

Metodi formali dell'Informatica

Introduzione alla semantica ed alla
logica dei programmi

a.a. 2023-24

Ugo de'Liguoro

Sommario

- Cosa sono e a cosa servono i metodi formali
- Il problema della verifica
- La formalizzazione delle proprietà dei programmi
- Analisi statica dei programmi
- Strumenti automatici e semiautomatici
- Perché Agda?
- Programma del corso

Cosa sono e a cosa servono i metodi formali

Cosa sono i “metodi formali”?

The screenshot shows the homepage of the NASA Langley Formal Methods website. At the top left is the NASA logo. To its right, the text "National Aeronautics and Space Administration" is displayed. On the far right, there is a link "+ Contact NASA". Below this header is a navigation bar with six items: "+ ABOUT NASA", "+ NEWS", "+ MISSIONS", "+ MULTIMEDIA", "+ CONNECT", and "+ ABOUT NASA". The main content area features a large banner image showing a robot and an astronaut shaking hands, with the International Space Station and two aircraft in the background. To the left of the banner, there is a logo consisting of three stylized letters (L, F, M) in red, purple, and blue, with the text "LANGLEY FORMAL METHODS" below it. Below the banner is another navigation bar with seven items: "+ HOME", "+ WELCOME", "+ QUICK PAGE", "+ PHILOSOPHY", "+ TEAM", "+ RESEARCH", and "+ LINKS".

WHAT IS FORMAL METHODS?

"Formal Methods" refers to mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems. The phrase "mathematically rigorous" means that the specifications used in formal methods are well-formed statements in a mathematical logic and that the formal verifications are rigorous deductions in that logic (i.e. each step follows from a rule of inference and hence can be checked by a mechanical process.) The value of formal methods is that they provide a means to symbolically examine the entire state space of a digital design (whether hardware or software) and establish a correctness or safety property that is true for all possible inputs. However, this is rarely done in practice today (except for the critical components of safety critical systems) because of the enormous complexity of real systems. Several approaches are used to overcome the astronomically-sized state spaces associated with real systems:

- Apply formal methods to requirements and high-level designs where most of the details are abstracted away
- Apply formal methods to only the most critical components
- Analyze models of software and hardware where variables are discretized and ranges drastically reduced.
- Analyze system models in a hierarchical manner that enables "divide and conquer"
- Automate as much of the verification as possible

Cosa sono i “metodi formali”?

- “In computer science **formal methods** are a particular kind of **mathematically based techniques** for the specification, **development and verification** of software and hardware systems” (Wikipedia)

Cosa sono i “metodi formali”?

- Calcoli logici
- Sistemi di riscrittura
- Linguaggi formali
- Teoria degli automi
- Sistemi di transizione
- Algebre dei dati
- Algebre dei processi
- Algebra relazionale
- Semantica dei linguaggi di programmazione
- Teoria dei tipi
- Analisi statica - Data Flow, Control Flow e Abstract Interpretation
- ...

A cosa servono?

- “The use of formal methods for software and hardware design is motivated by the expectation that performing appropriate mathematical analysis can contribute to the reliability and robustness of a design” (Wikipedia)

A cosa servono?

- Verifica di software e hardware
- Documentazione, specifica e sviluppo del software
- Analisi di protocolli
- Debugging
- Ottimizzazione del codice
- Trasformazione di programmi
- Monitoring
- Sicurezza
- ...

Verifica formale del software

 About Us How to Apply Research Groups Conferences Advanced Schools Bruno Awards Virtual Bookshelf Gallery

[Home](#) >> The 2nd Winter School in Engineering and Computer Science on Formal Verification

The 2nd Winter School in Engineering and Computer Science on Formal Verification

Event date: December 17 - December 21, 2017

Organizers:
Orna Kupferman (The Hebrew University)
Moshe Vardi (Rice University)

General Director: Michael Rabin

Safety-critical computers increasingly affect nearly every aspect of our lives. Computers control the planes we fly on, monitor our health in hospitals and do our work in hazardous environments. Computers with software deficiencies have resulted in catastrophic failures. The goal of formal verification is to improve the safety and reliability of such hardware and software systems.

Formal Verification is the study of algorithms and structures applicable to the verification of hardware and software designs. It draws upon ideas and results from logic, graph theory, and automata theory, and combines theoretical and experimental aspects. Thirty years ago this was a subject of academic interest only, today, many companies use formal verification as an integral part of the development process.

The IIAS Winter School on Formal Verification will bring together leading experts in the field to discuss the mathematical and algorithmic foundations of the field, as well as to discuss its application in industry, and its impact on related areas in computer science.

- [Home](#)
- [Registration Form](#)



Formal Verification

The 2nd School in Computer Science and Engineering

Organizers:
Orna Kupferman, The Hebrew University
Moshe Vardi, Rice University

Formal Verification is the study of algorithms and structures applicable to the verification of hardware and software designs, with the goal of improving the safety and reliability of hard and software systems. It draws upon ideas and results from logic, graph theory, and automata theory, and combines theoretical and experimental aspects. Thirty years ago this was a subject of academic interest only, today, many companies use formal verification as an integral part of the development process.

The IIAS Winter School on Formal Verification will bring together leading experts in the field to discuss the mathematical and algorithmic foundations of the field, as well as to discuss its application in industry, and its impact on related areas in computer science.

For more details: ias.huji.ac.il/CSE2

General Director: Michael Rabin, The Hebrew University

Speakers:
Thomas A. Henzinger
IST, Austria
Bernd Finkbeiner
University of Saarland,
Germany
Javier Esparza
TU Munich, Germany
Oren Geffen
Technion, Israel
Christel Baier
TU Dresden, Germany
Cindy Doser
IBM, Israel

Shlomit Schwartzblat
Mellonox, Israel
Gila Kamhi
Intel, Israel
Danny Dolev
The Hebrew University,
Israel
David Harel
Weizmann Institute,
Israel
Noam Nisan
The Hebrew University,
Israel

Metodi formali, fondamenti

Riscrittura


$$\begin{array}{lcl} \neg\neg A & \longrightarrow & A \\ \neg(A \wedge B) & \longrightarrow & \neg A \vee \neg B \\ \neg(A \vee B) & \longrightarrow & \neg A \wedge \neg B \\ (A \wedge B) \vee C & \longrightarrow & (A \vee C) \wedge (B \vee C) \\ A \vee (B \wedge C) & \longrightarrow & (A \vee B) \wedge (A \vee C) \end{array}$$

$$\begin{aligned} \neg(\neg p \vee q) \vee r &\longrightarrow (\neg\neg p \wedge \neg q) \vee r \\ &\longrightarrow (p \wedge \neg q) \vee r \\ &\longrightarrow (p \vee r) \wedge (\neg q \vee r) \end{aligned}$$



