(Cons(L, v)) = Length(L) + 1

Ex : Let [3,7,8]

Length([3,7,8])
= Length(Cons([3,7], 8))
= Length([3,7]) + 1

Example - List

- Use of recursion (e.g. Length)

```
Length(Create) = 0
Length(Cons(L, v)) = Length(L) + 1
```

```
Ex : Let [3,7,8]

Length([3,7,8])
= Length(Cons([3,7], 8))
= Length([3,7]) + 1
= Length(Cons([3], 7)) + 1
= Length([3]) + 1 + 1
```

Example - List

- Use of recursion (e.g. Length)

```
Length(Create) = 0
Length(Cons(L, v)) = Length(L) + 1
```

```
Ex : Let [3,7,8]

Length([3,7,8])
= Length(Cons([3,7], 8))
= Length([3,7]) + 1
= Length(Cons([3], 7)) + 1
= Length([3]) + 1 + 1
= Length(Cons([], 3)) + 1 + 1
= Length([]) + 1 + 1 + 1
```

Example - List

- Use of recursion (e.g. Length)

```
Length(Create) = 0
Length(Cons(L, v)) = Length(L) + 1
```

```
Ex : Let [3,7,8]

Length([3,7,8])
= Length(Cons([3,7], 8))
= Length([3,7]) + 1
= Length(Cons([3], 7)) + 1
= Length([3]) + 1 + 1
= Length(Cons([], 3)) + 1 + 1
= Length([]) + 1 + 1 + 1
= 0 + 1 + 1 + 1
= 3
```

Example - List

- Use of recursion (e.g. Tail)

Tail(Create) = Create
Tail(Cons(L, v)) = **if** L = Create **then** Create **else** Cons(Tail(L), v)

Ex : Let [3,7,8]

Tail([3,7,8])
= Tail(Cons([3,7], 8))
= Cons(Tail([3,7]), 8)

Example - List

- Use of recursion (e.g. Tail)

Tail(Create) = Create
Tail(Cons(L, v)) = **if** L = Create **then** Create **else** Cons(Tail(L), v)

Ex : Let [3,7,8]

Tail([3,7,8])
= Tail(Cons([3,7], 8))
= Cons(Tail([3,7]), 8)
= Cons(Tail(Cons([3], 7)), 8)

Example - List

- Use of recursion (e.g. Tail)

Tail(Create) = Create
Tail(Cons(L, v)) = **if** L = Create **then** Create **else** Cons(Tail(L), v)

Ex : Let [3,7,8]

Tail([3,7,8])
= Tail(Cons([3,7], 8))
= Cons(Tail([3,7]), 8)
= Cons(Tail(Cons([3], 7)), 8)
= Cons(Cons(Tail([3]), 7), 8)
= Cons(Cons(Tail(Cons([],3)), 7), 8)

Example - List

- Use of recursion (e.g. Tail)

Tail(Create) = Create
Tail(Cons(L, v)) = **if** L = Create **then** Create **else** Cons(Tail(L), v)

Ex : Let [3,7,8]

Tail([3,7,8])
= Tail(Cons([3,7], 8))
= Cons(Tail([3,7]), 8)
= Cons(Tail(Cons([3], 7)), 8)
= Cons(Cons(Tail([3]), 7), 8)
= Cons(Cons(Tail(Cons([],3)), 7), 8)
** if L = Create then Create
= Cons(Cons(Create, 7), 8)

Example - List

- Use of recursion (e.g. Tail)

Tail(Create) = Create
Tail(Cons(L, v)) = **if** L = Create **then** Create **else** Cons(Tail(L), v)

Ex : Let [3,7,8]

Tail([3,7,8])
= Tail(Cons([3,7], 8))
= Cons(Tail([3,7]), 8)
= Cons(Tail(Cons([3], 7)), 8)
= Cons(Cons(Tail([3]), 7), 8)
= Cons(Cons(Tail(Cons([],3)), 7), 8)
** if L = Create then Create
= Cons(Cons(Create, 7), 8)
= Cons([7], 8)
= [7,8]

Example - List

Title : LIST(E)
Sort : List
References : Integer

Description :
Defines a list where elements are added at the end and removed from the front. The operations are Create, which brings an empty list into existence, Cons, which creates a new list with an added member, Length, which evaluates the list size, Head, which evaluates the front element of the list, and Tail, which creates a list by removing the head from its input list. Undefined represents an undefined value of type E.

