Dafny

Un linguaggio per la verifica funzionale

Lorenzo Quellerba

Università degli Studi di Torino

May 2023

Contenuti

Introduzione alla verifica funzionale

- La verifica funzionale è una tecnica di analisi statica
- L'obiettivo è quello di stabilire con rigore logico la correttezza di un programma relativamente alla specifica del suo comportamento
- Tradizionalmente, il processo di verifica avviene attraverso dimostrazioni su carta o mediante l'uso di proof assistant (processo lungo e che richiede esperienza)
- In particolare ci si concentra sull'automatizzazione del processo

Introduzione alla verifica funzionale

Uno dei requisiti fondamentali alla base di una teoria per la verifica funzionale di programmi è la **modularità**

```
1 class C {
2  var x:int;
3  method i()
4   ensures x>old(x)
5  {x := x+1;}
6 }
7
```

```
1 class Client {
2  method m0(c: C)
3   ensures c.x>2*old(c.x)
4  {
5    c.x := 2*c.x;
6    c.i();
7  }
8 }
```

L'altro requisito è quello del supporto all'**incapsulamento**

```
1 class C {
2  var x:int;
3  method i()
4   ensures getX() > old(
      getX())
5  {x := x+1;}
6  method getX()
7  { return x;}
8 }
9
```

```
1 class Client {
2  method m0(c: C)
3   ensures c.getX()>2*old(c
      .getX())
4  {
5    c.x := 2*c.x;
6    c.i();
7  }
8 }
```

Introduzione alla verifica funzionale

- Si presenta Dafny, un linguaggio di programmazione imperativo orientato agli oggetti simile a Java
- Dafny supporta "nativamente" il processo di verifica attraverso *keyword* dedicate all'interno del linguaggio alla definizione della specifica e un SMT solver che si occupa di dimostrare la validità delle verification conditions in modo (semi-)automatico

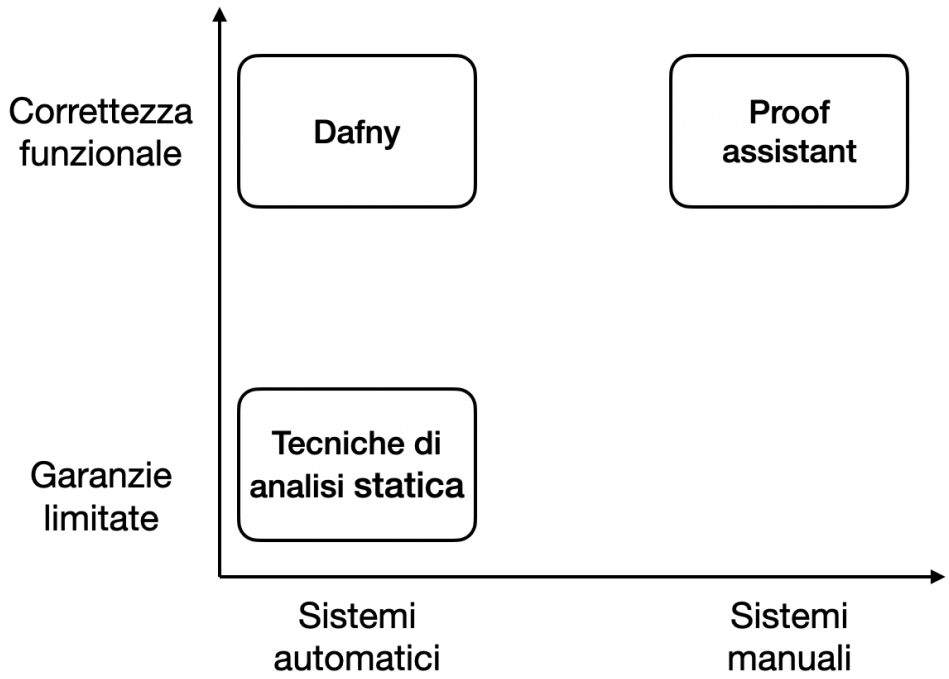


Figure: Sistemi di analisi statica

L'architettura del sistema che si presenta è la seguente

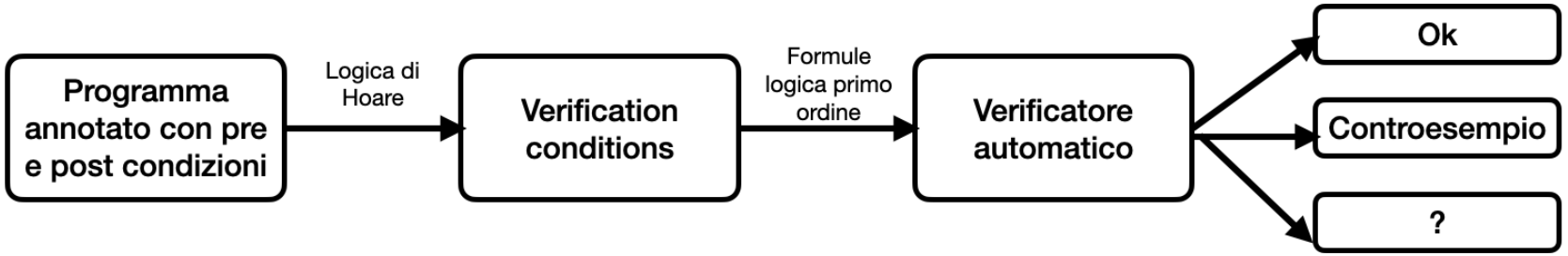


Figure: Processo di verifica

Logica di Floyd-Hoare

La logica di Floyd-Hoare è una logica di programma presentata come un sistema formale con assiomi e regole di inferenza
Il concetto alla base di tutto il sistema formale è quello della tripla di Hoare

Tripla di Hoare

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

dove P è la precondizione, C è il programma e Q è la postcondizione

A partire dalla tripla si definiscono le regole di inferenza e gli assiomi per tutti i costrutti

Assegnamento

$$\frac{}{\{Q[E/x]\}x := E\{Q\}}\ \text{Asgn}$$

Logica di Floyd-Hoare

Conseguenza

$$\frac{P \to P' \qquad \{\text{P'}\}\ \text{C}\ \{\text{Q'}\} \qquad Q' \to Q}{\{\text{P}\}\ \text{C}\ \{\text{Q}\}}\ \text{Conseq}$$