CNF

Riscrittura

$$0 + m \longrightarrow m$$

$$S(n) + m \longrightarrow S(n + m)$$

S = Successore

$$0 \cdot m \longrightarrow 0$$

$$S(n) \cdot m \longrightarrow m + (n \cdot m)$$

Riscrittura

$$\begin{aligned} 3 \cdot 2 &\longrightarrow 2 + (2 \cdot 2) \\ &\longrightarrow 2 + (2 + (2 \cdot 1)) \\ &\longrightarrow 2 + (2 + (2 + (2 + (2 \cdot 0)))) \\ &\longrightarrow 2 + (2 + (2 + 0)) \\ &\xrightarrow{2} S^2(2 + (2 + 0)) \\ &\xrightarrow{2} S^4(2 + 0) \\ &\xrightarrow{2} S^6(0) \equiv 6 \end{aligned}$$

dove ad es. $3 \equiv S(S(S(0))) \equiv S^3(0)$

Logica: sistemi alla Hilbert e Russell



Herbert Simon, Allen Newell

Il programma Logic Theorist dimostrò i primi 36 teoremi dei Principia di Russell e Whitehead

112

MATHEMATICAL LOGIC

[PART I]

- *247. $\vdash : \sim(p \vee q) \supset \sim p \vee q$ $\left[*245, *22 \frac{\sim p}{p} . \text{Syll} \right]$
- *248. $\vdash : \sim(p \vee q) \supset p \vee \sim q$ $\left[*246, *13 \frac{\sim q}{q} . \text{Syll} \right]$
- *249. $\vdash : \sim(p \vee q) \supset \sim p \vee \sim q$ $\left[*245, *22 \frac{\sim p}{p}, \frac{\sim q}{q} . \text{Syll} \right]$
- *250. $\vdash : \sim(p \supset q) \supset \sim p \supset q$ $\left[*247 \frac{\sim p}{p} \right]$
- *251. $\vdash : \sim(p \supset q) \supset p \supset \sim q$ $\left[*248 \frac{\sim p}{p} \right]$
- *252. $\vdash : \sim(p \supset q) \supset \neg p \supset \neg q$ $\left[*249 \frac{\sim p}{p} \right]$
- *2521. $\vdash : \sim(p \supset q) \supset q \supset p$ $\left[*252.17 \right]$
- *253. $\vdash : p \vee q \supset \sim p \supset q$

Dem.

- $\vdash : *212.38. \supset \vdash : p \vee q \supset \sim(\sim p) \vee q : \supset \vdash . \text{Pr.} \text{p}$
- *254. $\vdash : \sim p \supset q \supset p \vee q$ $\left[*214.38 \right]$
- *255. $\vdash : \sim p \supset p \vee q \supset q$ $\left[*253 . \text{Comm} \right]$
- *256. $\vdash : \sim q \supset p \vee q \supset p$ $\left[*255 \frac{q, p}{p, q} . \text{Perm} \right]$

- *26. $\vdash : \sim p \supset q \supset p \supset q \supset q$

Dem.

- $\vdash : *238$ $\vdash : \sim p \supset q \supset \sim p \vee q \supset q \vee q$ $\left[*238 \right] \quad (1)$
- $\vdash : \sim p \vee q \supset \neg q \vee q : \vdash : \sim p \vee q \supset q$ $\left[\text{Taut. Syll} \right] \quad (2)$
- $\vdash : (1). (2). \text{Syll. } \supset \vdash : \sim p \supset q \supset \sim p \vee q \supset q \supset q$ $\left[\vdash : (1). (2). \text{Syll. } \supset \vdash : \sim p \supset q \supset \sim p \vee q \supset q \supset q \right] . \text{Prop}$
- *261. $\vdash : p \supset q \supset \sim p \supset q \supset q$ $\left[*26 . \text{Comm} \right]$
- *262. $\vdash : p \vee q \supset p \supset q \supset q$ $\left[*253.6 . \text{Syll} \right]$
- *2621. $\vdash : p \supset q \supset p \vee q \supset q$ $\left[*262 . \text{Comm} \right]$
- *263. $\vdash : p \vee q \supset \sim p \vee q \supset q$ $\left[*262 \right]$
- *264. $\vdash : p \vee q \supset p \vee \sim q \supset p$ $\left[*263 \frac{q, p}{p, q} . \text{Perm} \right]$
- *265. $\vdash : p \supset q \supset p \supset q \supset \sim p$ $\left[*264 \frac{\sim p}{p} \right]$
- *267. $\vdash : p \vee q \supset q \supset p \supset q$

Dem.

- $\vdash : *254. \text{Syll} \vdash : p \vee q \supset q \supset \sim p \supset q \supset q$ $\left[*254. \text{Syll} \right] \quad (1)$
- $\vdash : *224. \text{Syll} \vdash : \sim p \supset q \supset q \supset p \supset q$ $\left[*224. \text{Syll} \right] \quad (2)$
- $\vdash : (1). (2). \text{Syll. } \supset \vdash . \text{Prop}$

Logica: la deduzione naturale

$$\frac{\frac{[\varphi \wedge \psi]^1}{\frac{[\varphi \wedge \psi]^1}{\psi} \wedge E} \wedge E}{\rightarrow E} \quad \frac{\frac{[\varphi \rightarrow (\psi \rightarrow \sigma)]^2}{\psi \rightarrow \sigma}}{\psi \rightarrow \sigma} \rightarrow E$$
$$\frac{\frac{\sigma}{\varphi \wedge \psi \rightarrow \sigma} \rightarrow I_1}{\varphi \wedge \psi \rightarrow \sigma} \rightarrow I_2$$
$$(\varphi \rightarrow (\psi \rightarrow \sigma)) \rightarrow (\varphi \wedge \psi \rightarrow \sigma)$$



