# Métodos Formais em Engenharia de Software

**Ana Paiva**

**apaiva@fe.up.pt**

Universidade do Porto
Faculdade de Engenharia
FEUP



## Advanced functions in VDM++

- Polymorphic functions
- Higher-order functions
- The type function
- The lambda expression



## Polymorphic functions

nomeFunção[@*TypeParam1*, @*TypeParam2*, ...] ...

- Are generic functions that can be used with different values
- They have special parameters (type parameters) that must be replaced by names of specific types using the function
- Names of these parameters start with "@" and are indicated in brackets after the function name
- Like function templates in C++



## Polymorphic functions

- Example: utility function, which checks whether a sequence of elements of some kind has doubled:

```
public static HasDuplicates[@T](s: seq of @T) res: bool ==
      exists i, j in set inds s & i <> j and s(i) = s(j);
```

- Example of its use:

```
class Publication
  instance variables
    private autores: seq of Autor := [];
    inv not HasDuplicates[Autor](autores);
```

A concrete type

# The function type

- Total function: *arg1Type * arg2Type * ...* **+>** *resultType*
- Partial function: *arg1Type * arg2Type * ...* **->** *resultType*

It can be seen as a single package type argument tuple (instance calls of the product of types)

- The type of a Function is defined by the types of arguments and result
- Instances of a Function type (i.e., concrete function) can be passed as argument or as return, and saved (by reference) in data structures



# Higher order functions

- Are functions that take other functions as arguments, or (Curried functions) that return functions as a result
- I.e. have arguments or a result of type function
- Example: a function that finds an approximate zero of a function between specified limits, with maximum error specified by the method of successive bisections:

```
findZero( f: real -> real , x1, x2, err: real) res: real ==
  if abs(x1 - x2) <= err and abs(f(x1) - f(x2)) <=  err then x1
  else let m = (x1 + x2) / 2
       in if sinal(f(m)) = sinal(f(x1)) then findZero(m,x2)
          else findZero(x1,m)
 pre sinal(f(x1)) <> sinal(f(x2));
```



# The function type

| Operator | Name | Type |
|---|---|---|
| f1 comp f2 | Function composition | (B->C) * (A->B) -> (A->C) |
| f ** n | Function iteration | (A->A) * nat -> (A->A) |

| Operator name | Semantics Description |
|---|---|
| Function composition | It yields the function equivalent to applying first f2 and then applying f1 to the result. |
| Function iteration | Yields the function equivalent to applying *f n* times. *n*=0 yields the identity function which just returns the values of its parameter; *n*=1 yields the function itself. For *n*>1, the result of *f* must be contained in its parameter type. |



# The lambda expression

- lambda *patternArg1*: *Type1*, ..., *patternArgN*: *TypeN* & *expr* λ

- Constructs a function on the fly
- Patterns are usually identifiers of arguments
- Normally used to pass as argument to another function (higher order)

Example: finding a real zero of a polynomial

```
findZero(lambda x: real & 5 * x**3 -  x**2 - 2 , 0, 1, 0.0000001)
```

# Types

- Types are value types
  - Instances are immutable values pure
  - Comparison and assignment operate with their own values
  - Variable of type T name (a type) has its own data
- Subdivided into:
  - Basic types - bool, nat, real, char, ...
  - Constructed types (collections, etc..) - set of T, seq of T, map T1 to T2, ...
- New types can be defined within classes in the "types“ section
- The definition may include invariant for restricting valid instances
- Use to model types of values of attributes (data types)



# Basic types

| Symbol | Description | Examples of values |
|---|---|---|
| **bool** | Boolean | true, false |
| **nat1** | Natural number different from 0 | 1, 2, 3, ... |
| **nat** | Natural number | 0, 1, 2, ... |
| **int** | Integer | ..., -2, -1, 0, 1, ... |
| **rat** | Rational number | |
| **real** | Real number (the same as "rat" because only rational numbers can be represented in the computer) | -12.78, 0, 3, 16.23 |
| **char** | Character | 'a', 'b', '1', '2', '+', '-', ... |
| **token** | Encapsulates a value (argument mk_token) of any type (useful if you know little about the type) | mk_token(1) |
| *<identificador>* | *Quotes* (literal names, typically used to define enumerated types) | <white>, <Black> |



