Metodi formali dell'Informatica

Introduzione alla semantica ed alla

logica dei programmi

a.a. 2023-24

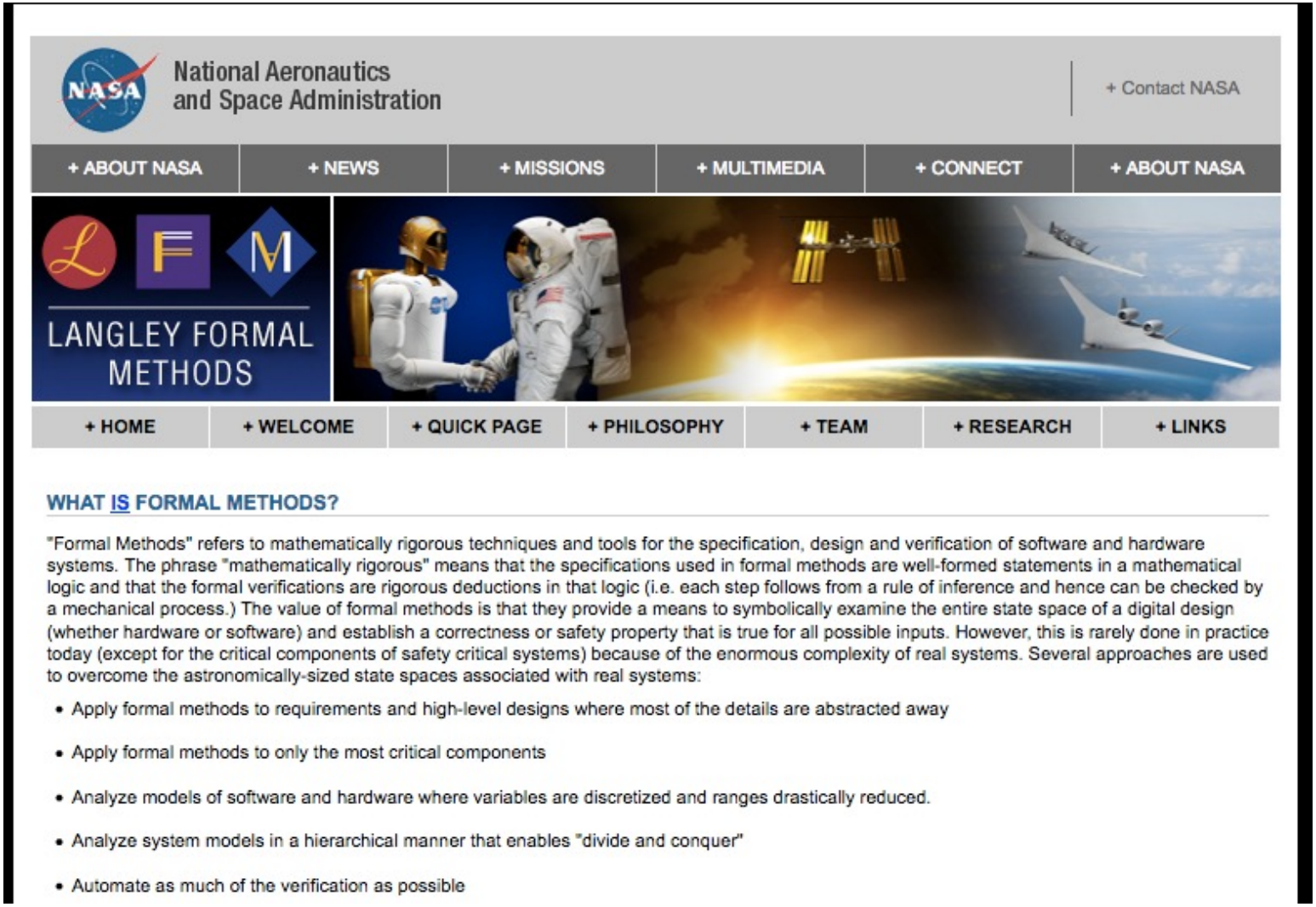
Ugo de'Liguoro

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”?



National Aeronautics and Space Administration

+ Contact NASA

+ ABOUT NASA | + NEWS | + MISSIONS | + MULTIMEDIA | + CONNECT | + ABOUT NASA

LANGLEY FORMAL METHODS

+ HOME | + WELCOME | + QUICK PAGE | + PHILOSOPHY | + TEAM | + RESEARCH | + 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

Metodi formali, fondamenti

Riscrittura

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

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

CNF

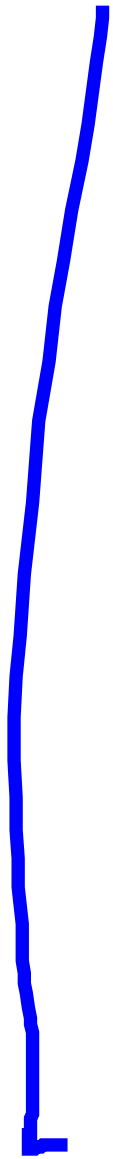
Riscrittura

$$0 + m \longrightarrow m$$
$$S(n) + m \longrightarrow S(n + m)$$

S = Successore

$$0 \cdot m \longrightarrow 0$$
$$S(n) \cdot m \longrightarrow m + (n \cdot m)$$

Riscrittura



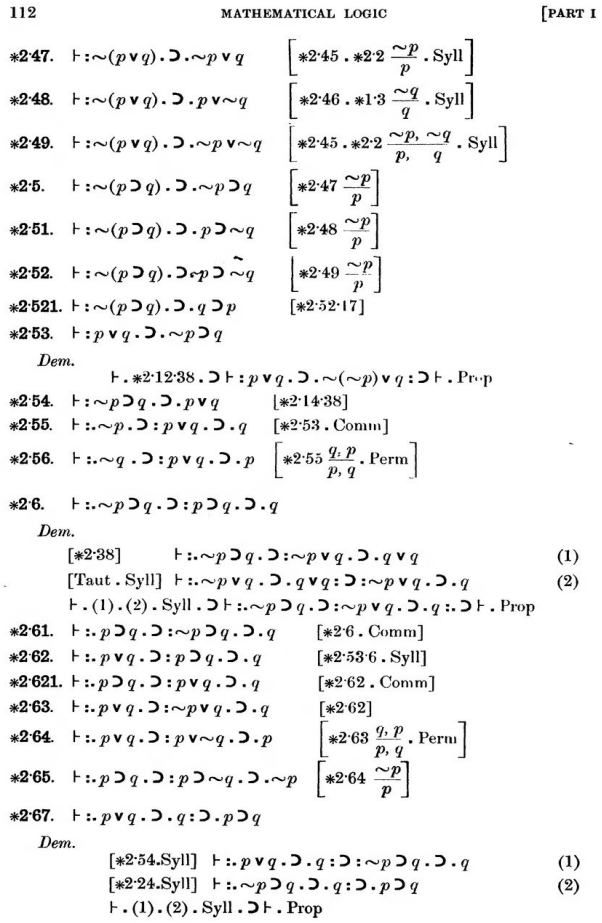
$$
\begin{aligned}
3 \cdot 2 &\longrightarrow 2 + (2 \cdot 2) \\
&\longrightarrow 2 + (2 + (2 \cdot 1)) \\
&\longrightarrow 2 + (2 + (2 + (2 \cdot 0))) \\
&\longrightarrow 2 + (2 + (2 + 0)) \\
&\xrightarrow{2} S^2(2 + (2 + 0)) \\
&\xrightarrow{2} S^4(2 + 0) \\
&\xrightarrow{2} S^6(0) \equiv 6
\end{aligned}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

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

*2·47. $\vdash :\sim(p \vee q).\supset.\sim p \vee q$ $\left[*2{\cdot}45 . *2{\cdot}2 \frac{\sim p}{p} . \text{Syll}\right]$

*2·48. $\vdash :\sim(p \vee q).\supset.p \vee \sim q$ $\left[*2{\cdot}46 . *1{\cdot}3 \frac{\sim q}{q} . \text{Syll}\right]$

*2·49. $\vdash :\sim(p \vee q).\supset.\sim p \vee \sim q$ $\left[*2{\cdot}45 . *2{\cdot}2 \frac{\sim p, \sim q}{p, q} . \text{Syll}\right]$

*2·5. $\vdash :\sim(p \supset q).\supset.\sim p \supset q$ $\left[*2{\cdot}47 \frac{\sim p}{p}\right]$

*2·51. $\vdash :\sim(p \supset q).\supset.p \supset \sim q$ $\left[*2{\cdot}48 \frac{\sim p}{p}\right]$

*2·52. $\vdash :\sim(p \supset q).\supset.p \supset \sim\sim q$ $\left[*2{\cdot}49 \frac{\sim p}{p}\right]$

*2·521. $\vdash :\sim(p \supset q).\supset.q \supset p$ [*2·52·17]

*2·53. $\vdash :p \vee q.\supset.\sim p \supset q$

Dem.