Gerhard Gentzen

λ -calcolo



$M, N ::= x \mid \lambda x.M \mid MN$

Alonzo Church

$$(\beta) : (\lambda x.M)N \longrightarrow M[x := N]$$

$$\underline{n} \equiv \lambda x \lambda y. x^n y \quad \text{es.} \quad \underline{2} \equiv \lambda x \lambda y. x(xy)$$

$$\text{succ} \equiv \lambda zxy. x(zxy), \quad \text{add} \equiv \lambda uv. u \text{ succ } v$$

$$\text{add } \underline{2} \ \underline{3} \xrightarrow{*} \underline{5}$$

YouTube IT

Cerca

Today we're going to talk about one of my favorite topics in Computer Science,

0:00 / 12:40

Subtitled Films

Lambda Calculus - Computerphile

892.348 visualizzazioni • 27 gen 2017

22.287 NON MI PIACE CONDIVIDI SALVA ...

https://www.youtube.com/watch?v=eis11j_iGMs

λ -calcolo tipato

$$\sigma, \tau ::= \alpha \mid \sigma \rightarrow \tau$$

$$\Gamma = x_1 : \sigma_1, \dots, x_n : \sigma_n$$

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma. M : \sigma \rightarrow \tau}$$

$$\frac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

$$\frac{\Gamma \vdash x : \sigma \rightarrow \tau \rightarrow \rho \quad \Gamma \vdash z : \sigma}{\Gamma \vdash xz : \tau \rightarrow \rho}$$

$$\frac{\Gamma \vdash y : \sigma \rightarrow \tau \quad \Gamma \vdash z : \sigma}{\Gamma \vdash yz : \tau}$$

$$\frac{\Gamma \vdash xz(yz) : \rho}{\vdash \lambda x : \sigma \rightarrow \tau \rightarrow \rho \; \lambda y : \sigma \rightarrow \tau \; \lambda z : \sigma. \; xz(yz) : (\sigma \rightarrow \tau \rightarrow \rho) \rightarrow (\sigma \rightarrow \tau) \rightarrow \rho}$$

$$\Gamma = x : \sigma \rightarrow \tau \rightarrow \rho, \; y : \sigma \rightarrow \tau, \; z : \sigma$$



An advanced, purely functional programming language



Haskell B. Curry

Ord=Ordinabile

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort [a | a < x]
        in smallerSorted ++ [x] ++ biggerSorted
```

Il problema della verifica

Verifica: il problema

- **Dati:** una descrizione **concreta di un sistema**, ad esempio il codice di un programma, ed una **specifica** del suo comportamento o di qualche sua proprietà
- **Risultati:** **un'evidenza** del fatto che il codice soddisfa la specifica, oppure un **controesempio**

Verifica di un programma

Programma *fact*:

```
int fact(int x)
{
    int y = 1;
    int z = 0;
    while (z < x) {
        z = z + 1;
        y = y * z;
    }
    return y;
}
```

Specifica *S*:

Se x è un intero ≥ 0 il programma *fact(x)* termina con $y = x!$ (y è il fattoriale di x)

Come posso essere sicuro del fatto che questo programma sia conforme alla specifica **per ogni possibile esecuzione?**



Verifica di un programma

Programma *fact*:

```
int fact(int x)
{
    int y = 1;
    int z = 0;
    while (z < x) {
        z = z + 1;
        y = y * z;
    }
    return y;
}
```

Il programma è
un'espressione formale

Specifica *S*:

Se x è un intero ≥ 0 il
programma *fact(x)* termina
con $y = x!$ (y è il fattoriale
di x)

Questa specifica non
è formalizzata



La formalizzazione delle proprietà dei programmi

Semantica operazionale

La semantica operazionale definisce il significato di un programma come il suo comportamento che, quando termina, trasforma uno stato in un altro.

Uno stato è una mappa dalle variabili ai valori:

$$\sigma : Var \rightarrow Val$$

$$(P, \sigma) = (P_0, \sigma_0) \rightarrow (P_1, \sigma_1) \rightarrow \dots \rightarrow (P_k, \sigma_k)$$

P_1 è la parte che resta da eseguire di P_0

σ_1 è lo stato risultante dall'esecuzione della prima istruzione di P_0 nello stato σ_0

Se P_k è vuoto (terminato) σ_k è il risultato della computazione:

$$(P, \sigma) \downarrow \sigma_k$$

Da Floyd a Hoare



Nell'articolo: "Assigning Meanings to Programs" (1967) introdusse il **metodo delle asserzioni**, consistente nel decorare diagrammi di flusso mediante formule logiche

R.J. Floyd (1936-2001)

Da Floyd a Hoare

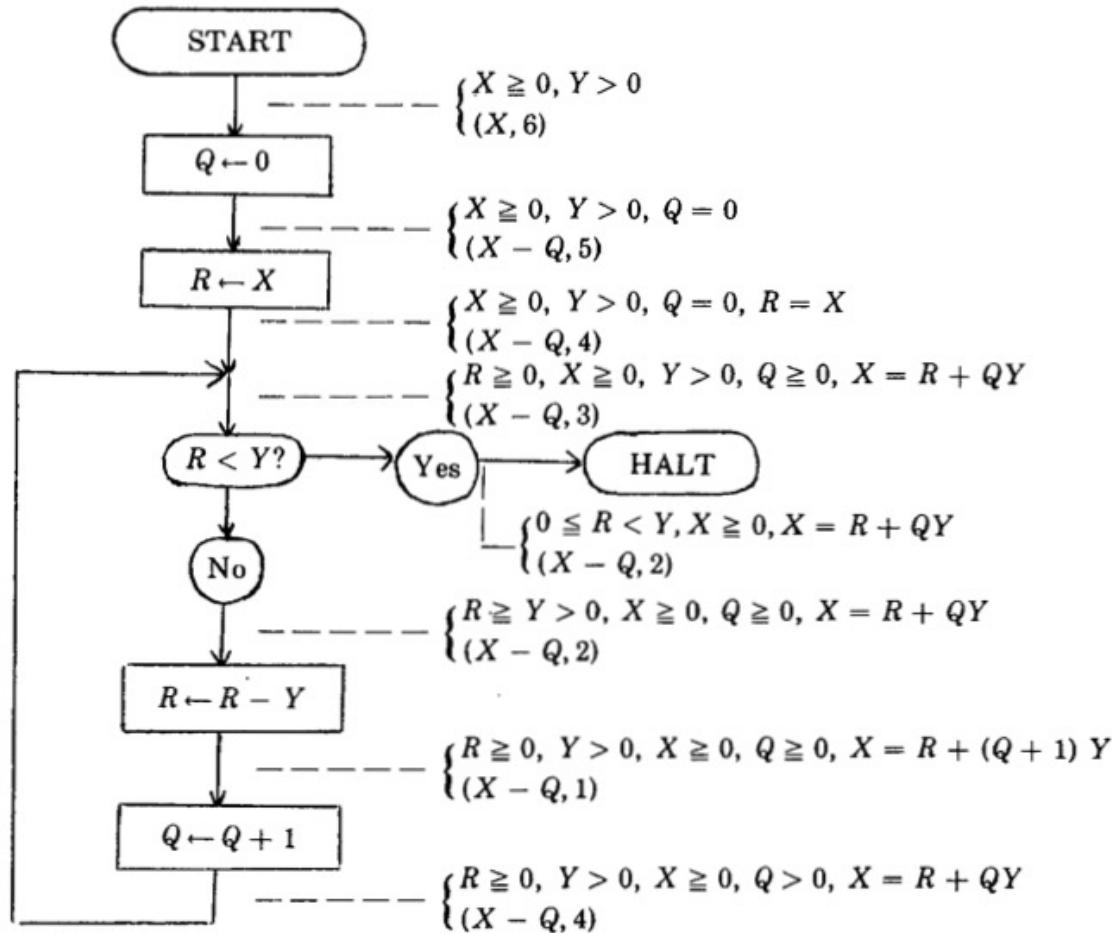


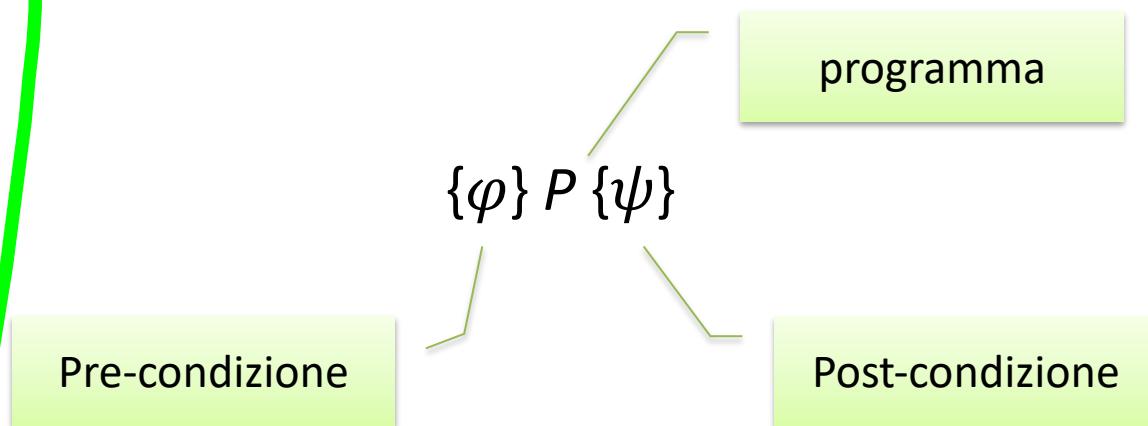
FIGURE 5. Algorithm to compute quotient Q and remainder R of $X \div Y$, for integers $X \geq 0, Y > 0$

Da Floyd a Hoare

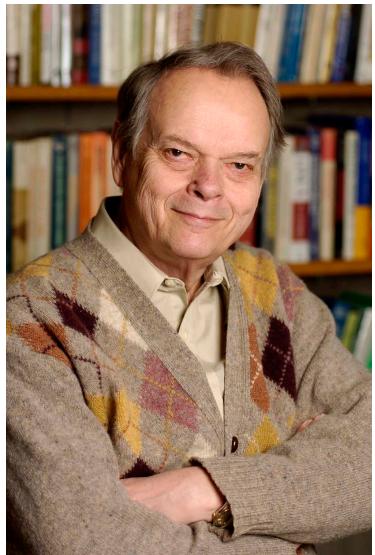
Nell'articolo "An Axiomatic Basis for Computer Programming" (1967) propose un **calcolo logico** che estende FOL mescolando formule logiche a codice di un linguaggio ALGOL-like per dedurre la correttezza dei programmi



C.A.R. Hoare (1934)



Separation Logic - SL



J. C. Reynolds
(1935-2013)

SMALL AXIOMS

Pointer Write (Store)

$$\{x \mapsto -\}[x] = v \{x \mapsto v\}$$

Pointer Read (Load)

$$\{x \mapsto v\}y = [x] \{y == v \wedge x \mapsto v\}$$

Allocation

$$\{emp\}x = \text{alloc}() \{x \mapsto -\}$$

De-Allocation

$$\{x \mapsto -\}\text{free}(x) \{emp\}$$

LOCAL REASONING RULES

Frame Rule

$$\frac{\{pre\}code \{post\}}{\{pre * frame\}code \{post * frame\}}$$

Concurrency Rule

$$\frac{\{pre_1\}process_1 \{post_1\} \quad \{pre_2\}process_2 \{post_2\}}{\{pre_1 * pre_2\}process_1 \parallel process_2 \{post_1 * post_2\}}$$

Analisi statica dei programmi

Verifica vs Testing

“Software testing is a process, or a series of processes, designed to make sure computer code does what it was designed to do and that it does not do anything unintended. [...]

Testing is the process of executing a program with the intent of finding errors”

“Therefore, don’t test a program to show that it works; rather, you should start with the assumption that the program contains errors (a valid assumption for almost any program) and then test the program to find as many of the errors as possible.”

G. J. Myers, The Art of Software Testing

```

int fact(int x)
//@ requires: 0 <= x
//@ ensures: \result == x!
{
    int y = 1;
    int z = 0;

    while (z < x) {

        z = z + 1;

        y = y * z;

    }

    return y;
}

```

}

pre e post condizioni

Questo specifica viene detta **contratto** della funzione fact()