# Constructed types - collections

| Description | Sintax | Example of an instance |
|---|---|---|
| **Set** of elements of type A | set of A | {1 , 2} |
| **Sequence** of elements of type A | seq of A | [1, 2, 1] |
| Not empty sequence | seq1 of A | |
| **Mapping** elements of type A to type B elements (function finite set of key-value pairs) | map A to B | { 0 \|-> false,<br>1 \|-> true } |
| Injected mapping (different key values correspond to different values) | inmap A to B | |



# More constructed types

| Description | Sintax | Example of an instance |
|---|---|---|
| **Products** of types A, B, ...<br>(instances are tuples) | A * B * ... | mk_(0, false) |
| ***Record*** T with fields *a*, *b*, etc. of types *A*, *B*, etc. (*) | T :: a : A<br>b : B<br>... | mk_T(0, false) |
| **Union** of types A, B, ...<br>(type A or type B or ...) | A \| B \| ... | |
| **Opcional** type (allows **nil**) | [A] | |

(*) Alternative definitions :

```
T :: a : A
     b :- B    -- field with ":-" is ignored in the comparison of records
     Fields can be accessed by: mk_T(x,y).b

T :: A B       -- anonymous fields
     Fields can be accessed by: mk_T(x,y).#2
```

# Strings

- Not predefined type string, but can easily be defined as string (seq of char)
- All operations on sequences can be used with strings
- String literals can be indicated with quotation marks
  - "I am" is equivalent to ['I', ' ', 'a', 'm']



# Example of a type definition

```
class Pessoa
 types
   public String = seq of char;                    } sequence
   public Date :: year : nat                       }
                  month: nat                       } record
                  day  : nat;                      }
   public Sexo = <Masculino> | <Feminino>;         } Enumerated type (defined with union and quote)
 instance variables
   private nome: String;
   private sexo: Sexo;
   private dataNascimento: Date;
   ...
end Pessoa
```

attribute — Data type



# The type reference

- Reference to class object
- Allows the modeling of associations between classes and work with objects of classes
- Example: :

```
class Pessoa
   instance variables
      private conjuge : [Pessoa];
      private filhos : set of Pessoa;
 ...
```

Guard reference to an object of class Person, or nil

Guard set of 0 or more references to objects of class Person



# Symbolic constants

- Are constants which is given a name in order to make the specification more readable and easy to change
- Are declared in the section *values* with the syntax:

  [private | public | protected] *nome* [: *tipo*] = *valor*;

- Example:

```
values
    public PI = 3.1417;
```

# Boolean Operators

| | | |
|---|---|---|
| **not** b | Negation | **bool** -> **bool** |
| a **and** b | Conjunction | **bool * bool** -> **bool** |
| a **or** b | Disjunction | **bool * bool** -> **bool** |
| a => b | Implication | **bool * bool** -> **bool** |
| a <=> b | Biimplication | **bool * bool** -> **bool** |
| a = b | Equality | **bool * bool** -> **bool** |
| a <> b | Inequality | **bool * bool** -> **bool** |



# Numeric operators

| | | |
|---|---|---|
| -x | Unary minus | real -> real |
| abs x | Absolute value | real -> real |
| floor x | Floor | real -> int |
| x + y | Sum | real * real -> real |
| x - y | Difference | real * real -> real |
| x * y | Product | real * real -> real |
| x / y | Division | real * real -> real |
| x div y | Integer division | int * int -> int |
| x rem y | Remainder | int * int -> int |
| x mod y | Modulus | int * int -> int |
| x ** y | Power | real * real -> real |
| x < y | Less than | real * real -> bool |
| x > y | Greater than | real * real -> bool |
| x <= y | Less or equal | real * real -> bool |
| x >= y | Greater or equal | real * real -> bool |
| x = y | Equal | real * real -> bool |
| x <> y | Not equal | real * real -> bool |