Create : → List
Cons : List x E → List
Head : List → E
Length : List → Integer
Tail : List → List

Head(Create) = Undefined (error empty list)
Head(Cons(L, v)) = **if** L = Create **then** v **else** Head(L)
Length(Create) = 0
Length(Cons(L, v)) = Length(L) + 1
Tail(Create) = Create
Tail(Cons(L, v)) = **if** L = Create **then** Create **else** Cons(Tail(L), v)

Exemple - Liste

Titre : LIST(E)
Sorte : List
Références : Integer

Description :
Défini une liste où les éléments sont ajoutés à la fin de la liste et retirés par la tête. Les opérations sont : Create, qui construit une liste vide, Cons, qui crée une liste avec un élément, Length, qui évalue le nombre d'éléments de la liste, Head, qui évalue l'élément de tête de la liste et Tail, qui créé une nouvelle liste en retirant l'élément de tête de liste fournit en argument. On prend la valeur Undefined qui représente une valeur non définie du type E.

Create : → List
Cons : List x E → List
Head : List → E
Length : List → Integer
Tail : List → List

Head(Create) = Undefined (error empty list)
Head(Cons(L, v)) = **if** L = Create **then** v **else** Head(L)
Length(Create) = 0
Length(Cons(L, v)) = Length(L) + 1
Tail(Create) = Create
Tail(Cons(L, v)) = **if** L = Create **then** Create **else** Cons(Tail(L), v)

Abstraction

Title : ABC(I)
Sort : Abc
References : Integer

$C : \rightarrow Abc$
$B : Abc \times I \rightarrow Abc$
$H : Abc \rightarrow I$
$L : Abc \rightarrow Integer$
$T : Abc \rightarrow Abc$

$H(C) = Undefined$
$H(B(X, i)) =$ **if** $X = C$ **then** i **else** $H(X)$
$L(C) = 0$
$L(B(X, i)) = L(X) + 1$
$T(C) = C$
$T(C(X, i)) =$ **if** $X = C$ **then** C **else** $B(T(X), i)$

→ The relations between the operations define the type semantic

Exemple – TimeCode

Title : Definition of a TimeCode type to represent time span of 24h maximum
Sort : TimeCode
References : Int, Bool

Description :
As long as the related TimeCode values are positive and strictly below 24h, describe a TimeCode as a 3-tuple (hours, minutes, seconds) that matches the usual way of counting time with a 24h time scale (base 60). A TimeCode can be converted into an integer as the sum of the time span seconds (and conversely). The operations for addition and substraction are available as well as a test for greater than or equal value.

----------------------------------- SIGNATURES ---------------------------------

Create : → TimeCode // Primary constructor
TimeCode : Int x Int x Int → TimeCode // Primary constructor
IntVersTimeCode : Int → TimeCode // Primary constructor
TimeCodeVersInt : TimeCode→ Int // inspection
GreaterThanEq : TimeCode x TimeCode → Bool // inspection
Addition : TimeCode x TimeCode → TimeCode // Secondary constructor
Substraction : TimeCode x TimeCode → TimeCode // Secondary constructor

------------------------------------ AXIOMES --

```
// % : modulo, the remainder after integer division
IntVersTimeCode(n) = IF n / 3600 < 23,
                        TimeCode(n / 3600, (n % 3600) / 60, (n % 3600) % 60)
                        ELSE exception "n is too big"
```

Exemple – TimeCode


```
------------------------------------- AXIOMES ---------------------------------------------------------
// % : modulo, the remainder after integer division
IntVersTimeCode(n) = IF n / 3600 < 23,
                     TimeCode(n / 3600, (n % 3600) / 60, (n % 3600) % 60)
                     ELSE exception "n is too big"

TimeCodeVersInt(Create) = 0
TimeCodeVersInt(IntVersTimeCode(n)) = n
//n = h * (3600) + m * 60 + s * 60

// greater than or equal to case
GreaterThanEq(tc1, Create) = FALSE
GreaterThanEq(IntVersTimeCode(n), IntVersTimeCode(m)) = TRUE, IF n >= m ELSE FALSE

Addition(tc1, Create) = tc1
Addition(IntVersTimeCode(n), IntVersTimeCode(m)) = IntVersTimeCode(n + m)

Substraction(tc1, Create) = tc1
// Equivalent formulation exists with absolute value
Substraction(IntVersTimeCode(n), IntVersTimeCode(m)) = IF n >= m, IntVersTimeCode(n - m)
                                                       ELSE Substraction(IntVersTimeCode(m), IntVersTimeCode(n))
```