$\vdash .*2{\cdot}12{\cdot}38 .\supset \vdash :p \vee q.\supset.\sim(\sim p) \vee q :\supset \vdash .$ Prop

*2·54. $\vdash :\sim p \supset q.\supset.p \vee q$ [*2·14·38]

*2·55. $\vdash :.\sim p.\supset :p \vee q.\supset.q$ [*2·53 . Comm]

*2·56. $\vdash :.\sim q .\supset :p \vee q.\supset.p$ $\left[*2{\cdot}55 \frac{q, p}{p, q} . \text{Perm}\right]$

*2·6. $\vdash :.\sim p \supset q.\supset :p \supset q.\supset.q$

Dem.

[*2·38] $\vdash :.\sim p \supset q.\supset :\sim p \vee q.\supset.q \vee q$ (1)

[Taut . Syll] $\vdash :.\sim p \vee q .\supset.q \vee q :\supset :\sim p \vee q.\supset.q$ (2)

$\vdash .(1).(2).\text{Syll}.\supset \vdash :.\sim p \supset q.\supset :\sim p \vee q.\supset.q :.\supset \vdash .$ Prop

*2·61. $\vdash :.p \supset q.\supset :\sim p \supset q.\supset.q$ [*2·6 . Comm]

*2·62. $\vdash :.p \vee q.\supset :p \supset q.\supset.q$ [*2·53·6 . Syll]

*2·621. $\vdash :.p \supset q.\supset :p \vee q.\supset.q$ [*2·62 . Comm]

*2·63. $\vdash :.p \vee q.\supset :\sim p \vee q.\supset.q$ [*2·62]

*2·64. $\vdash :.p \vee q.\supset :p \vee \sim q.\supset.p$ $\left[*2{\cdot}63 \frac{q, p}{p, q} . \text{Perm}\right]$

*2·65. $\vdash :.p \supset q.\supset :p \supset \sim q.\supset.\sim p$ $\left[*2{\cdot}64 \frac{\sim p}{p}\right]$

*2·67. $\vdash :.p \vee q.\supset.q :\supset.p \supset q$

Dem.

[*2·54.Syll] $\vdash :.p \vee q.\supset.q :\supset :\sim p \supset q.\supset.q$ (1)

[*2·24.Syll] $\vdash :.\sim p \supset q.\supset.q :\supset.p \supset q$ (2)

$\vdash .(1).(2).\text{Syll}.\supset \vdash .$ Prop

Logica: la deduzione naturale

$$\dfrac{\dfrac{\dfrac{[\varphi \wedge \psi]^1}{\psi}\wedge E \qquad \dfrac{\dfrac{[\varphi \wedge \psi]^1}{\varphi}\wedge E \qquad [\varphi \to (\psi \to \sigma)]^2}{\psi \to \sigma}\to E}{\sigma}\to E}{\varphi \wedge \psi \to \sigma}\to I_1}{(\varphi \to (\psi \to \sigma)) \to (\varphi \wedge \psi \to \sigma)}\to I_2$$

Gerhard Gentzen

λ-calcolo

Alonzo Church

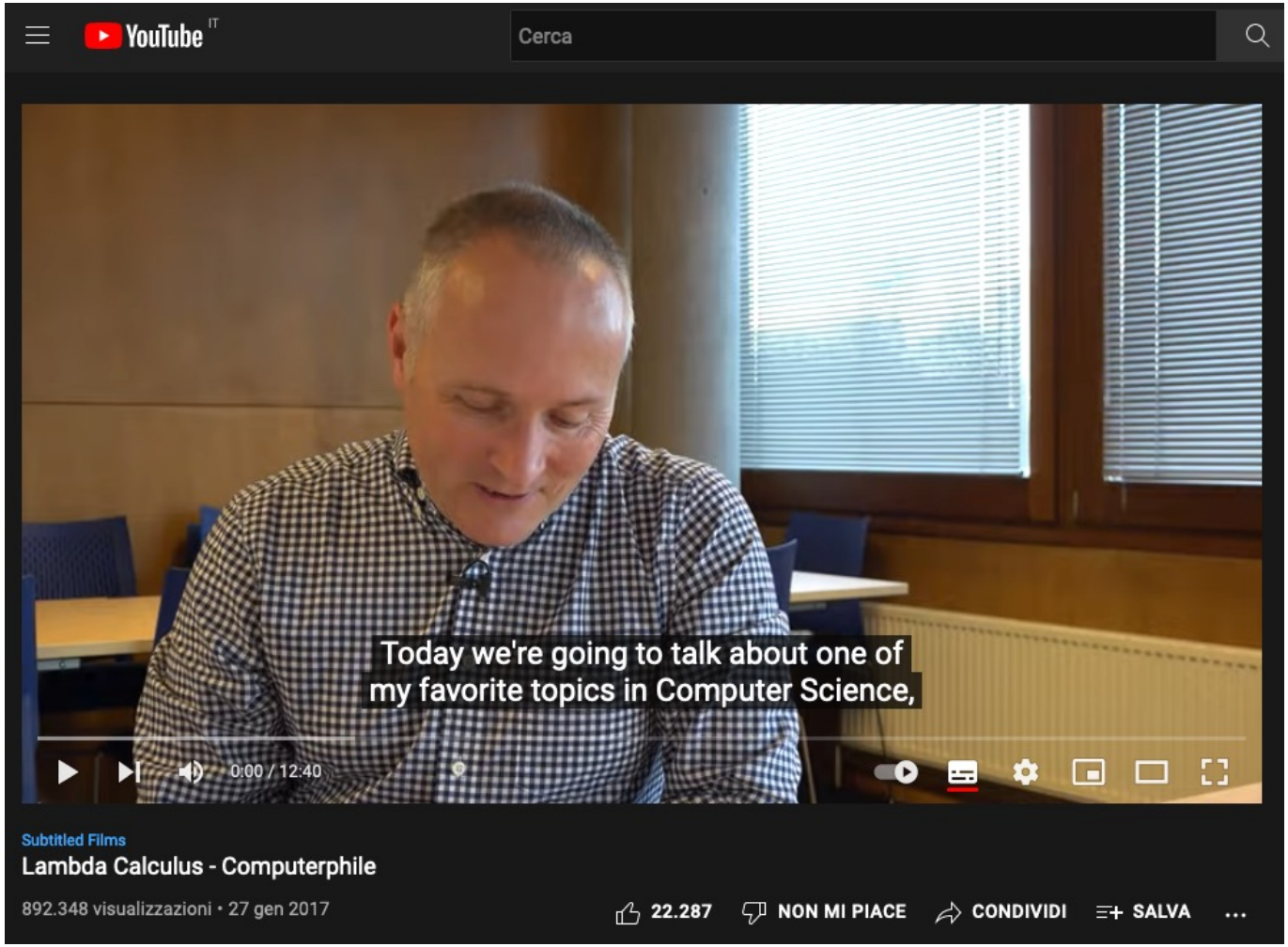
$$M, N ::= x \mid \lambda x.M \mid MN$$

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

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

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

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



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

λ-calcolo tipato

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

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

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma. M : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

$$\frac{\dfrac{\Gamma \vdash x : \sigma \to \tau \to \rho \quad \Gamma \vdash z : \sigma}{\Gamma \vdash xz : \tau \to \rho} \qquad \dfrac{\Gamma \vdash y : \sigma \to \tau \quad \Gamma \vdash z : \sigma}{\Gamma \vdash yz : \tau}}{\Gamma \vdash xz(yz) : \rho}$$

$$\vdash \lambda x : \sigma \to \tau \to \rho \; \lambda y : \sigma \to \tau \; \lambda z : \sigma. \; xz(yz) : (\sigma \to \tau \to \rho) \to (\sigma \to \tau) \to \rho$$