Composizione sequenziale

$$\frac{\{\text{P}\}\text{C1}\{\text{R}\} \qquad \{\text{R}\}\text{C2}\{\text{Q}\}}{\{\text{P}\}\ \text{C1;C2}\ \{\text{Q}\}}\ \text{Seq}$$

Selezione

$$\frac{\{P \wedge b\}C1\{Q\} \qquad \{P \wedge \neg b\}C2\{Q\}}{\{P\}\text{IF b THEN C1 ELSE C2}\{Q\}}\ \text{If}$$

Iterazione

$$\frac{\{P \wedge b\} C \{P\}}{\{P\}\text{WHILE b DO C}\{P \wedge \neg b\}}\ \text{While}$$

Example

Esempio di derivazione

$$\dfrac{\dfrac{x = 0 \rightarrow 2 + 1 = 3 \quad \dfrac{}{\{2 + 1 = 3\} y := 2 \{y + 1 = 3\}}\ \text{Asgn}}{\{x = 0\} y := 2 \{y + 1 = 3\}}\ \text{Asgn} \quad \dfrac{}{\{y + 1\} x := y + 1 \{x = 3\}}\ \text{Asgn}}{\{x = 0\} y := 2; x := y + 1 \{x = 3\}}\ \text{Conseq}$$

Logica di Floyd Hoare

L'utilizzo della logica di Floyd-Hoare nella verifica funzionale risente di tre problemi

1. l'assenza di una strategia esplicita per costruire una derivazione
2. l'obbligo di dover dimostrare implicazioni logiche nella regola della conseguenza
3. l'assenza di supporto alla *modularità* del ragionamento se si introduce lo *heap*

Predicate transformer sematics

- La semantica dei *predicate transformer* è un'idea introdotta da Dijkstra
- Definiscono la semantica di un linguaggio di programmazione assegnando ad ogni comando del linguaggio una funzione totale tra due predicati sullo spazio degli stati dei comandi
- Sono una riformulazione della logica di Hoare in delle strategie complete per costruire derivazioni valide
- Forniscono un algoritmo per ridurre il problema di verificare una tripla di Hoare nella verifica di una formula in logica del primo ordine
- Intuitivamente, fanno un esecuzione simbolica dei comandi trasformandoli in predicati
- Ne esistono due tipi
  - la weakest precondition *wp*
  - la strongest postocondition *sp*

Predicate transformer semantics

Definizione *wp*

Dato un comando C e una postcondizione Q, un predicato è la *weakest precondition* se

- la tripla [wp(C,Q)]C[Q] è valida
- per ogni P tale per cui [P]C[Q] è valida allora P $\Longrightarrow$ wp(C,Q)

Si definisce una regola per il calcolo di *wp* per ogni costrutto del linguaggio

wlp per l'assegnamento

wlp(x:=e, Q) = Q[x/e]

wlp per la composizione sequenziale

wlp(C1;C2, Q) = *wlp*(C1, *wlp*(C2,Q))

Predicate transformer semantics

wlp per la selezione

wlp(IF b THEN C1 ELSE C2, Q) $=$ (b $\Longrightarrow$ *wlp*(C1,Q)) $\wedge$ ($\neg$b $\Longrightarrow$ *wlp*(C2,Q))

wlp per l'iterazione

wlp(WHILE {I} b DO C, Q) $=$ I, dove I rappresenta l'invariante

Per verificare che $\models\{P\}C\{Q\}$ quindi

1. si calcola *wp*(C,Q)
2. si verifica che l'implicazione P $\Longrightarrow$ *wp*(C,Q) sia **valida**

Frame problem

Frame problem

Descrivendo formalmente i cambiamenti in un sistema, come specificare quali parti dello stato del sistema non sono influenzate dal cambiamento?

- In un contesto modulare è un problema particolarmente complesso

Frame problem

```
1  class Node {
2    var v:int;
3    var next:Node;
4  }
5  class List {
6    var c: Node;
7
8    constructor()
9      ensures len() == 0
10   {c := null;}
11
12   function len() returns int
13   {len_aux();}
14
15   function len_aux(p:Node) returns int
16   {
17     p = null ? 0 : 1+len_aux(p.next)
18   }
19 }
```

Frame problem

```
1 var A,B : List;
2 A := new List;
3 B := new List;
4 assert A.len() == 0;
```

- Dimostrare che l'asserzione è vera è impossibile
- Il costruttore assicura che *len() == 0* ma non garantisce nulla rispetto a ciò che potrebbe succedere durante *B := new List;* (ad esempio modifiche a *A.c*)
- Non c'è un modo per esprimere che "nessuna variabile del client è interessata dalla modifica" perché le variabili del client sono sconosciute
- Non è possibile esprimere il fatto che solo il campo *c* del nuovo oggetto è modificato, il client non conosce i dettagli implementativi interni
- Non c'è un modo per esprimere la presenza o l'assenza di *abstract aliasing*

Dynamic Frames

- L'idea alla base dei dynamic frames è quella dell'introduzione dei *footprint* di metodi e funzioni
- Il *footprint* rappresenta l'insieme di campi che un un metodo o una funzione può leggere o modificare (intuitivamente, l'insieme di locazioni di memoria da cui dipende nel calcolo)
- Nel caso di un metodo si introduce la keyword *modifies*, nel caso di una funzione la keyword *reads*

Dynamic frames

Un *dynamic frame* è una funzione pura la cui valutazione produce un insieme di campi (una **regione**)

Dynamic Frames

- Con l'aggiunta del *footprint* è ora possibile verificare l'asserzione mostrata in precedenza, è sufficiente garantire che il *footprint* del metodo *len()* è disgiunto da quello del costruttore di *B*
- Allo stesso modo è ora possibile per esempio esprimere proprietà come l'assenza di cicli all'interno della lista, proprietà che prima erano inesprimibili
- L'aggiunta dei *footprint* però non è sufficiente, sono necessarie altre due convenzioni
  - *swinging pivots*
  - *self framing*