# Operators on sets (set)

| Operador | Nome | Descrição | Tipo |
|---|---|---|---|
| e in set s1 | In | $e \in s1$ | *A* * set of *A* → bool |
| e not in set s1 | Not in | $e \notin s1$ | |
| s1 union s2 | Union | $s1 \cup s2$ | set of *A* * set of *A* → set of *A* |
| s1 inter s2 | Intersection | $s1 \cap s2$ | |
| s1 \ s2 | Difference | $s1 \setminus s2$ | |
| s1 subset s2 | subset | $s1 \subseteq s2$ | set of *A* * set of *A* → bool |
| s1 psubset s2 | proper subset | $s1 \subset s2$ $(s1 \subseteq s2 \wedge s1 \neq s2)$ | |
| s1 = s2 | equal | $s1 = s2$ | |
| s1 <> s2 | Not equal | $s1 \neq s2$ | |
| card s1 | Cardinal | # s1 | set of *A* → nat |
| dunion ss | Distributed union | $\bigcup_{s_i \in ss} s_i$ | set of set of *A* → set of *A* |
| dinter ss | Distributed intersection | $\bigcap_{s_i \in ss} s_i$ | |
| power s1 | Set of sets | $\mathbb{P}$(s1) | set of *A* → set of set of *A* |



# Exercises (sets)

- {1,...,6}
- {1,...,1}
- {4,...,1}
- {x | x in set {2,3,4,5} & x > 2}
- {x | x in set {2,3,4,5} & 22 < x}
- {} in set power {1,3,6}
- dunion {{1,2},{1,5,6},{3,4,6}}
- dinter {{1,2},{1,5,6},{3,4,6}}
- {1,2,3} psubset {1,2}

# Operators on sequences (seq)

| Operador | Nome | Descrição | Tipo |
|---|---|---|---|
| hd l | Cabeça (*head*) | Dá o 1º elemento de l, que não pode ser vazia | seq of $A \rightarrow A$ |
| tl l | Cauda (*tail*) | Dá a subsequência de l em que o 1º elemento foi removido. l não pode ser vazia | seq of $A \rightarrow$ seq of $A$ |
| len l | Comprimento | Dá o comprimento de l | seq of $A \rightarrow$ nat |
| elems l | Elementos | Dá o conjunto formado pelos elementos de l (sem ordem nem repetidos) | seq of $A \rightarrow$ set of $A$ |
| inds l | Índices | Dá o conjunto dos índices de l, i.e., {1, ..., len l} | seq of $A \rightarrow$ set of nat1 |
| l1 ^ l2 | Concatenação | Dá a sequência formada pelos elementos de l1 seguida pelos elementos de l2 | (seq of $A$) * (seq of $A$) $\rightarrow$ seq of $A$ |
| conc ll | Concatenação distribuída | Dá a sequência formada pela concatenação dos elementos de ll (que são por sua vez sequências) | seq of seq of $A \rightarrow$ seq of $A$ |
| l ++ m | Modificação de sequência | Os elementos de l cujos índices estão no domínio de m são modificados para o valor correspondente em m. Deve-se verificar: dom m subset inds l. | (seq of $A$) * (map nat1 to $A$) $\rightarrow$ seq of $A$ |
| l(i) | Aplicação de sequência | Dá o elemento que se encontra no índice i de l. Deve-se verificar: i in set inds l. | seq of $A$ * nat1 $\rightarrow A$ |
| l(i, ..., j) | Subsequência | Dá a subsequência de l entre os índices i e j, inclusive. Se i < 1, considera-se 1. Se j > len s, considera-se len(s). | seq of $A$ * nat * nat $\rightarrow$ seq of $A$ |



# Exercises (seq)

- Which of the following expressions are true?
- 6 in set elems [3,6,8,10,0]
- [] = tl [4]
- 6 in set inds [3,6,8,10,0]



# Exercises (seq)

- 2.2 What are the results of the following expressions:
- tl [1,2,3]
- len [[1,2],[1,2,3]]
- hd [[1,2],[1,2,3]]
- tl [[1,2],[1,2,3]]
- elems [1,2,2,3,3,4]
- elems [[1,2],[2],[3],[3],[3,4]]



# Exercises (seq)

- 2.3 What is the value of the following expressions
- len []
- len [1,2,3] + len [3]
- [hd [<A>,<B>]] ^ [hd [<C>,<D>]]
- tl [1,2,3,4,5] ^ [hd [1,2,2]]
- tl ([1,2]^[1,2])