

# Verifica funzionale di programmi con Dafny

Lorenzo Quellerba

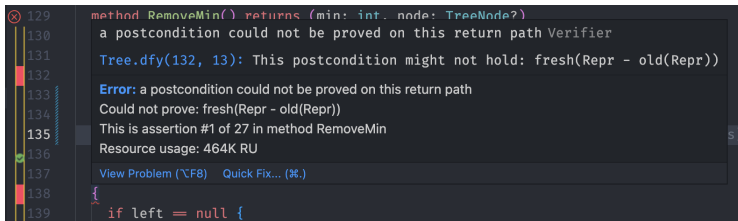
Università degli Studi di Torino

13 Giugno 2023

- Dafny è un linguaggio di programmazione che supporta nativamente la verifica funzionale
- Durante lo sviluppo, un programma Dafny viene annotato con la specifica formale del comportamento atteso e automaticamente un verificatore statico (un dimostratore automatico) controlla che il codice la rispetti (*correct by construction*)
- Il linguaggio supporta sia il paradigma imperativo che quello funzionale, i generici, l'ereditarietà, l'incapsulamento, l'allocazione dinamica della memoria e i tipi di dato induttivi



- Durante lo sviluppo il verificatore statico esegue costantemente la verifica in background e mostra a schermo eventuali errori: dal punto di vista dell'utente l'interazione è molto simile a quella che normalmente si ha con un compilatore



The screenshot shows the Dafny IDE interface. On the left, a vertical line of markers indicates the current position in the code. The main area displays the following code snippet:

```
method RemoveMin() returns (min: int, node: TreeNode?)  
  a postcondition could not be proved on this return path Verifier  
  Tree.dfy(132, 13): This postcondition might not hold: fresh(Repr - old(Repr))  
  
  Error: a postcondition could not be proved on this return path  
  Could not prove: fresh(Repr - old(Repr))  
  This is assertion #1 of 27 in method RemoveMin  
  Resource usage: 464K RU  
  
  View Problem (⌘F8) Quick Fix... (⌘.)
```

The error message indicates that a postcondition could not be proved on a specific return path. The error is highlighted in red, and the IDE provides options to view the problem or quick fix it.

- La formalizzazione della specifica è resa possibile da *keyword* riservate per le precondizioni, postcondizioni, invarianti, metriche di terminazione e per il framing della memoria nel caso in cui il programma abbia *side effects*

- Inoltre, per supportare ulteriormente la specifica, il linguaggio offre variabili *ghost* aggiornabili, funzioni ricorsive e tipi come *set* e liste
- I costrutti per la specifica e i costrutti *ghost* vengono utilizzati unicamente durante la verifica e sono omessi dal programma eseguibile
- Il programma finale può essere compilato in altri linguaggi come ad esempio (C++, Java e Go) per permettere l'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

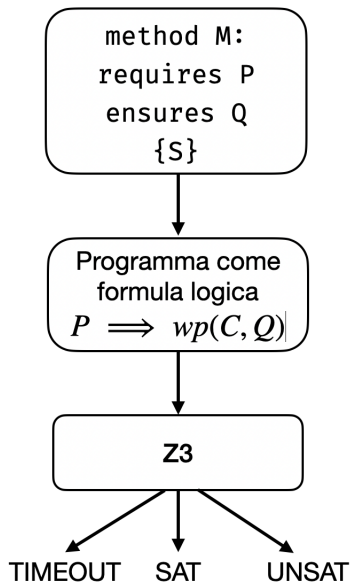
## *Predicate transformer semantics*

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

- I programmi Dafny sono principalmente composti da classi, metodi e predicati

```
class Nome {  
    var nomeVar : tipo  
  
    constructor(param : tipo)  
        ensures _postcondizione_  
    { Corpo.. }  
  
    predicate Valid()  
        reads _frame di memoria_  
    { Corpo.. }  
  
    method NomeMetodo (param : tipo) returns (valore : tipo)  
        requires _precondizione_  
        modifies _frame di memoria_  
        ensures _postcondizione_  
        decreases _metrica di terminazione_  
    { Corpo.. }  
}
```

# Caso di studio: BST

## Albero binario di ricerca

Un albero binario di ricerca è una struttura dati concatenata in cui ogni nodo è un oggetto. Ogni nodo al suo interno contiene una chiave e gli attributi *left* e *right* che puntano rispettivamente al figlio sinistro e al figlio destro

## Proprietà degli alberi binari di ricerca

Sia  $x$  un nodo in un albero binario di ricerca. Se  $y$  è un nodo nel sottoalbero sinistro di  $x$  allora  $y.key < x.key$ . Se  $y$  è un nodo nel sottoalbero destro di  $x$  allora  $y.key > x.key$



# 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) && // BST
    left.Valid()) &&
  (right ≠ null ⇒
    right in Repr &&
    right.Repr ≤ Repr && this !in right.Repr &&
    (forall e :: e in right.Contents ⇒ data < e) && // BST
    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) +
    {data}
}
```

# 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)) &&
    Contents = old(Contents) + {x}
  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;
  }
  Contents := Contents + {x};
}
```

# BST: ricerca

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

# 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.Contents = old(Contents) - {x}
  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);
    right := t;
    Contents := Contents - {x};
    if right ≠ null { Repr := Repr + right.Repr; }
  } else if x = data {
```

# BST: cancellazione

```
} else if x = data {  
  if left = null && right = null {  
    node := null;  
  } else if left = null {  
    node := right;  
  } else if right = null {  
    node := left;  
  } else {  
    // rotate  
    var min, r := right.RemoveMin();  
    data := min; right := r;  
    Contents := Contents - {x};  
    if right ≠ null { Repr := Repr + right.Repr; }  
  }  
}  
}
```

## BST: cancellazione

```

method RemoveMin() returns (min: int, node: TreeNode?)
  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
  ||||| 86 node.Contents = old(Contents) - {min}
  ensures min in old(Contents)
  ||||| 86 (forall x :: x in old(Contents) ⇒ min ≤ x)
  decreases Repr
{
  if left = null {
    min := data;
    node := right;
  } else {
    var t;
    min, t := left.RemoveMin();
    left := t;
    node := this;
    Contents := Contents - {min};
    if left ≠ null { Repr := Repr + left.Repr; }
  }
}

```