Swinging pivots

Sia S l'insieme di dynamic frames presenti nel *footprint* di un metodo. Il valore di qualsiasi dynamic frame in S può essere aumentato solo da locazioni presenti in qualche altro dynamic frame in S o da posizioni appena allocate

```
1  // Nel footprint e' presente solo repr(): il suo valore
2  // puo' essere incrementato solo da nuove locazioni
3  method insert(x:int)
4      modifies repr()
5  // Nel footprint sono presenti repr() e p.repr().
6  // repr() puo' essere aumentato solo da locazioni
7  // precedentemente in p.repr() o da nuove locazioni
8  method prepend(p:List)
9      reads p.repr()
10     modifies repr()
```

Per esprimere questa convenzione si fa ricorso alla keyword **fresh**

Self framing

Self framing

Dynamic frames che sono sconosciuti ad un metodo *m* e che sono disgiunti dal suo footprint, non devono cambiare quando *m* è invocato: il footprint di un dynamic frame deve essere il dynamic frame stesso

```
1 function rep() returns reg
2     reads rep();
3 { rep_aux(); }
4
5 function rep_aux(p: Node) returns reg
6     reads rep_aux()
7 { p = null? {} : {p.v, p.n} + rep_aux(p.n); }
```

Introduzione a Dafny

- Il linguaggio è stato ideato da Rustan Leino durante il suo lavoro presso Microsoft Research
- Supporta sia il paradigma imperativo che quello funzionale
- La specifica formale viene formulata al suo interno attraverso pre-post condizioni, invarianti di ciclo, metriche per la terminazione e una implementazione dei dynamic frames (il nome Dafny nasce dalla permutazione di alcune lettere di *dynamic frames*)
- Il supporto per l'interazione con l'utente è quasi inesistente fatto salve per l'istruzione *print*

Figure: Rustan Leino

Architettura del sistema

- In questa presentazione si fa riferimento alla versione 3.12 del linguaggio
- I file Dafny hanno estensione *.dfy*
- L'installazione di Dafny in realtà non installa solo il linguaggio col suo compilatore ma l'intero sistema sottostante dedicato alla verifica