```

int fact(int x)
//@ requires: 0 <= x
//@ ensures: \result == x!
{
    int y = 1;
    int z = 0;
    //@ loop_invariant: 0 <= z & z <= x & y == z!
    while (z < x) {
        //@ y * (z + 1) == (z + 1) !
        z = z + 1;
        //@ y * z == z !
        y = y * z;
        //@ y == z !
    }
    //@ z == x & y == z !
    return y;
}

```

Cosa mi assicura che
le conclusioni di
questo ragionamento
siano corrette?



Semantica della logica di Floyd -Hoare

$\{\varphi\} P \{\psi\}$ è vera nello stato σ se quando φ sia vera in σ e l'esecuzione di P da σ termini in σ' , ψ è **vera** in σ'

$$\sigma \models \varphi \wedge (P, \sigma) \downarrow \sigma' \Rightarrow \sigma' \models \psi$$

φ è vera in σ

ψ è vera in σ'

abbreviato con $\sigma \models \{\varphi\} P \{\psi\}$

Verifica model-based o model checking

- Nella verifica **model-based** o *model checking* si costruisce un **modello** (un automa, un sistema di transizione, ...) M del sistema/protocollo e se ne **specifica** il comportamento con una **formula temporale** (LTL, CTL, ...) φ ; quindi si stabilisce se:

$M \models \varphi$ ovvero “ M soddisfa φ ”

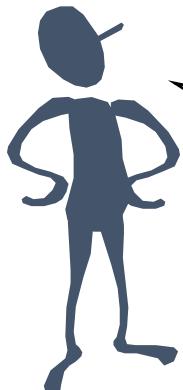


Per stabilire se M soddisfa φ occorre controllare tutti gli stati del sistema M , che quindi devono essere **finiti**

Verifica “proof-based” o deduttiva

- Nella verifica **proof-based** o **deduttiva** non si considerano tutti gli (infiniti) stati che il sistema/programma può attraversare, ma si cerca di dimostrare che la relazione input-output R del programma, descritta dalla formula φ_R , è deducibile in un calcolo logico da un insieme (finito) di ipotesi Γ :

$\Gamma \vdash \varphi_R$ ovvero “ φ_R è deducibile da Γ ”



La deducibilità in un calcolo logico significa l'esistenza di una *derivazione* nel calcolo

La logica di Hoare - HL

Rule for skip:

$$\{A\} \text{skip} \{A\}$$

Rule for assignments:

$$\{B[a/X]\} X := a \{B\}$$

Rule for sequencing:

$$\frac{\{A\}c_0\{C\} \quad \{C\}c_1\{B\}}{\{A\}c_0;c_1\{B\}}$$

Rule for conditionals:

$$\frac{\{A \wedge b\}c_0\{B\} \quad \{A \wedge \neg b\}c_1\{B\}}{\{A\}\text{if } b \text{ then } c_0 \text{ else } c_1\{B\}}$$

Rule for while loops:

$$\frac{\{A \wedge b\}c\{A\}}{\{A\}\text{while } b \text{ do } c\{A \wedge \neg b\}}$$

Rule of consequence:

$$\frac{\models (A \Rightarrow A') \quad \{A'\}c\{B'\} \quad \models (B' \Rightarrow B)}{\{A\}c\{B\}}$$

Correttezza di HL

Teorema. Se la tripla $\{\varphi\}P\{\psi\}$ è derivabile in HL, allora è **valida**:

$$\vdash \{\varphi\}P\{\psi\} \implies \models \{\varphi\}P\{\psi\}$$

dove $\{\varphi\}P\{\psi\}$ è **valida** se

$$\forall \sigma. \sigma \models \{\varphi\}P\{\psi\}$$

Strumenti automatici e semiautomatici

Limiti teorici

- FOL è corretta e completa, ma indecidibile
- HL è corretta, ma completa solo in senso debole; include FOL dunque è indecidibile
- In conseguenza del **Teorema di Rice** tutte le proprietà funzionali (inclusa la correttezza) dei programmi sono indecidibili

Limiti teorici

- La decisione di soddisfabilità nel calcolo proposizionale, decidibile, è NP-completa
- Tutti i frammenti decidibili di FOL sono EXPTIME

$$\text{SAT} = \left\{ \varphi \mid \exists \text{v}_\varphi \in \text{vars}^\text{app} \text{ d.o.} \text{ s.t. } \text{v}_\varphi \models \varphi \right\}$$

vars → variabili
d.o. → bool

Perciò nella verifica deduttiva
non si parla di dimostratori
automatici ma di proof assistant
interattivi



Problemi pratici

```
fib.key          x fact.key          x multiplication.key

1 \functions{
2   int fact(int);
3 }
4 \programVariables{
5   int x;
6   int y;
7   int z;
8 }
9 /* invariante:
10  fact(0) = 1 &
11  \forall int n; (0 < n -> fact(n) = fact(n - 1) * n) &
12  0 <= z & z <= x & y = fact(z)
13 */
14 \hoare{
15 {
16   fact(0) = 1 &
17   \forall int n; (0 < n -> fact(n) = fact(n - 1) * n) &
18   0 <= x
19 }
20 \{
21   y = 1;
22   z = 0;
23   while (z < x) {
24     z = z + 1;
25     y = y * z;
26   }
27 }\]
28 {y = fact(x)}
29 }
30 |
```



Tool e progetti: Infer



The image shows the homepage of the Infer static analysis tool. At the top, there's a navigation bar with links for Docs, Support, Blog, Twitter, Facebook, GitHub, and a search bar. Below the header is a large purple banner with white text that reads: "A tool to detect bugs in Java and C/C++/Objective-C code before it ships". Underneath the banner, a paragraph explains what Infer does: "Infer is a static analysis tool - if you give Infer some Java or C/C++/Objective-C code it produces a list of potential bugs. Anyone can use Infer to intercept critical bugs before they have shipped to users, and help prevent crashes or poor performance." There are two buttons below the text: "Get Started" and "Learn More". At the bottom of the banner, there's a star icon followed by the number "11,226".

Android and Java

Infer checks for null pointer exceptions, resource leaks, annotation reachability, missing lock guards, and concurrency race conditions in Android and Java code.

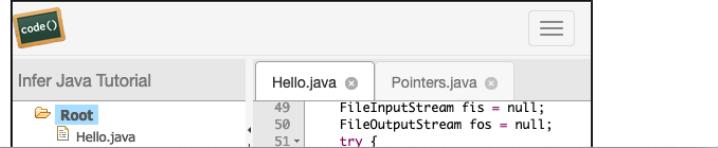
C, C++, and iOS/Objective-C

Infer checks for null pointer dereferences, memory leaks, coding conventions and unavailable API's.

Infer in Action