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

Haskell

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

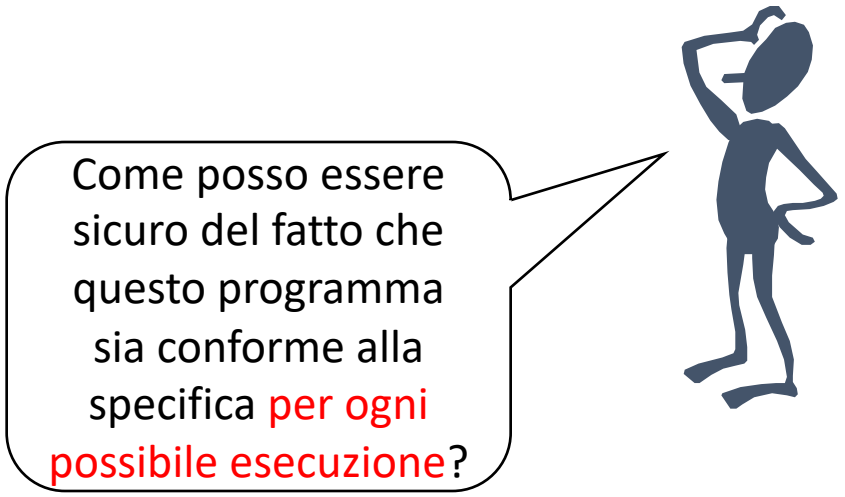
Verifica di un programma

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)



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: $\sigma : Var \rightarrow Val$

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

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

R.J. Floyd (1936-2001)

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

Da Floyd a Hoare

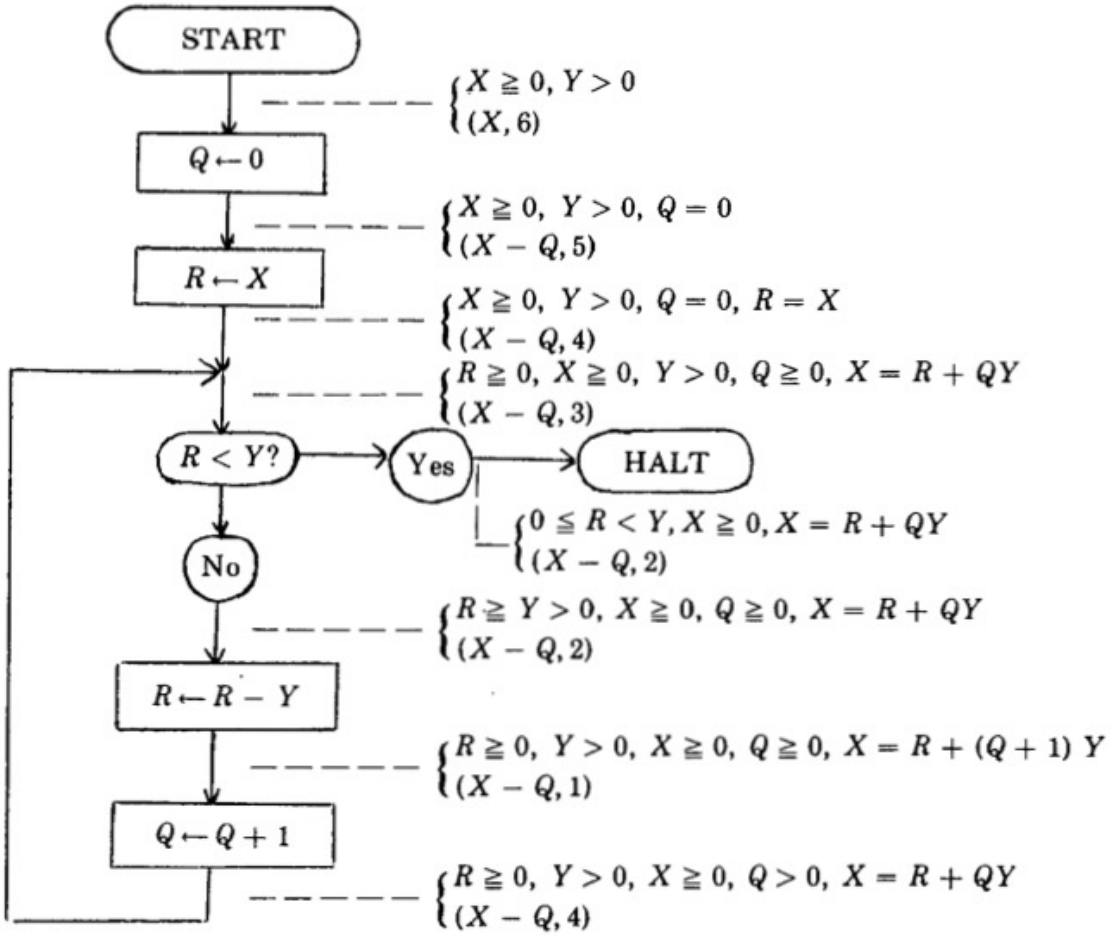


FIGURE 5. Algorithm to compute quotient Q and remainder R of $X \div Y$, for integers $X \geq 0$, $Y > 0$

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)

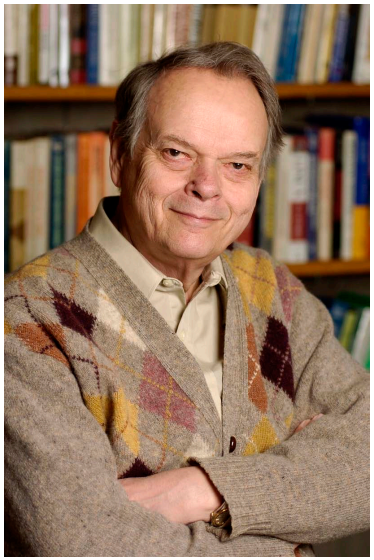
programma

$\{\varphi\}\ P\ \{\psi\}$

Pre-condizione

Post-condizione

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

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

De-Allocation

$$\{x \mapsto -\}\texttt{free}(x) \, \{emp\}$$

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

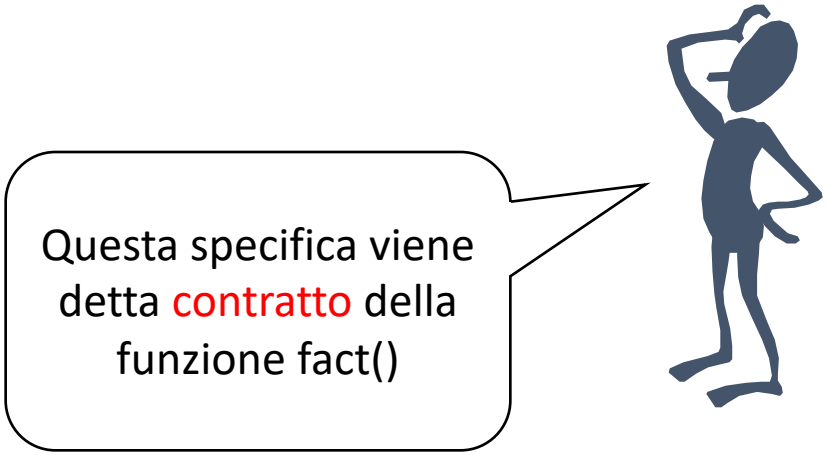
“Software testing is a process, or a series of processes, designed to make sure computer code does what it was designed to do and that it does not do anything unintended. […]
Testing is the process of executing a program with the intent of finding errors”

“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.”