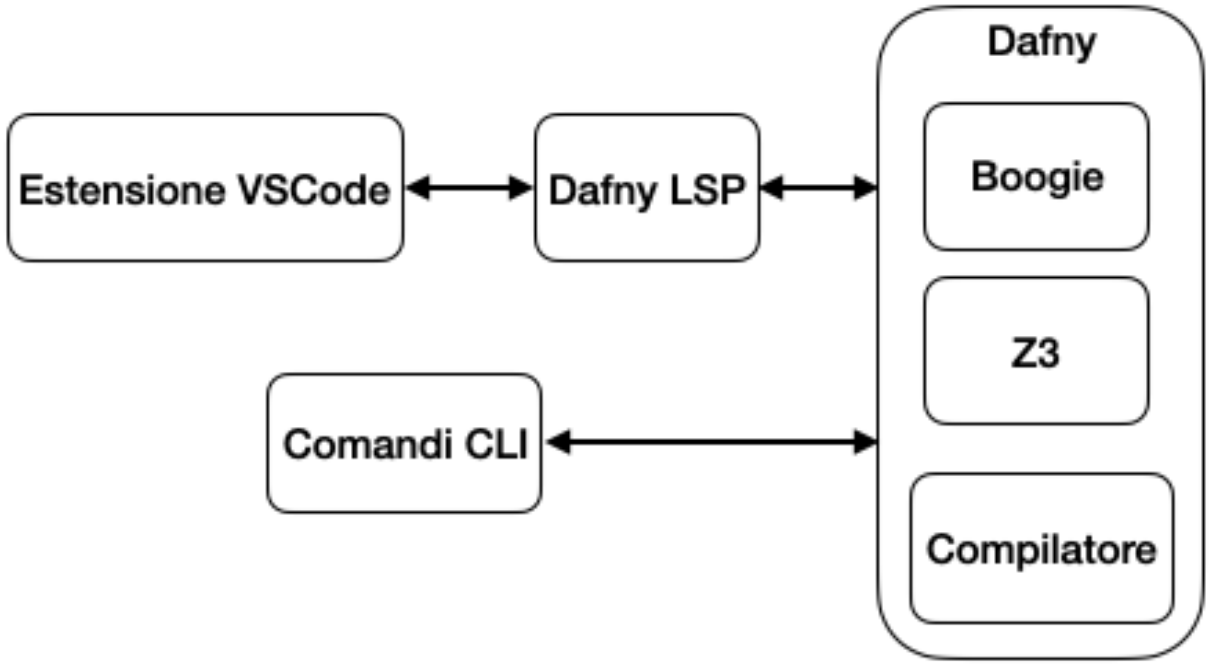


Figure: Componenti del sistema Dafny

Architettura del sistema

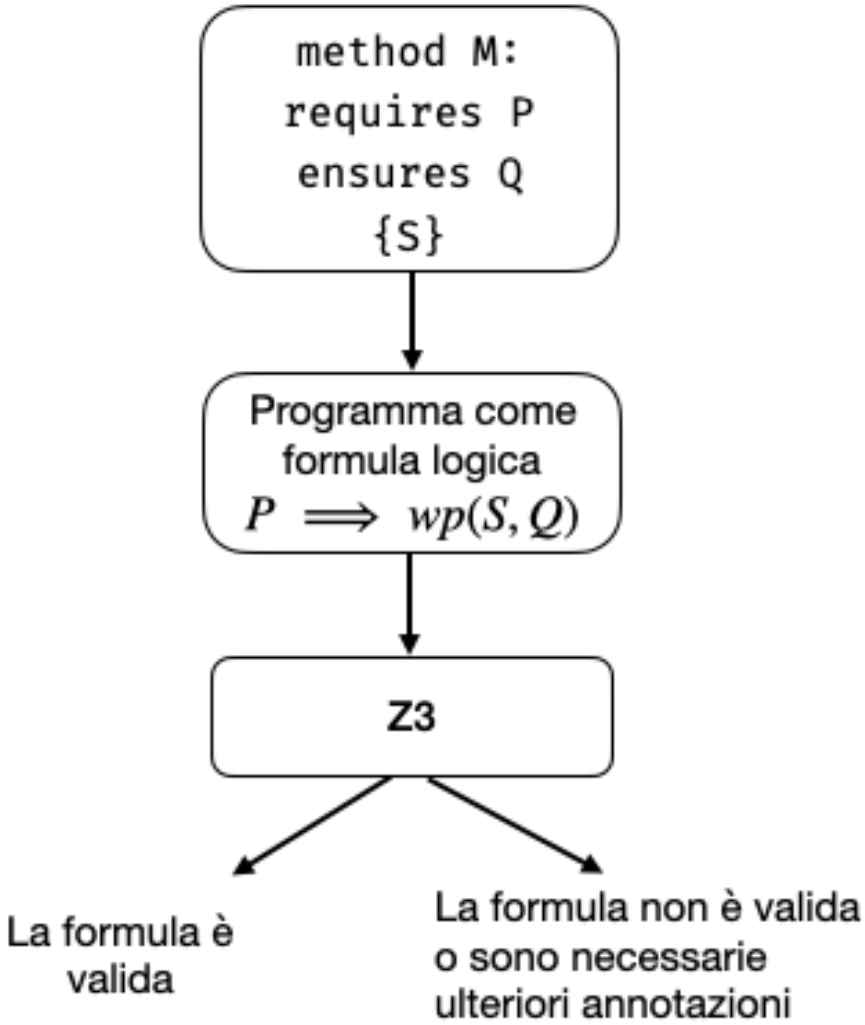


Figure: Processo di verifica

Boogie

- Boogie è un linguaggio intermedio di verifica creato da Microsoft Research
- È progettato per essere un layer intermedio per la costruzione di verificatori di programmi per altri linguaggi (VCC, Dafny, Chalice)
- Ci sono sostanzialmente due motivi per utilizzare un linguaggio intermedio
  - lo sviluppo del linguaggio "front-end" è indipendente dalla metodologia di verifica
  - non è necessario che il verificatore sia in grado di comprendere la semantica di linguaggi diversi, è sufficiente che comprenda Boogie

Boogie

- La sintassi e le caratteristiche di Boogie sono altamente tecniche, per comprenderne ad alto livello il funzionamento si riporta un esempio

```
1  int BinarySearch(int[] a, int len, int key)
2  // precondizione: 0 <= len <= |a| and forall i :: 0 <= i <= len - 2 and
3  // forall j :: i + 1 <= j <= len - 1 ==> a[i] < a[j]
4  {
5      int low = 0;
6      int high = len;
7      while (low < high)
8      // invariante: 0 <= low <= high <= len <= |a| and
9      // forall i :: 0 <= i <= low - 1 ==> a[i] < key and
10     // forall i :: high <= i <= len - 1 ==> a[i] > key
11     {
12         //Ricerca dell'elemento mediano
13         int mid = low + (high - low) / 2;
14         int val = a[mid];
15         if (key < val) {
16             low = mid + 1;
17         } else if (val < key) {
18             high = mid;
19         } else {
20             return mid;
21         }
22     }
23     return -1;
24 }
25 // postcondizione: (0 <= result ==> a[result] = key) and
26 // (result < 0 ==> forall i :: 0 <= i <= len - 1 ==> a[i] != key)
```

Boogie

- La semantica di ogni costrutto del linguaggio "ad alto livello" è definita in termini del linguaggio intermedio
- Ad esempio,

```
1  // if (cond) S; else T;
2  {assume cond; S;} [] {assume !cond; T;}
3  // while (cond) inv S[x,y] dove x e y sono le variabili
       di programma modificate dal loop
4  assert inv; // controllo invariante in entrata
5  havoc x,y;  // salto ad una interazione arbitraria
6  assume inv;
7  {
8      assume guard;
9      S;
10     assert inv; // controllo che l'invariante sia
       mantenuta
11     assume false;
12     []
13     assume !guard; //uscita dal loop
14 }
```

Boogie

```
1  int BinarySearch(int[] arr, int len, int key){
2      l1: assume pre
3      l2: int low = 0;
4      l3: int high = len;
5      l4: assert inv;
6      l5: havoc low, high;
7      l6: assume inv;
8      l7: {
9          m1: assume (low < high);
10         // Ricerca dell'elemento mediano
11         int mid = low + (high - low) / 2;
12         int val = a[mid];
13         m2: {
14             n1: assume (key < val);
15             n2: low = mid + 1;
16             []
17             n3: assume !(key < val);
18             n4: assume (val < key);
19             n5: high = mid;
20             []
21             n6: assume !(key < val);
22             n7: assume !(val < key);
23             n8: assert post[result := mid]
24             n9: assume false
25         }
26         m3: assert inv:
27         m4: assume false;
28         []
29         m5: assume !(low < high);
30     }
31     l8: assert post[result := -1]
32 }
```

Boogie

$$
\begin{aligned}
& wp(\texttt{BinarySearch}_C(\texttt{a}, len, \texttt{key}), \textbf{true}) \\
\equiv\ & \{\text{expand } \texttt{BinarySearch}_C\} \\
& wp(\ell_1; \ell_2; \ell_3; \ell_4; \ell_5; \ell_6; \ell_7; \ell_8, \textbf{true}) \\
\equiv\ & \{\ell_1 : \texttt{assume } pre\} \\
& pre \implies wp(\ell_2 \ell_3; \ell_4; \ell_5; \ell_6; \ell_7; \ell_8, \textbf{true}) \\
\equiv\ & \{\ell_2 : \texttt{int } low = 0;\} \\
& pre \implies \textbf{let } low = 0\ wp(\ell_3; \ell_4; \ell_5; \ell_6; \ell_7; \ell_8, \textbf{true}) \\
\equiv\ & \{\ell_3 : \texttt{int } high = len;\} \\
& pre \implies \textbf{let } low = 0, high = len\ wp(\ell_4; \ell_5; \ell_6; \ell_7; \ell_8, \textbf{true}) \\
\equiv\ & \{\ell_4 : \texttt{assert } inv\} \\
& pre \implies \textbf{let } low = 0, high = len\ inv \wedge wp(\ell_5; \ell_6; \ell_7; \ell_8, \textbf{true}) \\
\equiv\ & \{\ell_5 : \texttt{havoc } low, high\} \\
& pre \implies \textbf{let } low = 0, high = len\ inv \wedge \forall low, high\ .\ wp(\ell_6; \ell_7; \ell_8, \textbf{true}) \\
\equiv\ & \{\ell_6 : \texttt{assume } inv\} \\
& pre \implies \textbf{let } low = 0, high = len\ inv \wedge \forall low, high\ .\ inv \implies wp(\ell_7; \ell_8, \textbf{true}) \\
\equiv\ & \{\ell_8 : \texttt{assert } post[result \leftarrow -1]\} \\
& pre \implies \textbf{let } low = 0, high = len\ inv \wedge \forall low, high\ .\ inv \implies wp(\ell_7, post[result \leftarrow -1])
\end{aligned}
$$

