Métodos Formais em Engenharia de Software

Ana Paiva

apaiva@fe.up.pt

Universidade do Porto
Faculdade de Engenharia
FEUP

Expressions: examples

Expression	Example
new	new C()
self	Create: **nat** ==> C Create (n) == (a := n; **return self**) -- Inicializa um objecto de uma classe com uma variável de instância -- *a* de tipo *nat* e guarda a referência para esse objecto
is	**Ex1:** is_nat(5) **Ex2:** is_C(mk_C(5))
isofclass	**isofclass**(Class_name,object_ref) -- Retorna true se *object_ref* é da classe *Class_name* ou uma -- subclasse da classe *Class_name*.
isofbaseclass	**isofbaseclass**(Class_name,object_ref) -- Para que o resultado seja true, *object_ref* tem que ser da classe -- *Class_name*, e *Class_name* não pode ter superclasses.
sameclass	**sameclass**(obj1, obj2) -- true se e só se *obj1* e *obj2* são instâncias da mesma classe
samebaseclass	**samebaseclass**(obj1, obj2)
Object reference	o operador **=** só retorna true se os dois objectos são a mesma instância; o operador **<>** retorna true quando os objectos não são a mesma instância, mesmo que tenham os mesmos valores nas variáveis de instância.

Expressions: examples

Expression	Example
forall	**forall** bind list **&** expression e.g., forall x in set elems l & m<=n
exists	**exists** bind list **&** expression e.g., exists x in set elems & x<5
exists1	**exists1** bind list **&** expression e.g., exists1 x in set elems & x<5

Instructions: examples

Instrução	Exemplo
let	**let** cs' = {c |-> cs(c) **union** {s}}, ct' = {s |-> ct(s) **union** {c}} **in** sub_stmt1
let be	**let** i **in set inds** l **be st** Largest (**elems** l, l(i)) **in** sub_stmt2
define	**def mk_**(r,-) = OpCall() **in** (x := r / 2; **return** x) -- Allows binding of the result of an operation call to a pattern
if-then-else	**if** i = 0 **then return** <Zero> **elseif** 1 <= i **and** i <= 9 **then return** <Digit> **else return** <Number>
cases	**cases** a: **mk_**A(a',-,a') -> Expr(a'), **mk_**A(b.b.c) -> Expr2(b,c), **others** -> Expr3() **end**
assign	x := 5

Instructions: examples

Instrução	Exemplo
block	(**dcl** a: **nat** := 5;**dcl** b: **bool**; stmt1; ...; stmtn) -- Se *stmt1* returna um valor, a execução do bloco termina e esse -- valor é retornado como resultado de todo o bloco.
loop	**Ex1: for** id = lower **to** upper [**by** step] **do** stmt **Ex2: for all** pat **in set** setexpr **do** stmt **Ex3: for all** pat **in seq** seqexpr **do** stmt
while	**while** expr **do** stmt
always	(**dcl** mem: Memory; **always** Free(mem) **in** (mem := Allocate(); Command(mem, ...)))
return	**return** expr or **return**
exit	**exit** expr -- para sinalizar excepção

Instructions: examples

Instrução	Exemplo
exception handling	**Ex1: trap** pat **with** ErrorAction(pat) **in** (**dcl** mem: Memory; **always** Free(mem) **in** (mem := Allocate(); Command(mem, ...))) **Ex2:** DoCommand : () ==> int DoCommand () == (dcl mem : Memory; always Free(mem) in (mem := Allocate(); Command(mem, ...))); Example : () ==> int Example () == **tixe** { <NOMEM> |-> return -1, <BUSY> |-> DoCommand(), err |-> return -2 } **in** DoCommand()

Blocks and variable declarations

```
(
        dcl id1 : tipo1 [:= expr1],  id2 : tipo2 [:= expr2], ...;
        dcl ... ;
        ...
        instruction1;
        instruction2;
        ...
)
```

- A block must have at least one instruction
- Variables may be declared only at the beginning of the block
- Last statement does not need ";"
- The 1st statement that return a value (even without a "return", just call one operation that returns a value) is finishing the block

Assignment

state variable := expression

State variable name
- Instance variable of the object in question
- Static variable (static)
- Local variable of the transaction (stated with dcl)

Part of variable of type map, record or seq
- map_var(key) := valor
- seq_var(indice) := valor
- record_var.field := valor

STOP An identifier introduced with let , forall, etc. is not a variable in this sense

Multiple assignment

atomic (*sd1* := *exp1*; *sd2* := *exp2*; ...)