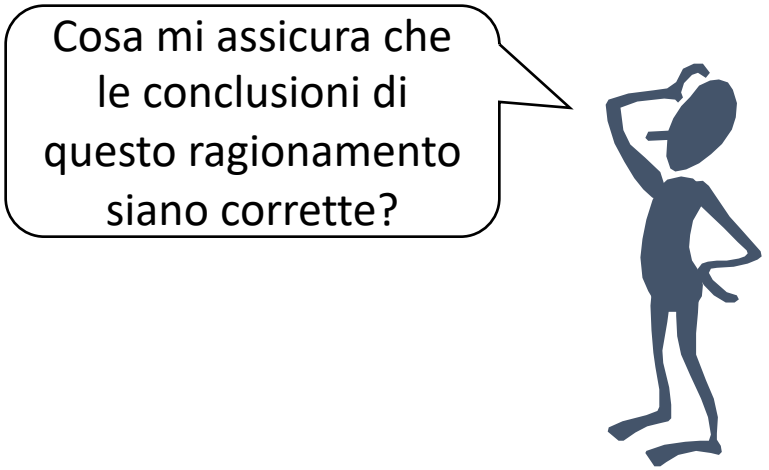
G. J. Myers, The Art of Software Testing

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

    while (z < x) {

        z = z + 1;

        y = y * z;

    }

    return y;
}
```

pre e post condizioni



```
int fact(int x)
//@ requires: 0 <= x
//@ ensaures: \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;
}
```



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

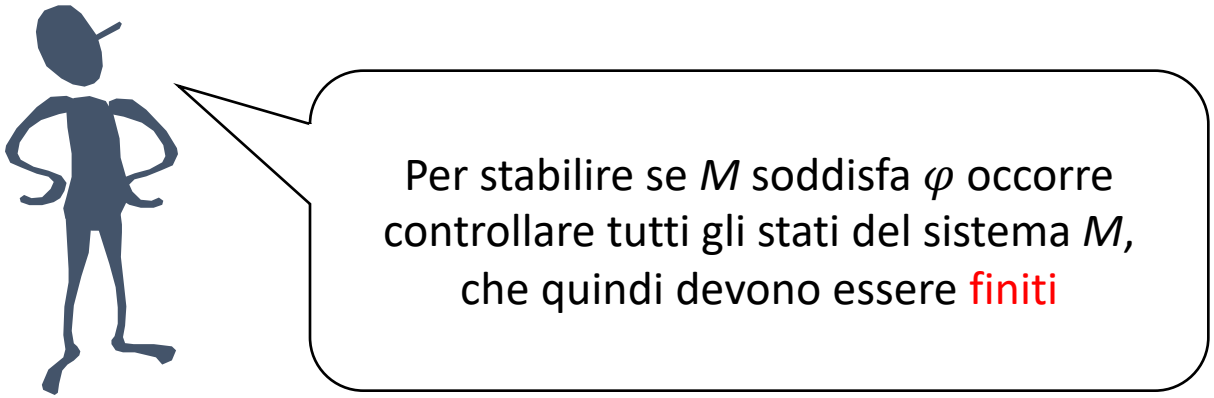
ψ è vera in σ'

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:

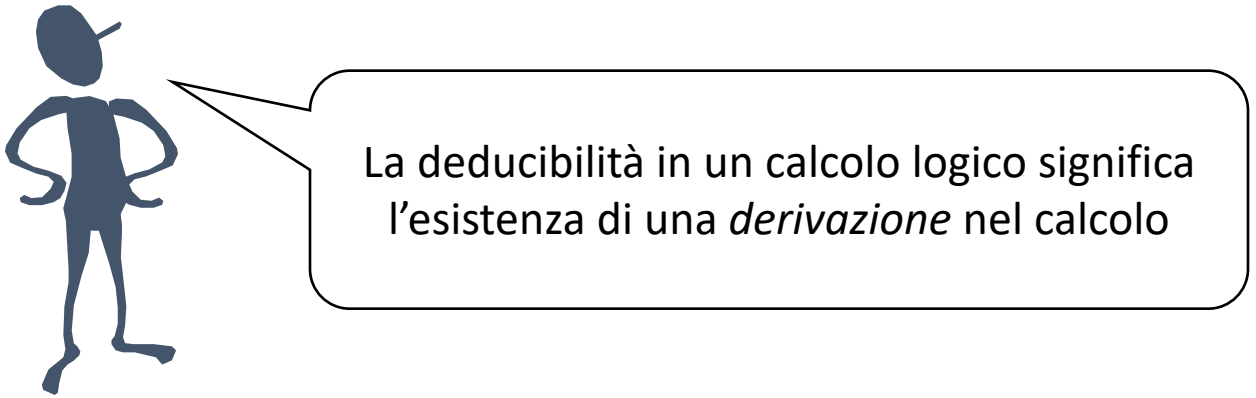
$M \models \varphi$ ovvero "M soddisfa φ"



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 logica di Hoare - HL

Rule for **skip**:

$$\{A\}\mathbf{skip}\{A\}$$

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:

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

Rule for while loops:

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

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

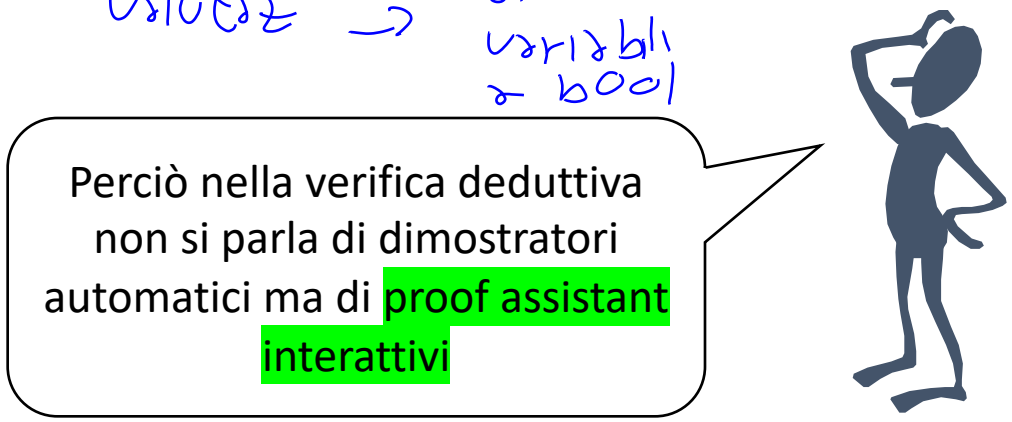
Limiti teorici

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

$SAT = \{ \varphi \mid \exists v . v \models \varphi \}$

CNF

valuta → mappa di variabili a bool

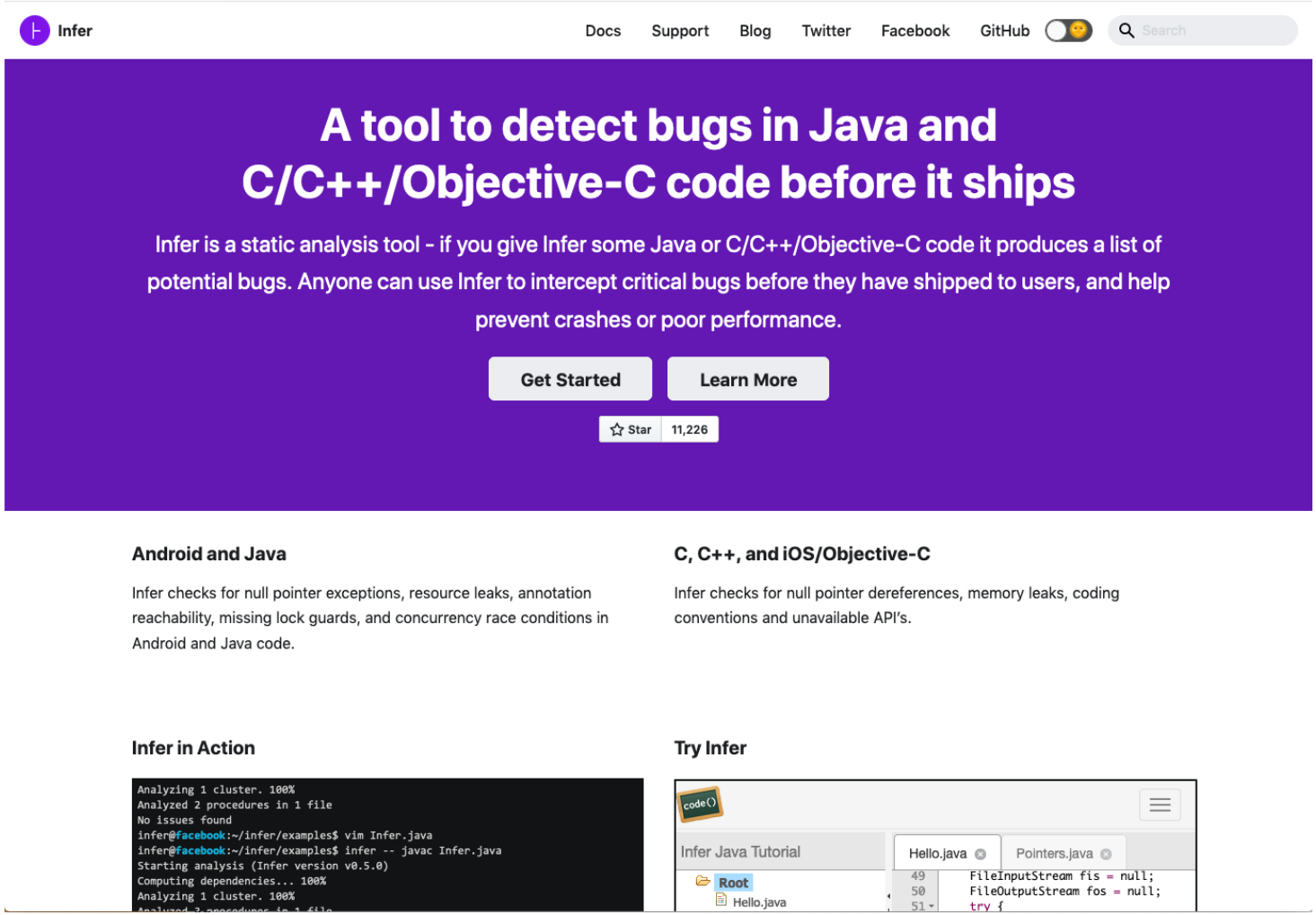


Problemi pratici

fib.key | fact.key | multiplication.key