$$
\begin{array}{ll}
 & wp(\ell_7, post[result \leftarrow -1]) \\
\equiv & \{\ell_7 : \{m_1; m_2; m_3; m_4\} \; [] \; m_5\} \\
 & wp(\{m_1; m_2; m_3; m_4\} \; [] \; m_5, post[result \leftarrow -1]) \\
\equiv & \{\text{by semantics of } []\} \\
 & wp(m_1; m_2; m_3; m_4, post[result \leftarrow -1]) \wedge wp(m_5, post[result \leftarrow -1]) \\
\equiv & \{m_5 : \texttt{assume } \neg(low < high)\} \\
 & wp(m_1; m_2; m_3; m_4, post[result \leftarrow -1]) \wedge (\neg(low < high) \implies post[result \leftarrow -1]) \\
\equiv & \{m_4 : \texttt{assume } \mathbf{false}\} \\
 & wp(m_1; m_2; m_3, \mathbf{true}) \wedge (\neg(low < high) \implies post[result \leftarrow -1]) \\
\equiv & \{m_3 : \texttt{assert } inv\} \\
 & wp(m_1; m_2, inv) \wedge (\neg(low < high) \implies post[result \leftarrow -1]) \\
\equiv & \{m_1 : \texttt{assume } low < high\} \\
 & (low < high \implies wp(m_2, inv)) \wedge (\neg(low < high) \implies post[result \leftarrow -1]) \\
\equiv & \{\text{by a full unfolding of } wp(m_2, inv)\} \\
 & \left(\begin{array}{l} low < high \implies \\ \quad \mathbf{let}\ mid = low + (high - low)/2 \\ \quad \mathbf{let}\ val = \mathtt{a}[mid] \\ \qquad (\mathtt{key} < val \implies inv[low \mapsto mid + 1]) \wedge \\ \qquad (\neg(\mathtt{key} < val) \wedge (val < \mathtt{key}) \implies inv[high \mapsto mid]) \wedge \\ \qquad (\neg(\mathtt{key} < val) \wedge \neg(val < \mathtt{key}) \implies post[result \mapsto mid]) \end{array} \right) \wedge \\
 & (\neg(low < high) \implies post[result \leftarrow -1])
\end{array}
$$

Boogie

$$
\begin{array}{l}
pre \implies \\
\textbf{let } low = 0 \\
\textbf{let } high = len \\
inv \wedge \\
\left(\begin{array}{l}
\forall low, high \ . \ inv \implies \\
\quad \neg(low < high) \implies post[result \mapsto -1] \\
\quad \wedge \left(\begin{array}{l}
low < high \implies \\
\quad \textbf{let } mid = low + (high - low)/2 \\
\quad \textbf{let } val = \mathtt{a}[mid] \\
\quad (\mathtt{key} < val \implies inv[low \mapsto mid + 1]) \wedge \\
\quad (\neg(\mathtt{key} < val) \wedge (val < \mathtt{key}) \implies inv[high \mapsto mid]) \wedge \\
\quad (\neg(\mathtt{key} < val) \wedge \neg(val < \mathtt{key}) \implies post[result \mapsto mid])
\end{array} \right)
\end{array} \right)
\end{array}
$$

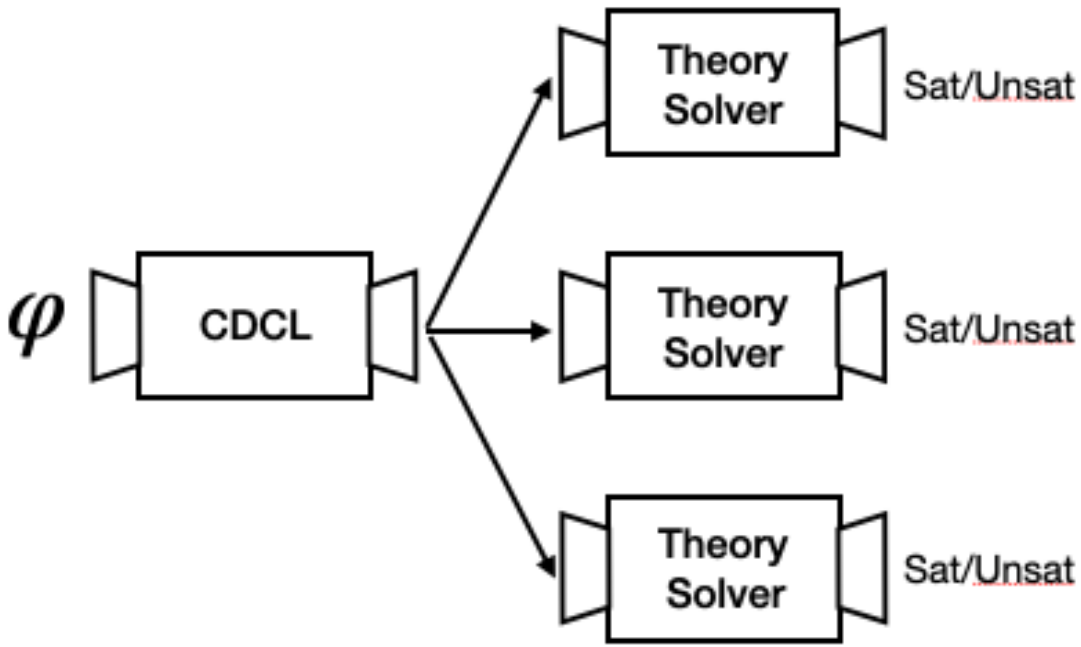
- Questa formula rappresenta il programma sottoforma di formula logica che deve essere risolta dall'SMT solver