- First evaluates all expressions on the right side and then assigns them to the variables at the left side at once!
- Checks invariants at the end of all attributions (otherwise, it would check invariants after each attribution)

Useful in the presence of invariants involving more than one instance variable (of the same object)

It does not solve the problem of inter-object invariants, i.e., involving multiple objects (why?)

Multiple assignment - example

```
instance variables
  private quantidade : real;
  private precoUnitario : real;
  private precoTotal : real;
  inv precoTotal = quantidade * precoUnitario;
operations
public SetQuantidade(q: real) ==
  (quantidade := q;  precoTotal := precoUnitario * q);
```

Breaks invariant after 1st atribution

```
public SetQuantidade(q: real) ==
  atomic(quantidade := q; precoTotal := precoUnitario * q);
```

```
public SetQuantidade(q: real) ==
  atomic(quantidade:=q; precoTotal:=precoUnitario * quantidade);
```

wrong: uses old value of the variable

Instructions “let” and “def”

let *definition1*, *definition2*, ... in *instruction*

let *identifier* in set *Set* [be st *condition*] in *instruction*

def *definition1*, *definition2*, ... in *instruction*

Have the same form as expressions "let" and "def", with instruction rather than expression in the “in” part

Using "def" instead of "let", when in the definitions part are invoked operations that change state

Identifiers introduced in the definition are not variables that can change the value (can not appear on the left side of assignments)!

Instructions “if” and “cases”

if *condition* then *instruction1* [else *instruction2*]

cases *expression*:
  pattern11, *pattern12*, ..., *pattern1N* -> *instruction1*,
  ... -> ...,
  patternM1, *patternM2*, ..., *patternMN* -> *instructionM*,
  others -> *instructionM1*
end

Have the same form as the expressions "if" and "cases", with instructions instead of expressions

In the "if" statement, the party of "else" is optional

Instruction “return”

return

- Used to end operations that do not return any value

return *expression*

- Used to complete transactions that return a value

Beware of return implicit: the 1st instruction to return a value (just call operation that returns value) does end the block

Expression: “new”

- Create object:
  - new *name-of-the-class(parmeters-constructor)*
- Delete object: automatic, like in Java and C#
  - Are automatically deleted when no longer referenced
  - What we can do is to explicitly remove an object or a collection by assigning nil dereference (obj_ref: = nil)
  - Prevents errors and simplifies the specification
  - In contrast, prevents to know that there are instances of a given class at any given time (in OCL is ClassName.allInstances)
- Modify state of the subject: see assignment operator

Syntactic aspects

- Comments begin with "--" and go to the ends of the line
- Distinction of uppercase and lowercase letters (case sensitive)
- Accents are partially supported, it is preferable not to use them
- To cite an instance member (instance variable or operation) of an object, we use the usual notation "object.member"
- To refer to a static member (variable, operation or static function), type or constant defined in another class, we use the notation “class`member”, not “classe.membro”
- Use "nil" and not "null"
- Use "self" and not "this“

Mathematical notation vs ASCII

·	&
×	*
≤	<=
≥	>=
≠	<>
$\xrightarrow{o}$	==>
→	->
⇒	=>
⇔	<=>

↦	|->
$\triangleq$	==
↑	**
†	++
⩊	munion
◁	<:
▷	:>
⩤	<-:
⩥	:->
⊂	psubset
⊆	subset
↷	^
∩	dinter
⋃	dunion
$\mathcal{F}$	power
...-set	set of ...
...*	seq of ...
...+	seq1 of ...
$\ldots \xrightarrow{m} \ldots$	map ... to ...

$\ldots \xleftrightarrow{m} \ldots$	inmap ... to ...
μ	mu
$\mathbb{B}$	bool
$\mathbb{N}$	nat
$\mathbb{Z}$	int
$\mathbb{R}$	real
¬	not
∩	inter
∪	union
∈	in set
∉	not in set
∧	and
∨	or
∀	forall
∃	exists
∃!	exists1
λ	lambda
ι	iota
...-1	inverse ...