```
Analyzing 1 cluster. 100%
Analyzed 2 procedures in 1 file
No issues found
infer@facebook:~/infer/examples$ vim Infer.java
infer@facebook:~/infer/examples$ infer -- javac Infer.java
Starting analysis (Infer version v0.5.0)
Computing dependencies... 100%
Analyzing 1 cluster. 100%
Analyzed 2 procedures in 1 file
```

Try Infer



A screenshot of the Infer Java Tutorial interface. It shows a code editor with the following Java code:

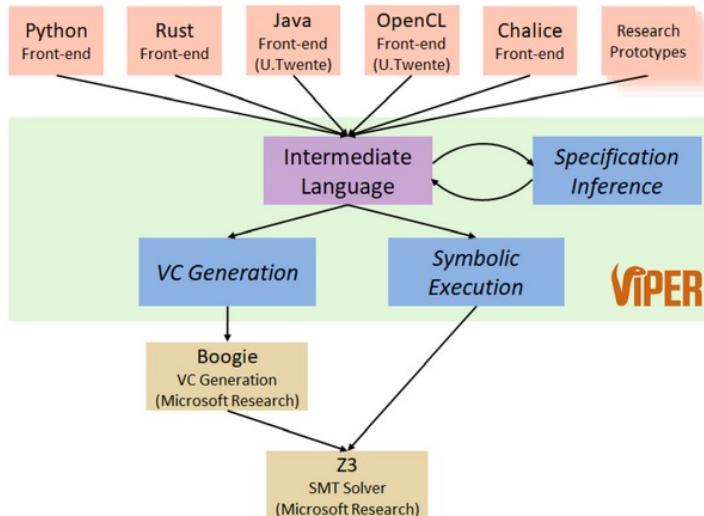
```
code()
Infer Java Tutorial
Hello.java Pointers.java
Root Hello.java
49 FileInputStream fis = null;
50 FileOutputStream fos = null;
51 try {
```

Tool e progetti: Viper

Viper

NEW: for an introduction to Viper's features, try the [Viper Tutorial here!](#) →

Viper (Verification Infrastructure for Permission-based Reasoning) is a language and suite of tools developed at ETH Zurich, providing an architecture on which new verification tools and prototypes can be developed simply and quickly. It comprises a novel intermediate verification language, also named *Viper*, and automatic verifiers for the language, as well as example front-end tools. The Viper toolset can be used to implement verification techniques for front-end programming languages via translations into the Viper language.



The Viper toolchain is designed to make it easy to implement verification techniques for sequential and concurrent programs with mutable state. It provides native support for reasoning about the program state using *permissions* or *ownership*, e.g. in the style of *separation logic*. New verification techniques can be implemented dir-

Project Members

Linard Arquint →
Vytautas Astrauskas →
Aurel Bílý →
Martin Clochard →
Thibault Dardinier →
Jérôme Dohrau →
Marco Eilers →
Christoph Matheja →
João Carlos Mendes Pereira →
Peter Müller →
Wytse Oortwijn →
Fábio Pakk Selmi-Dei →
Gaurav Parthasarathy →
Federico Poli →
Malte Schwerhoff →
Alex Summers (UBC) →
Arshavir Ter-Gabrielyan →
Felix Wolf →

Links

[Try the Viper Tutorial](#) →
[Download Viper](#) →
See additional [Viper examples online](#) →

Tool e progetti: Key

The screenshot shows the KeY Project website homepage on the left and the KeY tool interface on the right.

Website Header:

- The KeY Project
- News
- Applications ▾
- The KeY Book
- Publications
- About ▾
- Downloads

Homepage Content:

Functional Verification

We can do better than testing!

KeY lets you specify the desired behavior of your program in the well-known specification language JML, and helps you proving that your programs conforms to its specification. That way, you did not only show that your program behaves as expected for some set of test values - you proved that it works correctly for *all possible values*!

Go beyond testing - start proving!

KeY Tool Interface:

- Proof Tree:** Shows a tree of proof steps, starting with 0:int.
- Inner Node:** Displays the JML code for a method named `wellFormed`. The code handles integer division and modulus operations, including error handling for null inputs and division by zero.
- Tool Status:** Shows "KeY 2.5" at the top.
- Help Hint:** A tooltip indicates that pressing ALT over a term shows its unfoldings.

Tool e progetti: VeriFast

reverse.c stdlib.h malloc.h prelude.h prelude_core.gh list.gh

```
predicate nodes(struct node *node, ints values) =
    node == 0
    values == ints nil
    :
    node->next |-> ?next && node->value |-> ?value && malloc block node(node) &&
    nodes(next, ?values0) && values == ints cons(value, values0);

predicate stack(struct stack *stack, ints values) =
    stack->head |-> ?head && malloc block stack(stack) && nodes(head, values);
@*/
struct stack *create_stack()
    /* requires true;
     * ensures stack(result, ints nil);
    {
        struct stack *stack = malloc(sizeof(struct stack));
        if (stack == 0) { abort(); }
        stack->head = 0;
        /* close nodes(0, ints nil);
        /* close stack(stack, ints nil);
        return stack;
    }

void stack push(struct stack *stack, int value)
    /* requires stack(stack, ?values);
     * ensures stack(stack, ints cons(value, values));
    {
        /* close stack(stack, ints cons(value, values));
    }
```

reverse.c stdlib.h malloc.h prelude.h prelude_core.gh list.gh

```
predicate nodes(struct node *node, ints values) =
    node == 0
    values == ints nil
    :
    node->next |-> ?next && node->value |-> ?value && malloc block node(node) &&
    nodes(next, ?values0) && values == ints cons(value, values0);

predicate stack(struct stack *stack, ints values) =
    stack->head |-> ?head && malloc block stack(stack) && nodes(head, values);
@*/
struct stack *create_stack()
    /* requires true;
     * ensures stack(result, ints nil);
    {
        struct stack *stack = malloc(sizeof(struct stack));
        if (stack == 0) { abort(); }
        stack->head = 0;
        /* close nodes(0, ints nil);
        /* close stack(stack, ints nil);
        return stack;
    }