Z3

- Z3 è un SMT solver progettato dal gruppo RiSE (Research in Software Engineering) di Microsoft Research
- È stato sviluppato per risolvere problemi della verifica di programmi
- Tra le teorie supportate sono presenti
  - aritmetica lineare
  - array, liste
  - funzioni non interpretate
  - uguaglianze
  - quantificatori

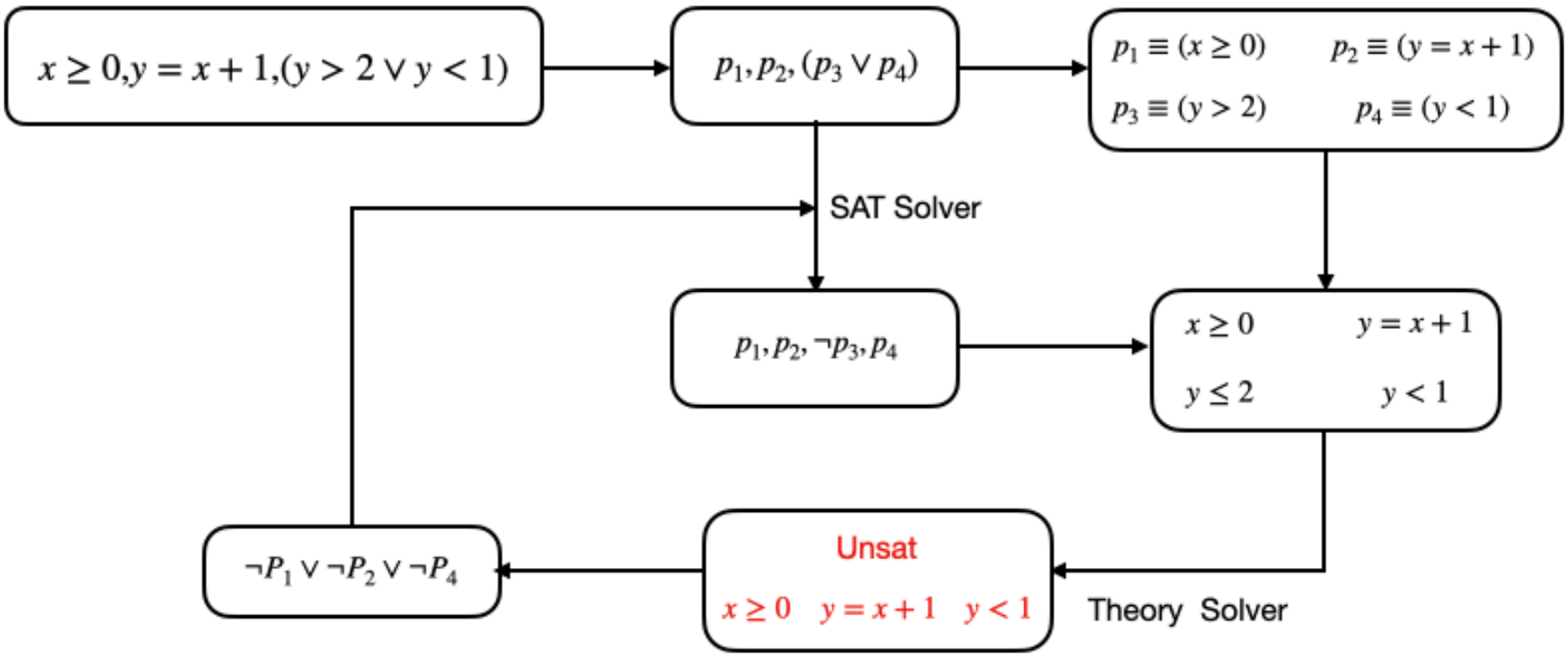
SMT solver per la verifica funzionale

- L'obiettivo è stabilire la validità di una formula ϕ
- Il problema della dimostrazione di validità di una formula si può ridurre al problema di dimostrare la soddisfaciblità di $\neg\phi$
- Un SMT solver estende il problema della soddisfacibilità di una formula in logica proposizione (SAT problem) ad una formula *FOL* attraverso le *teorie*



SMT solver per la verifica funzionale

- Intuitivamente il funzionamento può essere schematizzato nel seguente modo



Un semplice esempio

Example

1. Si supponga di partire dalla congiunzione delle seguenti formule:
   $f(f(x) - f(y)) = a$
   $f(0) = a + 2$
   $x = y$
2. La formula contiene sia aritmetica lineare che la teoria delle funzioni non interpretate, vanno separate attraverso la *purificazione*
   $f(e_1) = a$
   $e_1 = f(x) - f(y)$
3. A sua volta e_1 può ulteriormente essere scomposta in
   $e_1 = e_2 - e_3$
   $e_2 = f(x)$
   $e_3 = f(y)$

Un semplice esempio

Example

4 Lo stesso procedimento si applica per $f(0) = a + 2$
   $f(e_4) = e_5$
   $e_4 = 0$
   $e_5 = a + 2$
5 Ora le formule sono scomposte in due teorie:
   - Quelle nella teoria delle funzioni non interpretate
     $f(e_1) = a$
     $e_2 = f(x)$
     $e_3 = f(y)$
     $f(e_4) = e_5$
     $x = y$
   - Quelle nella teoria dell'aritmetica
     $e_1 = e_2 - e_3$
     $e_4 = 0$
     $e_5 = a + 2$
     $x = y$

Un semplice esempio

Example

6. Considerando le formule nella teoria delle funzioni non interpretate, nella teoria è contenuta una regola che afferma che se $x = y$ allora $f(x) = f(y)$. Applicando la regola si ottiene $f(x) = f(y)$ e siccome $f(x) = e_2$ e $f(y) = e_3$ allora $e_2 = e_3$
7. L'uguaglianza $e_2 = e_3$ viene aggiunta all'insieme di formule della teoria dell'aritmetica
8. A questo punto il risolutore della teoria dell'aritmetica può scoprire che $e_2 - e_3 = 0$ e che quindi $e_1 = e_4$
9. Questa scoperta viene restituita al risolutore della teoria delle funzioni non interpretate da cui si ottiene che $a = e_5$

