

# Verifica funzionale di programmi con Dafny

Lorenzo Quellerba

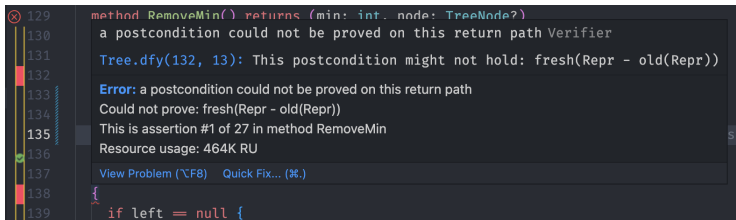
Univeristà degli Studi di Torino

June 2023

- Dafny è un linguaggio di programmazione che supporta nativamente la verifica funzionale
- Durante lo sviluppo, un programma Dafny viene annotato con la specifica del comportamento e automaticamente un verificatore statico controlla che il codice rispetti la specifica (correct by construction)
- Il linguaggio supporta (caratteristiche ad alto livello)



- La formalizzazione della specifica è resa possibile da precondizioni, postcondizioni, invarianti, metriche di terminazione e il framing della memoria nel caso di side effects
- Il verificatore statico esegue continuamente la verifica in background grazie ad un LSP e riporta gli errori: per l'utente l'interazione è la medesima che si avrebbe con un compilatore



```
129 method RemoveMin() returns (min: int, node: TreeNode?)
130   a postcondition could not be proved on this return path Verifier
131   Tree.dfy(132, 13): This postcondition might not hold: fresh(Repr - old(Repr))
132
133   Error: a postcondition could not be proved on this return path
134   Could not prove: fresh(Repr - old(Repr))
135   This is assertion #1 of 27 in method RemoveMin
136   Resource usage: 464K RU
137   View Problem (⌘F8) Quick Fix... (⌘.)
138
139   {
140     if left == null {
```

- Il programma finale può essere compilato in altri linguaggi (elenco) per permette integrazione in altri programmi

# Dafny: funzionamento

## Tripla di Hoare

$$\{P\}C\{Q\}$$

Se l'asserzione  $P$  è vera prima dell'esecuzione del comando  $C$  allora l'asserzione  $Q$  sarà vera al termine dell'esecuzione

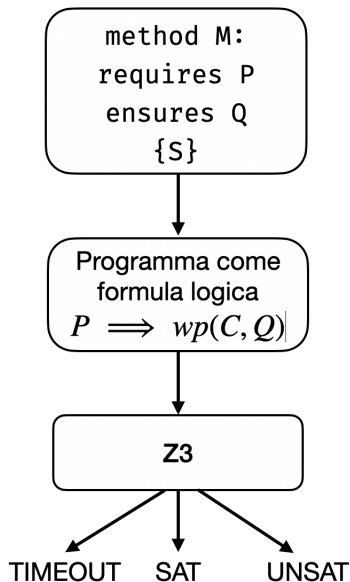
## *Predicate transformer semantics*

La semantica dei *predicate transformer* è una riformulazione della logica di *Floyd-Hoare* che definisce una strategia completa per la costruzione di deduzioni valide

## *Weakest precondition*

Dato un comando  $C$  e una postcondizione  $Q$  la *weakest precondition* ( $wp$ ) è un predicato  $\phi$  tale per cui per ogni preconditione  $P$ ,  $\{P\}C\{Q\}$  se e solo se  $P \implies \phi$

# Dafny: funzionamento



# Caratteristiche del linguaggio

- Reference types e value types
- Generici
- predicati metodi funzioni classi
-

Richiamo di teoria della struttura

# BST: variabili d'istanza

```
class TreeNode
{
  var data: int
  var left: TreeNode?
  var right: TreeNode?

  ghost var Repr: set<object>
  ghost var Contents: set<int>
```



# BST: invariante di struttura

```
ghost predicate Valid()  
  reads this, Repr  
{  
  this in Repr &&  
  (left != null ==>  
    left in Repr &&  
    left.Repr <= Repr && this !in left.Repr &&  
    (forall e :: e in left.Contents ==> e < data) &&  
    left.Valid()) &&  
  (right != null ==>  
    right in Repr &&  
    right.Repr <= Repr && this !in right.Repr &&  
    (forall e :: e in right.Contents ==> data < e) &&  
    right.Valid())  
  && (left != null && right != null ==> left.Repr < right.Repr)  
  && Contents == (if left == null then {} else left.Contents  
                  (if right == null then {} else right.Contents))
```

# BST: costruttore

```
constructor (x: int)
  ensures fresh(Repr - {this})
  ensures Valid()
  ensures Contents == {x}
{
  data := x;
  left := null;
  right := null;

  Repr := { this };
  Contents := { data };
}
```

# BST: inserimento

```
method Insert(x: int)
  requires Valid()
  modifies Repr
  ensures Valid() && fresh(Repr - old(Repr)) && Con
  decreases Repr
{
  if x == data { return; }
  if x < data {
    if left == null {
      left := new TreeNode(x);
    } else {
      left.Insert(x);
    }
    Repr := Repr + left.Repr;
  } else {
    if right == null {
      right := new TreeNode(x);
    } else {
      right.Insert(x);
    }
    Repr := Repr + right.Repr;
  }
}
```

```
method Find(x: int) returns (present: bool)
  requires Valid()
  ensures present <==> x in Contents
  decreases Repr
  ensures Valid()
{
  if x == data
  {
    present := true;
  }
  else if left != null && x < data
  {
    present := left.Find(x);
  }
  else if right != null && data < x
  {
    present := right.Find(x);
  }
}
```

# BST: cancellazione

```
method Remove(x: int) returns (node: TreeNode?)
  requires Valid()
  modifies Repr
  ensures fresh(Repr - old(Repr))
  ensures node != null ==> node.Valid()
  ensures node == null ==> old(Contents) <= {x}
  ensures node != null ==> node.Repr <= Repr && node
  decreases Repr
{
  node := this;
  if left != null && x < data {
    var t := left.Remove(x);
    left := t;
    Contents := Contents - {x};
    if left != null { Repr := Repr + left.Repr; }
  } else if right != null && data < x {
    var t := right.Remove(x);
```

# BST: cancellazione

```
method RemoveMin() returns (min: int, node: TreeNod
  requires Valid()
  modifies Repr
  ensures fresh(Repr - old(Repr))
  ensures node != null ==> node.Valid()
  ensures node == null ==> old(Contents) == {min}
  ensures node != null ==> node.Repr <= Repr && nod
  ensures min in old(Contents) && (forall x :: x in
    decreases Repr
{
  if left == null {
    min := data;
    node := right;
  } else {
    var t;
    min, t := left.RemoveMin();
    left := t;
```