FITR304 - Software Validation

Deductive methods for proving imperative programs

Christophe Garion
DMIA – ISAE

License CC BY-NC-SA 3.0

Quotes...

Beware of bugs in the above code; I have only proved it correct, not tried it.

Donald Knuth, 1977

If you find that you're spending almost all your time on theory, start turning some attention to practical things; it will improve your theories. If you find that you're spending almost all your time on practice, start turning some attention to theoretical things; it will improve your practice.

Donald Knuth

Outline

1 - Introduction on formal methods

1. **Why formal methods?**
2. **Programming languages semantics**
3. **Some techniques**
4. **Agenda**

Outline of part 1 - Introduction on formal methods

1. **Why formal methods?**
2. **Programming languages semantics**
3. **Some techniques**
4. **Agenda**

Critical softwares

Software is **critical** in lots of domains: aerospace, health care, defense...

Failures in critical softwares may lead to:

- loss of money
- mission loss
- lifes loss

Question

What are the challenges to build **reliable** softwares?

Software is particular

Good practises from civil eng.

- precise calculations/estimations of forces, stress, etc.
- hardware redundancy ("make it a bit stronger than necessary")
- robust design (single fault not catastrophic)
- clear separation of subsystems
- follows design patterns that are proven to work

Software is particular

...that do not work for software

- software systems compute **non-continuous** functions
  ➥ single bit-flip may change behaviour completely
- redundancy as replication does not help against bugs
- no physical or modal separation of subsystems
  ➥ local failures often affect whole system
- software designs have **very high logic complexity**
- most SW engineers untrained in **correctness**
- **cost efficiency** more important than reliability
- **design practice** for reliable software in **immature** state

A central strategy: testing

Testing against **bugs** and **external faults**.

But:

- testing can **show the presence** of bugs, **not their absence**
- test cases are difficult to produce when searching rare/unexpected faults
- testing is expensive

Example: how do you verify that a sort program with the following signature is correct?

```
void sort (int* array, int n) {
    ...
}
```

➥ not so easy...

Sorting: a mathematical formalization

✎

We have first to "mathematically characterize" our sorting algorithm.

Definition (sorting a sequence)

Let s be a sequence of elements of type E, n be the length of s and $\prec$ a total order on E, then the function $sort$ applied to s returns a sequence s' that is a permutation of s and is sorted w.r.t. $\prec$.

OK, this is a clear specification for sorting, but can **you** write a more precise specification w.r.t. the programming language we use?

What are formal methods?

Definition (informal, from Clarke and Wing 1996)

Formal methods are **mathematically-based languages**, **techniques** and **tools** to verify software systems.

Clarke, Edmund M. and Jeannette. M. Wing (1996).
Formal Methods: State of the Art and Future Directions.
Technical Report CMU-CS-96-178.
Department of Computer Science, Carnegie-Mellon University.

Are formal methods really used (and useful)?

YES, e.g.:

- railway signalling and train control
- banking systems
- Airbus A380 with SCADE, CAVEAT and ASTRÉE
- Microsoft SLAM project and Static Driver Verifier (SDV) tool
- the SeL4 microkernel project at NICTA
- the INRIA CompCert project

Woodcock, Jim et al. (2009).
"Formal Methods: Practice and Experience".
In: **ACM Computing Surveys** 41.4,
Pp. 1–40.

Are formal methods really used (and useful)?

YES, e.g.:

- railway signalling and train control
  - RER Line A (1989), retro-engineering and formal proof
    - ➥ 10 unsafe bugs found
  - Line 14 (METEOR) of the Paris Métro (1999), developped using the B method for safety-critical parts
    - ➥ no unit or integration tests
    - ➥ delivery of a safe software at first shot
  - Roissy Airport shuttle (2007)
- banking systems
- Airbus A380 with SCADE, CAVEAT and ASTRÉE
- Microsoft SLAM project and Static Driver Verifier (SDV) tool
- the SeL4 microkernel project at NICTA
- the INRIA CompCert project

Are formal methods really used (and useful)?