Un semplice esempio

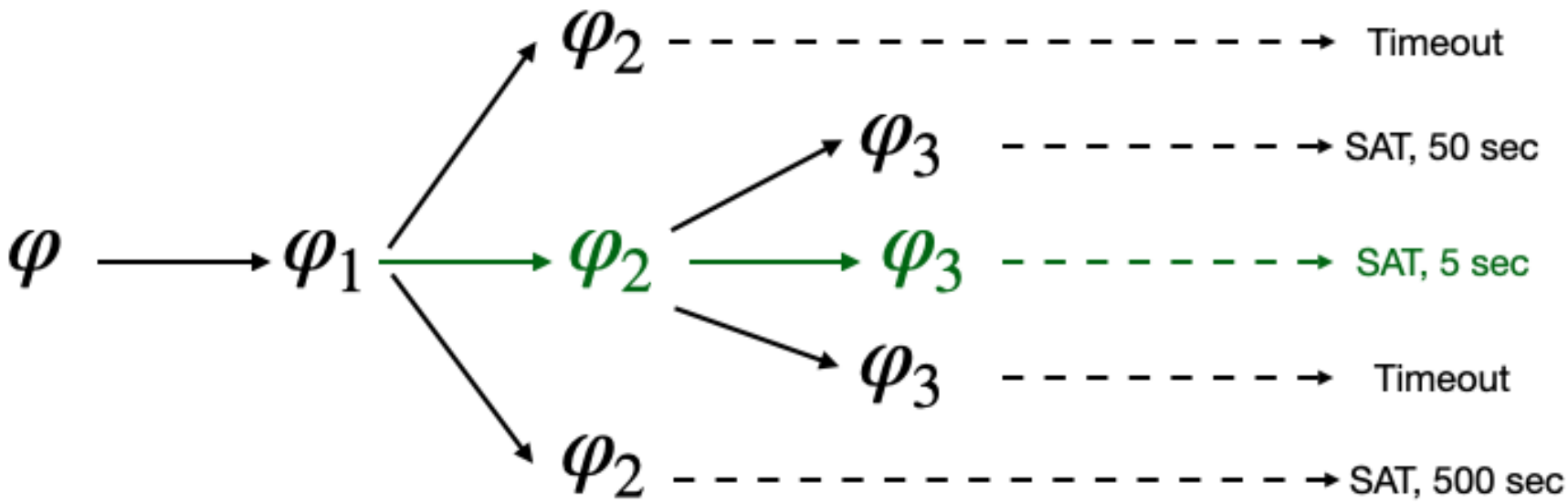
Example

10 L'insieme finale di vincoli è il seguente (interamente nella teoria dell'aritmetica)

- $e_1 = e_2 - e_3$
- $e_4 = 0$
- $e_5 = a + 2$
- $x = y$
- $e_2 = e_3$
- $a = e_5$

11 $a = e_5$ e al tempo stesso $e_5 = a + 2$, da cui si conclude che la formula originaria è insoddisfacibile

SMT solver per la verifica funzionale



Anatomia di un programma Dafny

Un programma Dafny è composto da 4 "blocchi" fondamentali

- Funzioni
- Predicati
- Metodi
- Classi

Funzioni

```
function Nome<T>(a: A, b: B): T
    requires _precondizione_
    reads _frame di memoria_
    ensures _postcondizione_
    decreases _metrica di terminazione_
{
    Corpo
}
```

- Le funzioni sono *ghost* di default a meno che non vengano definite come *function method*
- Sono funzioni nel senso matematico, non possono avere *side-effects*

Predicati

```
1 predicate sorted(a: array<int>)
2     reads a
3 {
4 forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]
5 }
```

- Sono identici alle funzioni ma possono ritornare esclusivamente un valore booleano

Metodi

```
method Nome<T>(a: A, b: B) returns (x: X, y: Y)
    requires _precondizione_
    modifies _frame di memoria_
    ensures _postcondizione_
    decreases _metrica di terminazione_
{
    Corpo
}
```

- Vari tipi di metodi (costruttori, lemmi, lemmi *twostate*..)
- Può essere reso *ghost* attraverso la dichiarazione *ghost method*
- Non è necessario *return esplicito*, i parametri in input sono immutabili

Classi

```
class Nome
{
    var nome: tipo
    constructor(x: tipo)
        ensures _postcondizione_
    {
        Corpo
    }
    predicate Valid()
        reads _frame di memoria_
    {
        Corpo
    }
    method NomeMetodo(y: tipo)
        requires _precondizione_
        modifies _frame di memoria_
        ensures _postcondizione_
        decreases _metrica di terminazione_
    {
        Corpo
    }
}
```

- Identiche ad altri linguaggi di programmazione (ad esempio Java) fatta eccezione per il *subclassing*

Keyword per la specifica funzionale

- Il supporto alla verifica funzionale è reso possibile dalle *keyword* riservate alla specifica del comportamento:
  - *requires*
  - *ensures*
  - *decreases*
  - *invariant*
  - *assert*
  - *assume*
  - *reads*
  - *modifies*

requires

- Rappresenta la precondizione
- Se viene omessa, si assume *true*
- Se la precondizione è particolarmente lunga è possibile dividerla in più *requires* diverse che vengono considerate come se fossero in congiunzione tra di loro
- Il verificatore controlla che la precondizione sia soddisfatta ad ogni chiamata

```
1 method FindMax(a: array<int>) returns (i:int)
2     requires a.Length >= 1
```

ensures

- Simmetrica a *requires*, rappresenta la postcondizione

```
1 method Find(a:array<int>, key:int) returns (index:int)
2     ensures 0 <= nidex ==> index < a.Length &&
3     a[index] == key
4     ensures index < 0 ==> forall k ::
5     0 <= k < a.Length ==> a[k] != key
```

decreases

- La keyword *decreases* è dedicata alla terminazione
- L'idea è quella di annotare ogni iterazione di un ciclo (o ogni chiamata ricorsiva) con un valore per cui esista una relazione d'ordine (ossia per cui non esistono catene discendenti infinite) e assicurarsi che iterazioni successive decrementino l'etichetta
- L'etichetta prende il nome di *variant*
- Si faccia riferimento al seguente esempio di una funzione che calcola la somma di una lista di numeri interi

```
1 function Sum(xs: seq<int>): int
2     decreases xs;
3 {
4     if xs == [] then 0 else xs[0] + Sum(xs[1..])
5 }
```

invariant

- Rappresenta il concetto di invariante per un ciclo
- Esattamente come da definizione, deve essere un'asserzione valida rispettivamente
  - subito prima dell'ingresso nel ciclo
  - alla fine dell'esecuzione del corpo del ciclo
  - all'uscita dal ciclo

assert

- Utilizzata principalmente in fase di *debug* per sincerarsi che certe proprietà che sono "evidenti" per l'utente siano dimostrabili anche dal verificatore
- Sono *ghost statements*
- Talvolta sono necessarie per guidare il processo di verifica

assume

- Utilizzata durante la costruzione di una prova
- Permette la specifica di una formula che il verificatore assumerà come vera senza la necessità di una prova
- Utile per rimandare la verifica di sottoproblemi nell'ambito di una prova più grande
- Un programma contente *assume* non può essere compilato

