Datatypes

Para definir uma árvore binária:

```
datatype Tree = Leaf | Node(Tree, Tree)
```

- Declaração de tipo

  ```
  datatype D<T> = Ctor1 | Ctor2 | ...

  datatype List<T> = Nil | Cons(head: T, tail: List<T>)
  datatype Semaphore = Green | Yellow | Red
  ```

- Instancia

  ```
  var list1 := Cons(5, Nil); var sem1 := Green;
  ```

- Actualizar um valor

  ```
  d[CtorParam := Value]

  list1 := list1[head := 1];
  ```

- Verificar um constructor

  ```
  d.Ctor?
  ```

- Selecionar um campo

  ```
  d.CtorParam

  function Length(x: List<T>) : nat {
  if x.Cons? then 1 + Length(x.tail) else 0
  }
  ```

- Análise de casos:

  ```
  function Length(x: List<T>) : nat { match x {
  case Nil => 0
  case Cons(h, t) => 1 + Length(t) } }
  ```

Copiar os valores duma árvore para um array

```
type T = int
datatype Tree<T> = Leaf | Node(Tree, T, Tree)
method Fill(t: Tree<T>, a: array<T>, start: int)
                returns (end: int)
{
match t {
    case Leaf => end := start;
    case Node(left, x, right) =>
      end := Fill(left, a, start);
    if end < a.Length {
        a[end] := x;
        end := Fill(right, a, end + 1);
    }
}
}
```

Verificar a correção de Fill

```
function contains<T(==)>(t: Tree<T>, v: T): bool
{
match t {
case Leaf => false
case Node(left, x, right) =>
x == v || contains(left, v) || contains(right, v)
}
}

}

method Main()
{ var q := Node(Node(Leaf,4,Leaf),5, Node(Leaf,3,Leaf));
  var t := new int[10];
  var n := Fill(q,t,0);
  assert contains(q,3);
  print t[0], t[1], t[2];
}
```

Pre and Pos Conditions

```
method Fill(t: Tree<T>, a: array<T>, start: int)
          returns (end: int)
requires 0 <= start <= a.Length
decreases t
```

```
modifies a
ensures  start <= end <= a.Length
ensures forall i:: 0 <= i < start ==> a[i] == old(a[i])
ensures forall i:: start <= i < end ==> contains(t, a[i])
```

Selecionadores de campos/destruídores

To access the fields of a datatype one can name them

```
type T = int
datatype Tree<T> = Leaf | Node(left:Tree, key:T, right:Tree)
```

and

```
function countLeaf(t: Tree<T>): nat {
 if t.Leaf? then 1
 else countLeaf(t.left) + countLeaf(t.right)
}
```

Lists

Ver ficheiros sobre listas

Lemas e Indução

```
predicate OddPre(n: nat) {
    n%2 == 1
}

function Odd(n:nat): bool
decreases n
{ if n == 1 then true
else if  n == 0 then false
else !Odd(n-1)
}



function OddTail(n:nat,o:bool): bool
decreases n
{ if n == 1 then o
else if  n == 0 then ! o
else OddTail(n-1, !o )
}

lemma {:induction} TestOdd(n: nat)
```

```
decreases n
ensures Odd(n) == OddPre(n)
{}

datatype Nat = Z | S(Nat)

function   Plus(x:Nat, y:Nat): Nat
decreases x;
{
match x {
    case Z => y
    case S(z) => S(Plus(z,y))
}
}

lemma {:induction n} example(n:Nat)
decreases n;
ensures Plus(n,Z) == n
{}

lemma {:induction false} example0 (n: Nat)
decreases n
ensures Plus(n,Z) == n
{
    match n {
    case Z => { assert Plus(Z,Z) == Z;}
    case S(k) => {
        assert Plus(n,Z) == S(Plus(k,Z));
        example0(k);
        assert S(Plus(k,Z)) == S(k);
        assert  S(k) == n;
        }
    }

}
```

DbC: design by contract

- Conceito introduzido por Bertrand Meyer para a linguagem Eiffel em 1980's
- Um *contrato* de um procedimento ou objecto é um par (pré-condições, pós-condições) tal que

  pré-condições : indicam as obrigações que são requesitos de quem evoca expressando condições sobre os dados de entrada ou estado inicial de um objecto

pós-condições benefícios que são esperados quando o procedimento termina ou do estado final de um objecto