```
\functions{
  int fact(int);
}
\programVariables{
  int x;
  int y;
  int z;
}
/* invariante:
  fact(0) = 1 &
  \forall int n; (0 < n -> fact(n) = fact(n - 1) * n) &
  0 <= z & z <= x & y = fact(z)
*/
\hoare{
{
  fact(0) = 1 &
  \forall int n; (0 < n -> fact(n) = fact(n - 1) * n) &
  0 <= x
}
\[{
      y = 1;
      z = 0;
      while (z < x) {
        z = z + 1;
        y = y * z;
      }
}\]
  {y = fact(x)}
}
```

Tool e progetti: Infer

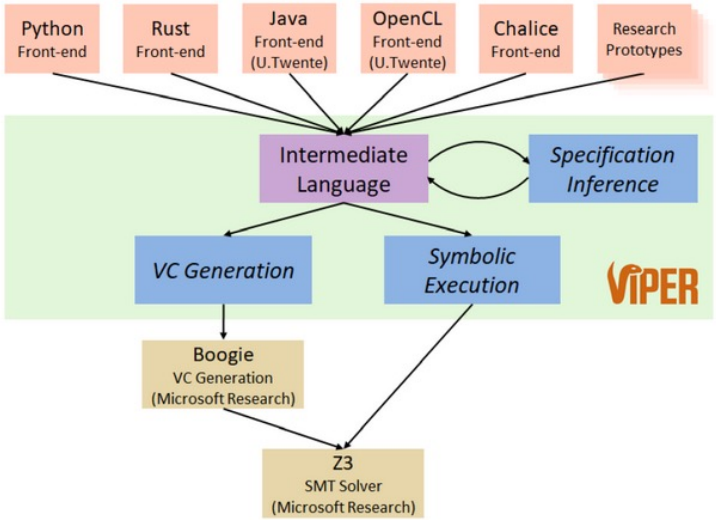


Tool e progetti: Viper

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 →

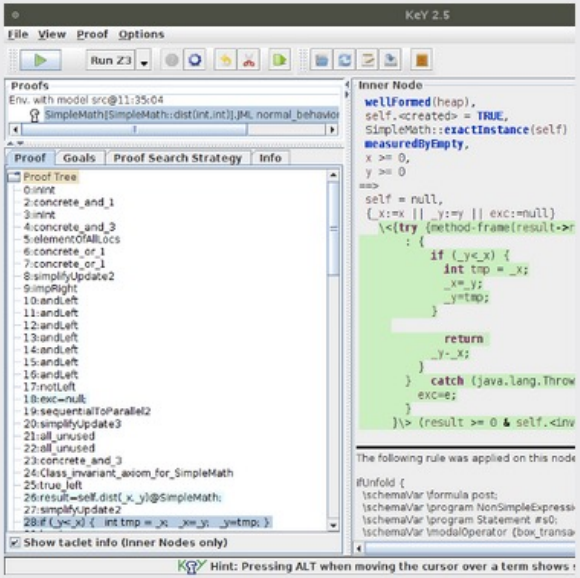
Tool e progetti: Key

The KeY Project | News | Applications | The KeY Book | Publications | About | Downloads

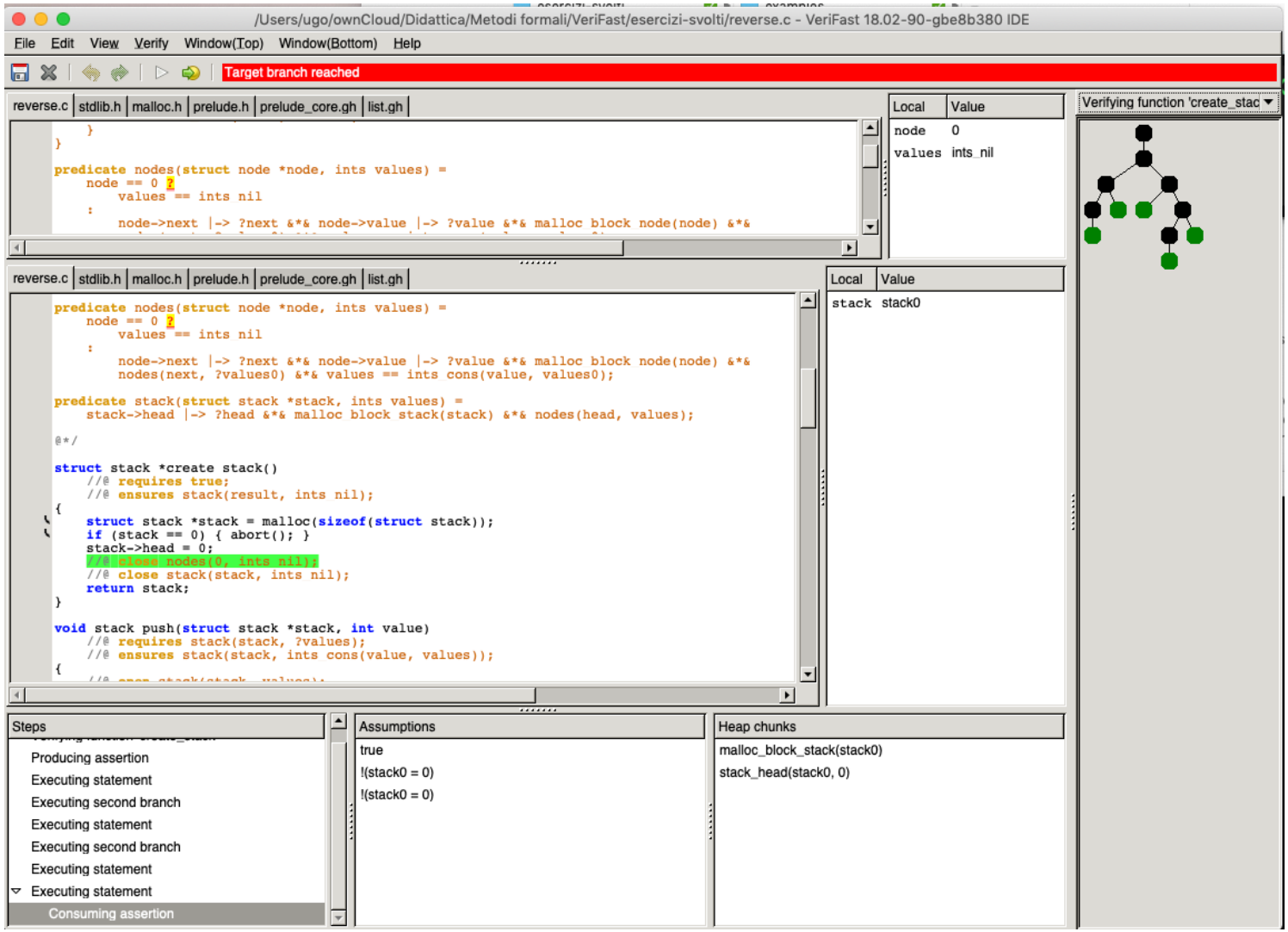
Functional Verification
We can do better than testing!

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!



Tool e progetti: VeriFast



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

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

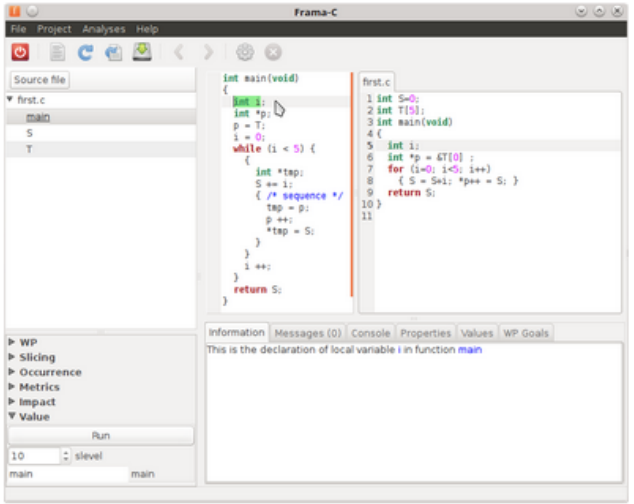
When invoked with the command-line:

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

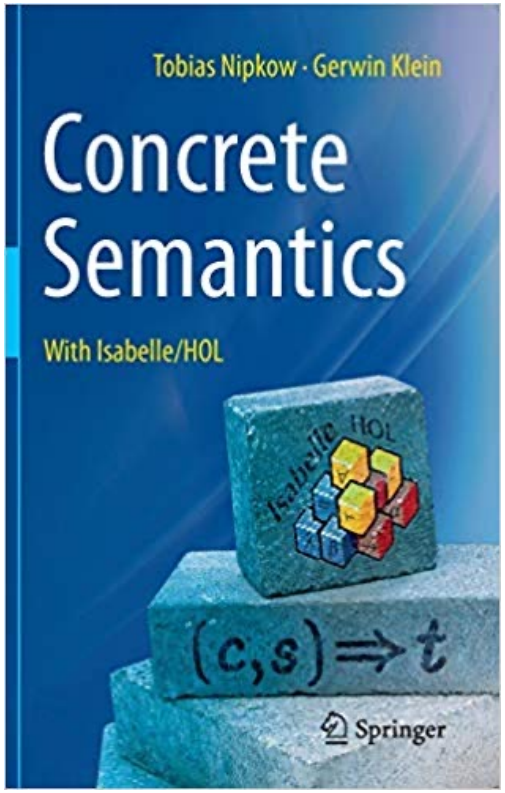
Frama-C creates an analysis project for the file first.c.