void stack push(struct stack *stack, int value)
    /* requires stack(stack, ?values);
     * ensures stack(stack, ints cons(value, values));
    {
        /* close stack(stack, ints cons(value, values));
    }
```

Steps

- Producing assertion
- Executing statement
- Executing second branch
- Executing statement
- Executing second branch
- Executing statement
- Executing statement
- Executing statement
- Consuming assertion

Assumptions

- true
- !(stack0 = 0)
- !(stack0 = 0)

Heap chunks

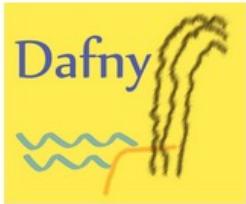
- malloc_block_stack(stack0)
- stack_head(stack0, 0)

Target branch reached

Verifying function 'create_stack'

Tool e progetti: Dafny

The Dafny Programming and Verification Language



Dafny is a verification-aware programming language that has native support for recording specifications and is equipped with a static program verifier. By blending sophisticated automated reasoning with familiar programming idioms and tools, Dafny empowers developers to write provably correct code (w.r.t. specifications). It also compiles Dafny code to familiar development environments such as C#, Java, JavaScript, Go and Python (with more to come) so Dafny can integrate with your existing workflow. Dafny makes rigorous verification an integral part of development, thus reducing costly late-stage bugs that may be missed by testing.

In addition to a verification engine to check implementation against specifications, the Dafny ecosystem includes several compilers, plugins for common software development IDEs, a LSP-based Language Server, a code formatter, a reference manual, tutorials, power user tips, books, the experiences of professors teaching Dafny, and the accumulating expertise of industrial projects using Dafny.

Dafny has support for common programming concepts such as

- mathematical and bounded integers and reals, bit-vectors, classes, iterators, arrays, tuples, generic types, refinement and inheritance,
- [inductive datatypes](#) that can have methods and are suitable for pattern matching,
- [lazily unbounded datatypes](#),
- [subset types](#), such as for bounded integers,
- [lambda expressions](#) and functional programming idioms,
- and [immutable and mutable data structures](#).

Quick Links

- [Installation](#) (or a [VSCode plugin for Dafny](#))
- [Dafny Reference Manual and User Guide](#)
- [Dafny Resources for Users](#)
- [Dafny GitHub project \(for developers of the Dafny tools themselves\)](#)
- [Other documentation snapshots](#)
- [Book on Program Proofs using Dafny!: Program Proofs](#), by Rustan Leino, MIT Press

Blog

Tool e progetti: Frama-C



Software Analyzers

ABOUT FEATURES DOCUMENTATION PUBLICATIONS BLOG JOBS

Overview of a Frama-C analysis for a simple C program

Browsing the analysis results with Frama-C

Interface
Overview

Value
Analysis

Effects
Analysis

Dependency
Analysis

Impact
Analysis

When invoked with the command-line:

```
frama-c -eva -eva-precision 1 first.c
```

Frama-C creates an analysis project for the file first.c.

The `-eva` option on the command-line causes the Eva plug-in to run and have its results ready before the interface appears.

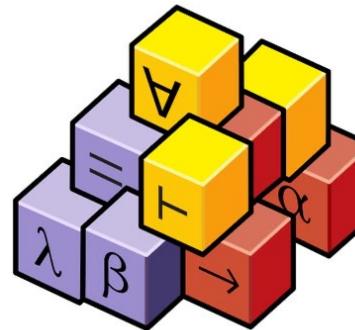
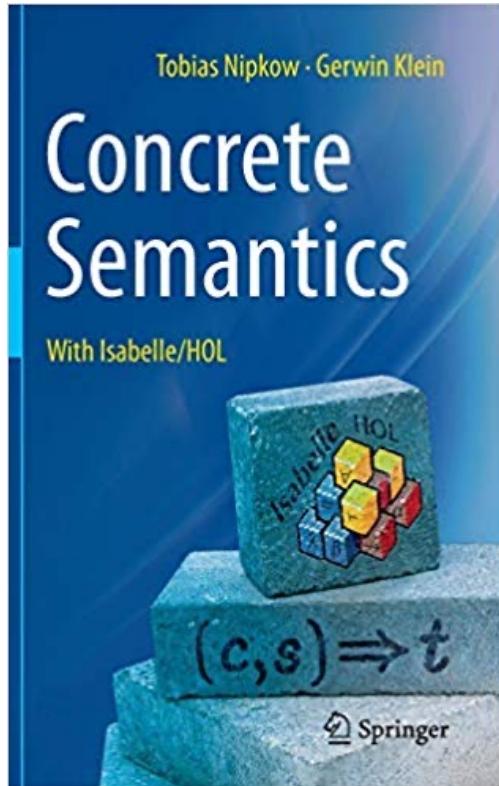
The `-eva-precision` option is one of several options that influence the precision of the Eva plug-in. The actions of creating new analysis projects and activating plug-ins can also be done interactively.

The screenshot shows the Frama-C interface with the following details:

- Source File:** The left pane displays the C code for `first.c`.

```
int main(void)
{
    int i;
    int *p;
    p = T;
    i = 0;
    while (i < 5) {
        int *tmp;
        S += i;
        /* sequence */
        tmp = p;
        *tmp = S;
    }
    i++;
}
return S;
```
- Analysis Results:** The right pane shows the analysis results for the `main` function. It includes:
 - A summary table with columns: WP, Slicing, Occurrence, Metrics, Impact, and Value.
 - A detailed view of local variable `i` in the `main` function.
 - An information panel at the bottom stating: "This is the declaration of local variable i in function main".

Isabelle



Tobias Nipkow, Gerwin Klein
Concrete Semantics With Isabelle HOL
Springer

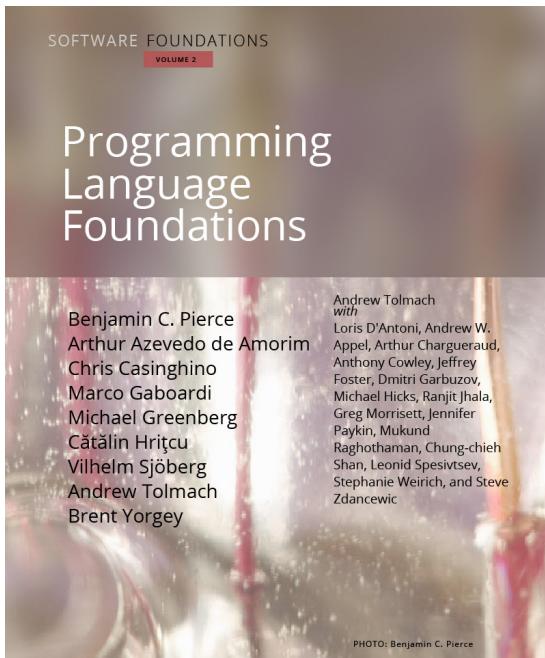
Disponibile liberamente come pdf:
<http://www.concrete-semantics.org/>

Isabelle

The screenshot shows the Isabelle IDE interface with the following details:

- Title Bar:** Chapter3.thy
- Toolbar:** Standard icons for file operations, search, and navigation.
- Text Editor:** Displays the theory file Chapter3.thy. The code includes:
 - A definition of `is optimal`.
 - Three `value` declarations showing the result of applying the `optimal` function to different arguments.
 - A `lemma` block with an induction step using `asimp_const.induct` and `aexp.split`.
 - A `text` block containing a note about proof termination.
- Proof State:** Shows the current proof state with three subgoals:
 - $\lambda n. \text{optimal} (\text{asimp_const} (N\ n))$
 - $\lambda x. \text{optimal} (\text{asimp_const} (V\ x))$
 - $\lambda a_1\ a_2. \text{optimal} (\text{asimp_const}\ a_1); \text{optimal} (\text{asimp_const}\ a_2) \Rightarrow \text{optimal} (\text{asimp_const} (\text{Plus}\ a_1\ a_2))$
- Bottom Navigation:** Buttons for Output, Query, Sledgehammer, Symbols, and status information (34,47 (909/14464) (isabelle,isab)).

Coq



Benjamin Pierce ed altri Programming Language Foundations

Disponibile liberamente alla url:
<https://softwarefoundations.cis.upenn.edu/plf-current/>

Coq

The screenshot shows the Coq IDE interface with the following details:

- File Menu:** File, Edit, View, Navigation, Try Tactics, Templates, Queries, Tools, Compile, Windows, Help.
- Toolbar:** Includes icons for file operations (New, Open, Save, Print, Copy, Paste, Find, Replace, Undo, Redo), a search bar, and a help icon.
- Left Panel:** Displays the source code of the file `Arith.v`. The code includes proofs for various comparison lemmas and properties of the maximum function. A red box highlights the proof for `compare_le_iff`.
- Right Panel:** Shows the proof state with two subgoals:
 - Subgoal 1: $\forall m : \text{nat}, (n \geq m) \Leftrightarrow Gt \leftrightarrow n \leq m$
 - Subgoal 2: $n \leq m$
- Bottom Panel:** Contains tabs for Messages, Errors, and Jobs. The status bar at the bottom indicates: "Ready in Nat, proving compare_le_iff", "Line: 211 Char: 18", "Coq is ready", and "0 / 0".

Dimostrazioni come programmi

A proof written as a functional program