- um contrato é violado se quem invoca não respeita as pré-condições ou se as pós-condições não são satisfeitas

```
class interface ACCOUNT

create
    make

feature

    balance: INTEGER
        ...

    deposit (sum: INTEGER)
            -- Deposit sum into the account.
        require
            sum >= 0
        ensure
            balance = old balance + sum

    withdraw (sum: INTEGER)
            -- Withdraw sum from the account.
        require
            sum >= 0
            sum <= balance - minimum_balance
        ensure
            balance = old balance - sum

    may_withdraw ...

end -- ACCOUNT
```

Objectos em Dafny

- um objecto é uma instância duma classe
- é acedido por referência e tem um estado associado
- uma classe define as componentes dum objecto, *campos/atributos*, e operações, *methods*: *os membros*da classe.

Invariantes de Classes (Objectos)

- Um *invariante de classe* é uma condição que define os estados válidos (valores dos atributos) de objectos (instâncias da classe).
- Para além das pré/pós condições de cada método, dentro da classe, tem de respeitar o invariante da classe:

- Cada constructor tem de assegurar que todos os invariantes se verificam no fim da sua execução.
- Cada método tem de assegurar que todos os invariantes se verificam no fim da sua execução, supondo que se verificam no ínicio.

```
class
```

```
class Stack
{
    const elems: array<T>;
    var size : nat;
    var capacity: nat;
    constructor (cap: nat)
    {
        elems := new T[cap];
        capacity := cap;
        size := 0;
    }
    predicate  isEmpty()
    {
        size == 0
    }
    predicate   isFull()
    {
        size == elems.Length
    }
    method push(x : T)
    {
        elems[size] := x;
        size := size + 1;
    }
    function   top() : T
    {
         elems[size-1]
    }
    method pop()
    {
         size := size-1;
    }
}
```

Objectos

- atributos/campos: `elems, size`

- constructores: chamados quando os objectos são criados ()
- métodos: `method`
- criação: `var s:= new Stack(3)`

```
Stack

method {:verify false} testStack()
{
    var s := new Stack(3);
    assert s.isEmpty();
    s.push(1);
    s.push(2);
    s.push(3);
    assert s.top() == 3;
    assert s.isFull();
    s.pop();
    assert s.top() == 2;
}
```

Objectos

- classes (ou *trait*) podem estender outras classes
- Métodos podem ser declarados estáticos
- O objecto é referido por `this`
- e os seus membros por by ".": `this.size` (mas `this` pode ser omitido nalguns contextos)
- Uma variável duma classe (instância) pode ser `null` se declarada com `?` a seguir ao nome da classe
- Os construtores podem ter nomes
- O tipo de todas as classes é `object`
- Atributos imutáveis podem ser declarados com `const`
- Podem incluir `predicates` ou `functions`.

```
class A {
 constructor()
}

method main(){
```

```
var a: A := new A();
var o:object = a;
var n:A? = null;
}
```

Invariante de classe- `Valid`

- predicado `Valid()` descreve o invariante da classe
- este invariante é requerido (pré-condição) por todos os métodos/funções (excepto os constructores)
- e verificado no fim da execução (pós-condição) por todos os construtores e modificadores (qualquer método)
- com o atributo `:autocontracts` o Dafny adiciona automaticamente `requires/ensures` do predicado `Valid()` e outros contratos necessários.

***framing* para objectos**

- Os objectos são mutáveis e alocados na *heap*
- é usado o mecanismo de *dynamic frames*
- É necessário indicar para *function* (ou *predicate*) que parte é que são lidas `reads` e para os *method* que partes são modificadas `modifies` (supondo-se que as restantes não são alteradas)
- se apenas um campo de um objecto for lido isso pode ser indicado com ‘ : `reads this‘Value`

Counter

```
class Counter {
var Value: int;

predicate Valid()
reads this‘Value
{
Value >= 0
}
constructor Init ()
ensures Value == 0
ensures Valid()
{ Value := 0;}
method GetValue() returns (x: int)
```

```
modifies this
requires Valid()
ensures x == Value
{x := Value;}
```