Un semplice esempio: successione di Fibonacci

- Come primo semplice programma consideriamo l'implementazione di un metodo che calcoli l'n-esimo numero della successione di Fibonacci
- La definizione matematica è $F_n = F_{n-1} + F_{n-2}$ con $F_1 = F_2 = 1$ e $F_0 = 0$.
- Implementarla direttamente in questo modo avrebbe complessità esponenziale
- L'idea è quella di utilizzare un contatore e calcolare ripetutamente coppie adiacenti di numeri della sequenza fino a quando non viene raggiunto il numero desiderato

Un semplice esempio: BinarySearch

- L'algoritmo di ricerca binaria trova l'indice in cui è presente una chiave all'interno di un array ordinato in tempo logaritmico nel caso peggiore

Algorithm 1 BinarySearch(A[0..n], key)

Require: L'array A è ordinato in ordine non decrescente
Ensure: Se *key* è contenuto in A restituisce l'indice della sua posizione, -1 altrimenti

```
low ← 0
high ← n
while low < high do
                    ▷ Invariant: Se la chiave è in A[0..n] allora la chiave è in A[low..high]
    mid ← (low + high)/2
    if key > A[mid] then
        low ← mid + 1
    else if key < a[mid] then
        high ← mid
    else
        return mid
    end if
end while
return −1
```

Implementazione dei dynamic frames

- Per mostrare l'implementazione di strutture dati è necessario approfondire l'implementazione del formalismo dei dynamic frames
- In Dafny i dynamic frames sono implementati attraverso l'uso di campi *ghost* e delle keyword *reads* e *modifies*
- Il *footprint* di un metodo viene rappresentato attraverso variabili *ghost*
- Senza entrare nei dettagli, concretamente la specifica di programmi facenti uso dello *heap* è estremamente **idiomatica**

Dynamic frames in Dafny

Representation set

Ogni oggetto composto ha al suo interno una variabile *ghost* che rappresenta l'insieme di oggetti contenuti al suo interno.

```
1 ghost var Repr: set<object>
```

Invariante di struttura

È un predicato solitamente chiamato *Valid* che cattura tutte le proprietà che devono essere vere affinché l'oggetto in questione sia valido

```
1 ghost predicate Valid()
2     reads this, Repr
3     ensures Valid() ==> this in Repr
4 {
5     this in Repr && ..
6 }
```

Dynamic frames in Dafny

Costruttore

Crea e inizializza un nuovo oggetto

```
1 constructor()
2     ensures Valid() && fresh(Repr)
```

Funzioni

Una funzione non può avere *side effects* (non può modificare la memoria)

```
1 functin Fun(a:A):B
2     requires Valid()
3     reads Repr
```

Metodi

Un metodo a differenza di una funzione pùo sia leggere che scrivere in memoria

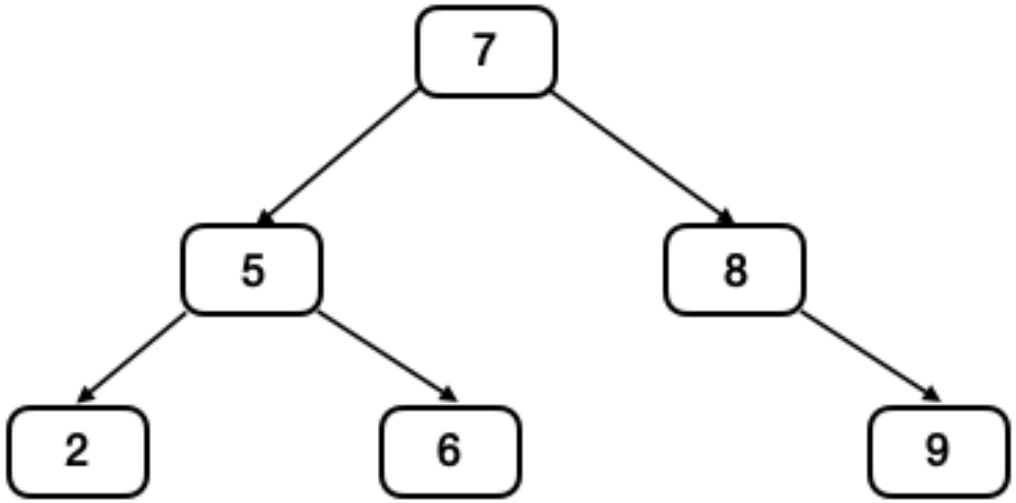
```
1 method Met(a:A) returns (b:B)
2     requires Valid()
3     modifies Repr
4     ensures Valid() && fresh(Repr - old(Repr))
```

Il predicato *Valid()* è un invariante perché viene utilizzato come tale

- Esiste una *feature* del linguaggio che si utilizza con {*:autocontracs*} che permette di ridurre la quantità di codice *boilerplate*

Albero binario di ricerca

- Per vedere un esempio di utilizzo dei dynamic frames si illustra l'implementazione di un albero binario di ricerca



Considerazioni finali

- L'impiego di un SMT solver rende il processo di verifica opaco
- "Effetto farfalla" durante la prova
- Un SMT solver non è un oracolo, i limiti al calcolo rimangono
- Alcuni limiti del calcolo possono essere superati da accortezze nel linguaggio (quantificatori)
- Il linugaggio può essere utilizzato anche come *proof assistant* attraverso lemmi e *calc*