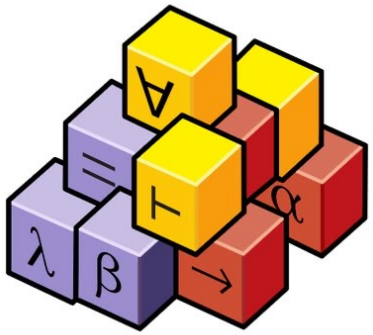
The `-eva` option on the command-line causes the Eva plug-in to run and have its results ready before the interface appears.

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.



Isabelle

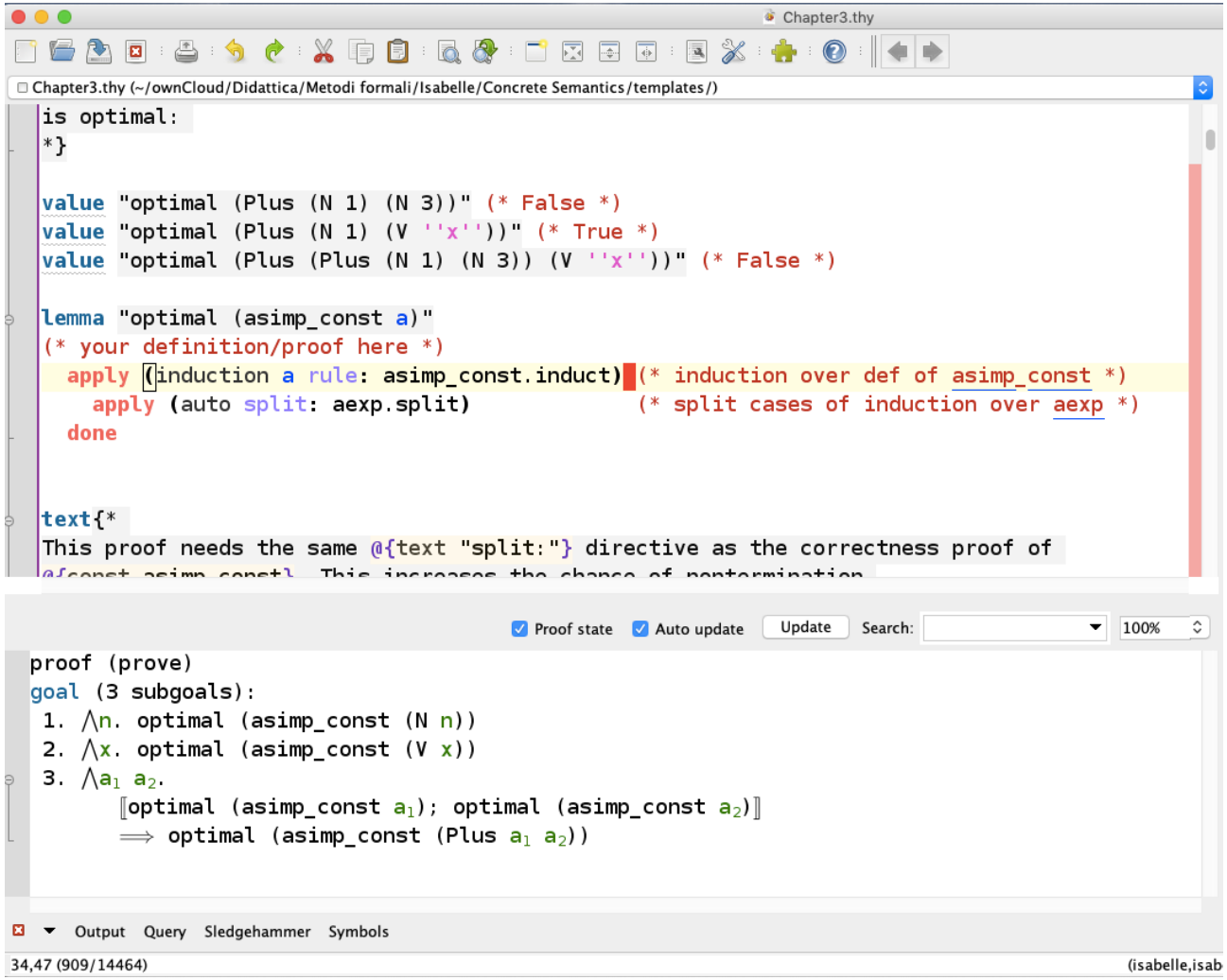




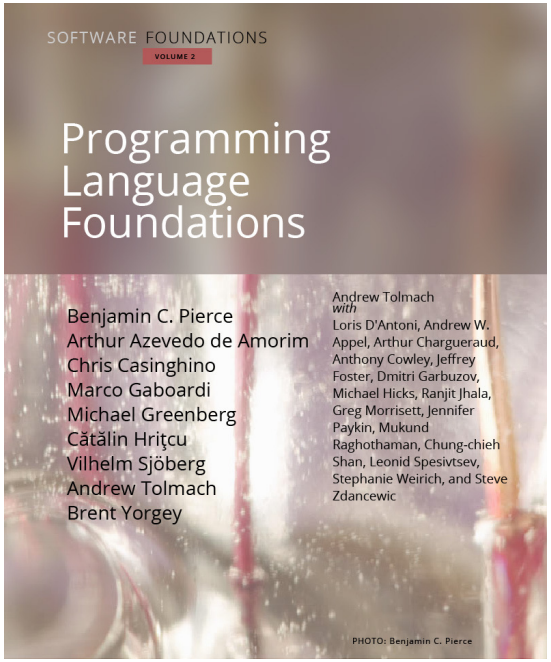
Tobias Nipkow, Gerwin Klein
Concrete Semantics With Isabelle HOL
Springer

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

Isabelle


Chapter3.thy
Chapter3.thy (~/ownCloud/Didattica/Metodi formali/Isabelle/Concrete Semantics/templates/)
is optimal:
*}
value "optimal (Plus (N 1) (N 3))" (* False *)
value "optimal (Plus (N 1) (V ''x''))" (* True *)
value "optimal (Plus (Plus (N 1) (N 3)) (V ''x''))" (* False *)
lemma "optimal (asimp_const a)"
(* your definition/proof here *)
apply (induction a rule: asimp_const.induct) (* induction over def of asimp_const *)
apply (auto split: aexp.split) (* split cases of induction over aexp *)
done
text{*
This proof needs the same @{text "split:"} directive as the correctness proof of
Proof state
Auto update
Update
Search:
100%
proof (prove)
goal (3 subgoals):
1. ⋀n. optimal (asimp_const (N n))
2. ⋀x. optimal (asimp_const (V x))
3. ⋀a1 a2.