Example – Air traffic control

- « Sector » for airport traffic control
  - A sector is a controlled area of airspace (a set of aircraft)
  - An aircraft is identified by its call-sign (CS)
  - All aircraft in a sector must be seprated by +/- 300m

SECTOR

sort Sector
imports INTEGER, BOOLEAN

Enter - adds an aircraft to the sector if safety conditions are satisfed
Leave - removes an aircraft from the sector
Move - moves an aircraft from one height to another if safe to do so
Lookup - Finds the height of an aircraft in the sector

Create - creates an empty sector
Put - adds an aircraft to a sector with no constraint checks
In-space - checks if an aircraft is already in a sector
Occupied - checks if a specified height is available

Enter (Sector, Call-sign, Height) → Sector
Leave (Sector, Call-sign) → Sector
Move (Sector, Call-sign, Height) → Sector
Lookup (Sector, Call-sign) → Height

Create → Sector
Put (Sector, Call-sign, Height) → Sector
In-space (Sector, Call-sign) → Boolean
Occupied (Sector, Height) → Boolean

Example – Air traffic control

- Axioms

```
Enter (S, CS, H) =
   if      In-space (S, CS )  then  S exception (Aircraft already in sector)
   elsif   Occupied (S, H) then  S exception (Height conflict)
   else    Put (S, CS, H)

Leave (Create, CS) = Create exception (Aircraft not in sector)
Leave (Put (S, CS1, H1), CS) =
     if CS = CS1 then S else Put (Leave (S, CS), CS1, H1)

Move (S, CS, H) =
    if      S = Create then Create  exception (No aircraft in sector)
   elsif    not In-space (S, CS) then S  exception (Aircraft not in sector)
   elsif   Occupied (S, H) then S exception (Height conflict)
   else   Put (Leave (S, CS), CS, H)

-- NO-HEIGHT is a constant indicating that a valid height cannot be returned

Lookup (Create, CS) =  NO-HEIGHT  exception (Aircraft not in sector)
Lookup (Put (S, CS1, H1), CS) =
    if CS = CS1  then H1 else Lookup (S, CS)

Occupied (Create, H) = false
Occupied (Put (S, CS1, H1), H) =
     if      (H1 > H and H1 - H ≤ 300) or (H > H1 and  H - H1 ≤ 300)  then true
    else    Occupied (S, H)

In-space (Create, CS) = false
In-space (Put (S, CS1, H1), CS ) =
    if CS = CS1 then true else In-space (S, CS)
```

Soundness and completeness

- Based on logic and formal system research

- Soundness
  - Do not define or derive contradictory axioms

- Completeness
  - Define a sufficient number of axioms to describe the operations semantic
  - Good practice : for each inspection operation (m), write an axiom for each constructor operation (n).

    ➔ m * n axioms

Discussion

- Benefits
  - Prepare the implementation phase
    - signature → interface
    - operations list
  - Unit tests
    - Check whether the software meets the constraints expressed over the operations results
- Dedicated languages and approaches
  - ML, (CaML) Prolog, LISP
  - Méthode Z, B

References

- Thanks for your attention
  - TD in normal room (S104, S201, P201 ...)
  - Question(s) ?

- I. Sommerville, *Software Engineering*. Pearson Education, 2009.
- https://ifs.host.cs.st-andrews.ac.uk/Books/SE9/WebChapters/PDF/Ch_27_Formal_spec.pdf

- D. Hofstadter, *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books,1979 (tr. Gödel, Escher, Bach : Les Brins d'une Guirlande Éternelle, Dunod, 1985).

Annexes

Exemple

LIST (Elem)

sort List
imports INTEGER

Defines a list where elements are added at the end and removed from the front. The operations are Create, which brings an empty list into existence, Cons, which creates a new list with an added member, Length, which evaluates the list size, Head, which evaluates the front element of the list, and Tail, which creates a list by removing the head from its input list. Undefined represents an undefined value of type Elem.

Create → List
Cons (List, Elem) → List
Head (List) → Elem
Length (List) → Integer
Tail (List) → List

Head (Create) = Undefined **exception** (empty list)
Head (Cons (L, v)) = **if** L = Create **then** v **else** Head (L)
Length (Create) = 0
Length (Cons (L, v)) = Length (L) + 1
Tail (Create) = Create
Tail (Cons (L, v)) = **if** L = Create **then** Create **else** Cons (Tail (L), v)