```
plus_comm =
  fun n m : nat =>
  nat_ind (fun n0 : nat => n0 + m = m + n0)
    (plus_n_0 m)
    (fun (y : nat) (H : y + m = m + y) =>
      eq_ind (S (m + y))
        (fun n0 : nat => S (y + m) = n0)
        (f_equal S H)
        (m + S y)
        (plus_n_Sm m y)) n
    : forall n m : nat, n + m = m + n
```

The proof of commutativity of addition on natural numbers in the proof assistant Coq. `nat_ind` stands for [mathematical induction](#), `eq_ind` for substitution of equals, and `f_equal` for taking the same function on both sides of the equality. Earlier theorems are referenced showing $m = m + 0$ and $S(m + y) = m + Sy$.

Perché Agda?

The image shows a YouTube video player interface. On the left, there is a video frame showing a man with glasses and a beard, wearing a patterned shirt and a brown tie, gesturing with his hands. To the right of the video frame is a white slide with text and logos. The slide title is *Programming Languages* in *Foundations in Agda*. Below the title, it says "Philip Wadler" and "Philip Wadler (with Wen Kokke and Jeremy Siek) University of Edinburgh / IOHK / Rio de Janeiro wadler.com". It also mentions "Full Stack Fest, Sitges, 5 September 2019". At the bottom of the slide is a New Relic logo. The YouTube player interface includes a progress bar showing "0:02 / 44:34", a play button, and other control icons. The URL of the video is displayed at the bottom: [\(Programming Languages\) in Agda = Programming \(Languages in Agda\) by Philip Wadler](https://www.youtube.com/watch?v=R49VgxNLmsY).

<https://www.youtube.com/watch?v=R49VgxNLmsY>

Agda

Programming Language Foundations in Agda

[The Book](#) [Announcements](#) [Getting Started](#) [Citing](#) [中文](#)

Table of Contents

This book is an introduction to programming language theory using the proof assistant Agda.

Comments on all matters—organisation, material to add, material to remove, parts that require better explanation, good exercises, errors, and typos—are welcome. The book repository is on [GitHub](#). Pull requests are encouraged.

Front matter

- [Dedication](#)
- [Preface](#)
- [Getting Started](#)

Part 1: Logical Foundations

- [Naturals](#): Natural numbers
- [Induction](#): Proof by induction
- [Relations](#): Inductive definition of relations
- [Equality](#): Equality and equational reasoning
- [Isomorphism](#): Isomorphism and embedding
- [Connectives](#): Conjunction, disjunction, and implication



Philip Wadler
Programming Language Foundations
in Agda

Disponibile liberamente alla url:
<https://plfa.github.io/>

Agda

```
AA_helloPeano.agda
module AA_helloPeano where
  -- il nome del modulo e quello del file devono coincidere
  -- questo è un commento su una riga
  {- questo è un
   commento su più
   righe
  -}

  -- definizione del tipo dei naturali, N : Set vuol dire che N è un tipo
  data N : Set where  -- N si scrive '\bN'
    zero : N          -- indentazione necessaria
    suc  : N → N      -- → si scrive '\->'

  -- definizione di + come operatore infisso; il tipo è currificato
  _+_ : N → N → N
  zero + n = n
  (suc m) + n = suc (m + n)

  -- il commento che segue è un Pragma
  -- che consente di usare interi in notazione decimale
  {-# BUILTIN NATURAL N #-}

  -- importa ed apre la teoria dell'egualianza
  import Relation.Binary.PropositionalEquality as Eq
  open Eq using (_≡_; refl)
  open Eq.≡-Reasoning using (begin_; _≡(); _≡)

  {- _ è un termine anonimo per la 'dimostrazione'
   del tipo, ossia della formula, 2 + 3 ≡ 5
  -}
  _ : 2 + 3 ≡ 5

  -- ≡ si scrive '==' e rappresenta l'identità delle forme normali
  □:--- AA_helloPeano.agda  Top L42  (Agda:Checked)
  □

  [-:%- *All Done* All L1  (AgdaInfo)
<nil> <mouse-1> is undefined
```

MFI – Programma 2022/23

- Metodi formali
 - Logica equazionale e riscrittura
 - Deduzione naturale
 - Lambda calcolo
- Il sistema Agda
 - Funzioni e tipi di dato
 - Logica costruttiva
 - Liste e strutture dati *
 - Verifica di programmi funzionali *
- Fondamenti di linguaggi di programmazione procedurali
 - Sintassi del linguaggio IMP
 - Semantica operazionale big-step e small-step
 - Sistemi di tipo per IMP *
 - Logica di Hoare
 - Completezza relativa
 - Verification conditions
 - Separation logic *