⟦optimal (asimp_const a1); optimal (asimp_const a2)⟧
⟹ optimal (asimp_const (Plus a1 a2))
Output
Query
Sledgehammer
Symbols
34,47 (909/14464)
(isabelle,isab

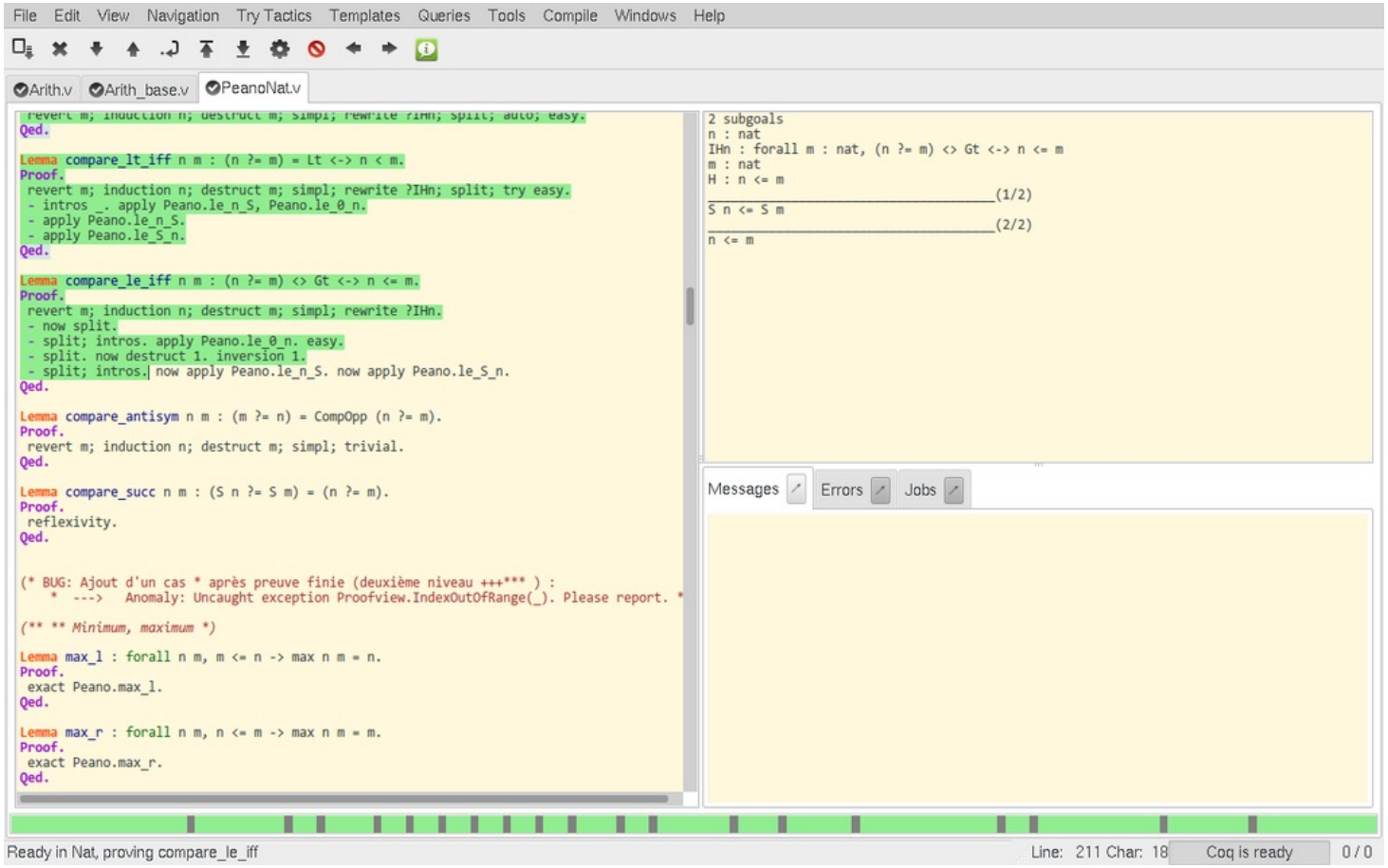
Coq



Benjamin Pierce ed altri
Programming Language Foundations

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

Coq


File Edit View Navigation Try Tactics Templates Queries Tools Compile Windows Help
Arith.v
Arith_base.v
PeanoNat.v
Qed.
Lemma compare_lt_iff n m : (n ?= m) = Lt <-> n < m.
Proof.
revert m; induction n; destruct m; simpl; rewrite ?IHn; split; try easy.
- intros _. apply Peano.le_n_S, Peano.le_0_n.
- apply Peano.le_n_S.
- apply Peano.le_S_n.
Qed.
Lemma compare_le_iff n m : (n ?= m) <> Gt <-> n <= m.
Proof.
revert m; induction n; destruct m; simpl; rewrite ?IHn.
- now split.
- split; intros. apply Peano.le_0_n. easy.
- split. now destruct 1. inversion 1.
- split; intros. now apply Peano.le_n_S. now apply Peano.le_S_n.
Qed.
Lemma compare_antisym n m : (m ?= n) = CompOpp (n ?= m).
Proof.
revert m; induction n; destruct m; simpl; trivial.
Qed.
Lemma compare_succ n m : (S n ?= S m) = (n ?= m).
Proof.
reflexivity.
Qed.
(* BUG: Ajout d'un cas * après preuve finie (deuxième niveau +++***) :