Counter

```
method Inc ()
 modifies this
 requires Valid()
 ensures Value == old(Value)+1
 ensures Valid()
 {Value := Value + 1;}

method Dec ( )
modifies this
requires Value > 0 && Valid()
ensures Value == old ( Value ) -1 && Valid()
{Value := Value -1;}
}
```

Alocação de novos objectos

- se durante a execução de um método novos objectos são alocados na *heap* isso deve ser indicado com `fresh`.

```
method NewSorted(n: nat) returns (v: array<int >)
ensures fresh ( v )
ensures Sorted ( v )
ensures v.Length = n
{}
```

Invariante de `Stack`

```
ghost predicate Valid()
reads this
{
size <= capacity == elems.Length
}

constructor (cap: nat)
```

```
    requires cap > 0
    ensures fresh(elems)
    ensures Valid()
    ensures cap == capacity && size ==0
    {
        elems := new T[cap];
        capacity := cap;
        size := 0;
    }

predicate isEmpty()
    reads this
    {
        size == 0
    }

predicate isFull()
    reads this
    {
        size == elems.Length
    }

method  push(x : T)
    modifies this, elems
    requires Valid() && size < capacity
    {
        elems[size] := x;
        size := size + 1;
    }

function  top(): T
    requires Valid() && size > 0
    reads this, elems
    {
         elems[size-1]
    }

method pop()
    requires Valid() && size > 0
    modifies this
    ensures Valid()
    ensures elems[..size] == old(elems[..size-1])
    {
         size := size-1;
    }
}
```

```
:autocontracts

class{:autocontracts} Stack
{
    const elems: array<T>;
    var size : nat;
    var capacity: nat;
    ghost predicate Valid()
    reads this
    { size <= capacity == elems.Length
    }
    constructor (cap: nat)
    ensures cap == capacity && size ==0
    {
        elems := new T[cap];
        capacity := cap;
        size := 0;
    }
    predicate method  isEmpty()
    {
        size == 0
    }
    predicate method isFull()
    {
        size == elems.Length
    }


:autocontracts


    method push(x : T)
    requires size < capacity
    modifies this, elems
    ensures elems[..size]==old(elems[..size])+[x]
    {
        elems[size] := x;
        size := size + 1;
    }

    function  top(): T
    requires size > 0
    {
         elems[size-1]
    }
```

```
    method  pop()
    requires size > 0
    modifies this
    ensures elems[..size] == old(elems[..size-1])
    {
         size := size-1;
    }
}
```

Herança

Uma subclasse não pode quebrar o contrato da sua super classe, mas

- A pré-condição pode ser enfraquecida
- A pós-condição pode ser fortelecida

Exemplo: A classe `Circle` herda da classe `Shape` .

```
trait  Shape
{
    var center: (real, real);

    ghost predicate Valid()
      reads this

    function  getSizeX(): real
       requires Valid()
       reads this

    function  getSizeY(): real
       requires Valid()
       reads this

    method resize(factor: real)
      requires factor > 0.0 && Valid()
      modifies this
      ensures Valid()
      ensures getSizeX() == factor * old(getSizeX())
      ensures getSizeY() == factor * old(getSizeY())
      ensures center == old(center)
}

class Circle extends Shape
{
    var radius: real;
```

```
    predicate Valid()
      reads this
    {
        radius > 0.0
    }
    constructor Circle(center: (real, real), radius: real)
      requires radius > 0.0
      ensures this.center == center && this.radius == radius && Valid()
    {
        this.center := center;
        this.radius := radius;
    }
    function  getSizeX(): real
      requires Valid()
      reads this
    {
        radius
    }
    function  getSizeY(): real
      requires Valid()
      reads this
    {
        radius
    }
}
```

Subclasse

```
    method resize(factor: real)
      requires factor != 0.0 && Valid()
      modifies this
      ensures center == old(center)
      ensures radius == abs(factor) * old(radius)
      ensures Valid()
    {
       radius := abs(factor) * radius;
    }

    function abs(x: real): real
    {
        if x >= 0.0 then x else -x
    }
}
```

See that

$$factor > 0.0 \rightarrow factor \neq 0.0$$
$$radius == abs(factor) * old(radius) \rightarrow getSizeX() == factor * old(getSizeX())$$