* ---> Anomaly: Uncaught exception Proofview.IndexOutOfRange(_). Please report. *
(** ** Minimum, maximum *)
Lemma max_l : forall n m, m <= n -> max n m = n.
Proof.
exact Peano.max_l.
Qed.
Lemma max_r : forall n m, n <= m -> max n m = m.
Proof.
exact Peano.max_r.
Qed.
2 subgoals
n : nat
IHn : forall m : nat, (n ?= m) <> Gt <-> n <= m
m : nat
H : n <= m
(1/2)
S n <= S m
(2/2)
n <= m
Messages
Errors
Jobs
Ready in Nat, proving compare_le_iff
Line: 211 Char: 18
Coq is ready
0 / 0

Dimostrazioni come programmi

A proof written as a functional program

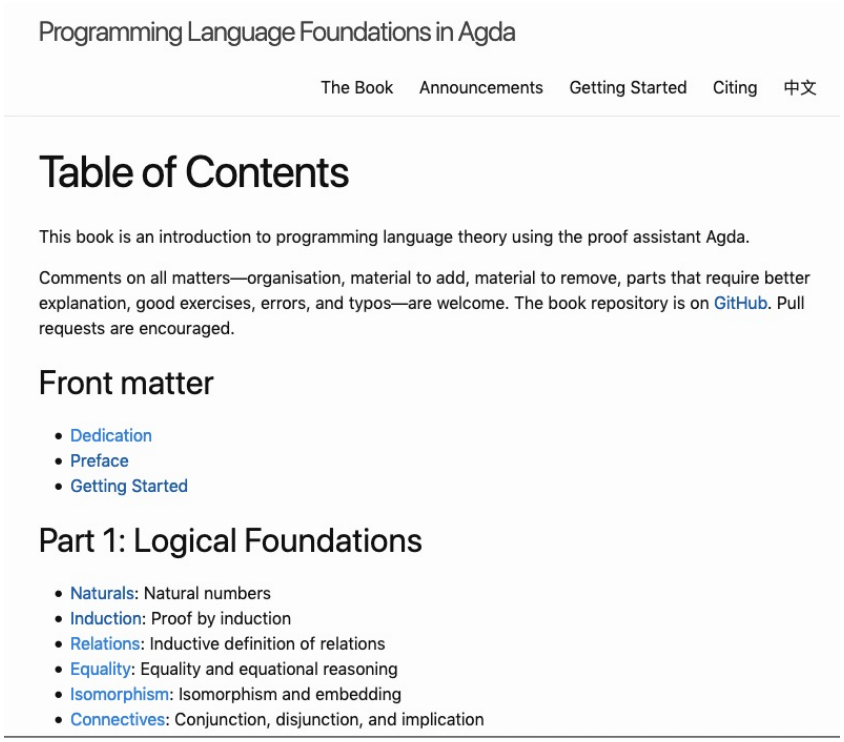
```
plus_comm =
fun n m : nat =>
nat_ind (fun n0 : nat => n0 + m = m + n0)
  (plus_n_0 m)
  (fun (y : nat) (H : y + m = m + y) =>
   eq_ind (S (m + y))
     (fun n0 : nat => S (y + m) = n0)
     (f_equal S H)
     (m + S y)
     (plus_n_Sm m y)) n
     : forall n m : nat, n + m = m + n
```

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?

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

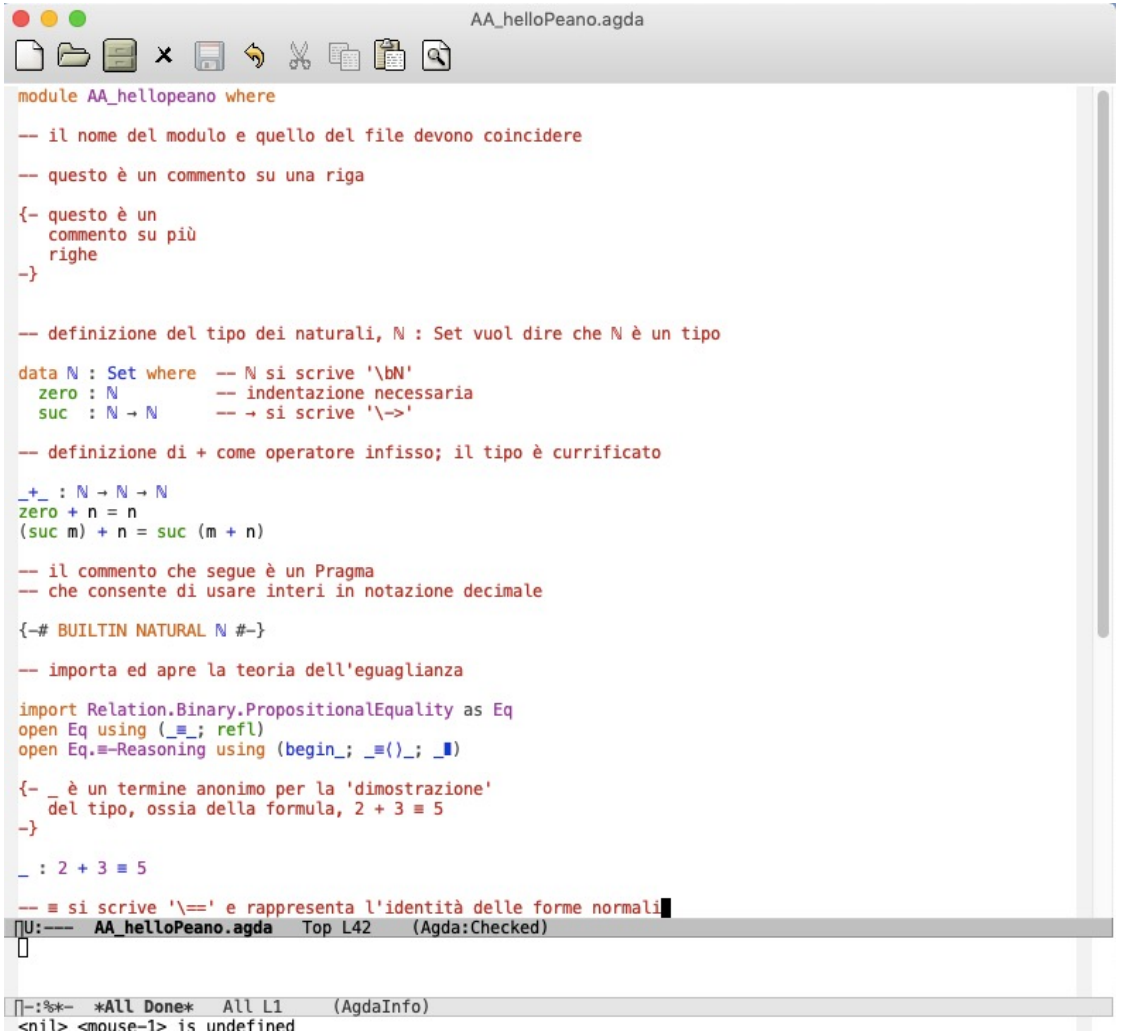
Part 1: Logical Foundations

- 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'eguaglianza
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
U:--- 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 *