YES, e.g.:

- railway signalling and train control
- banking systems
  Mondex Smart Card (1990), a smartcard-based electronic cash system
  - ➥ proof using Z
  - ➥ high-level of security
  - ➥ revived as a pilot for the Grand Challenge in Verified Software
- Airbus A380 with SCADE, CAVEAT and ASTRÉE
- Microsoft SLAM project and Static Driver Verifier (SDV) tool
- the SeL4 microkernel project at NICTA
- the INRIA CompCert project

Are formal methods really used (and useful)?

YES, e.g.:

- railway signalling and train control
- banking systems
- Airbus A380 with SCADE, CAVEAT and ASTRÉE
  - ➥ 70% of code generated automatically, significant decrease in coding errors
  - ➥ high-level of security
  - ➥ revived as a pilot for the Grand Challenge in Verified Software
- Microsoft SLAM project and Static Driver Verifier (SDV) tool
- the SeL4 microkernel project at NICTA
- the INRIA CompCert project

Are formal methods really used (and useful)?

YES, e.g.:

- railway signalling and train control
- banking systems
- Airbus A380 with SCADE, CAVEAT and ASTRÉE
- Microsoft SLAM project and Static Driver Verifier (SDV) tool
  - ➥ drivers formally verified, the end of Blue Screen of Death (almost ☺)

  People life ☺: J. Wing is now Corporate Vice President of Microsoft Research, hence the importance of FM for Microsoft...
- the SeL4 microkernel project at NICTA
- the INRIA CompCert project

Are formal methods really used (and useful)?

YES, e.g.:

- railway signalling and train control
- banking systems
- Airbus A380 with SCADE, CAVEAT and ASTRÉE
- Microsoft SLAM project and Static Driver Verifier (SDV) tool
- the SeL4 microkernel project at NICTA
  - ➥ formal proof of functional correctness of the Kernel
  - ➥ a high-assurance drone is being built

> Heiser, Gernot and Kevin Elphinstone (2016).
> "L4 Microkernels: The Lessons from 20 Years of Research and Deployment".
> In: **ACM Transactions on Computer Systems** 34.1, 1:1–1:29.
> DOI: 10.1145/2893177.

- the INRIA CompCert project

Are formal methods really used (and useful)?

YES, e.g.:

- railway signalling and train control
- banking systems
- Airbus A380 with SCADE, CAVEAT and ASTRÉE
- Microsoft SLAM project and Static Driver Verifier (SDV) tool
- the SeL4 microkernel project at NICTA
- the INRIA CompCert project
  - ➥ have you ever looked at GCC's bugs (`https://gcc.gnu.org/bugzilla/`)?
  - ➥ a **proven** compiler for a realistic part of the C programming language

Yang, Xuejun et al. (2011).
"Finding and understanding bugs in C compilers".
In: **Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)** .
DOI: 10.1145/1993498.1993532.

Outline of part 1 - Introduction on formal methods

1. Why formal methods?
2. **Programming languages semantics**
3. Some techniques
4. Agenda

What is semantics?

In order to prove properties on programs, we need to define precisely the semantics of the underlying programming language.

Floyd, Robert W. (1967).
"Assigning meanings to programs".
In: **Mathematical aspects of computer science**.
Ed. by J. T. Schwartz.
American Mathematical Society,
Pp. 19–32.
ISBN: 0821867288.

There are of course several semantics for programming languages.

Operational semantics (small steps)

Operational semantics defines a program semantics with **states**, i.e. functions from memory locations (variables) to values.

Rules define the semantics of the constructs of the program:

$$\frac{\langle b, \sigma\rangle \to \text{true} \quad \langle c_0, \sigma\rangle \to \sigma'}{\langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma\rangle \to \sigma'}$$

$$\frac{\langle b, \sigma\rangle \to \text{false} \quad \langle c_1, \sigma\rangle \to \sigma'}{\langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma\rangle \to \sigma'}$$

This leads to **traces**, i.e. sequences of states.

Proofs can be done using this formal system about the final state of the program.

Trace semantics

We can characterize the set of traces of a program:

$$\{s_0 \rightarrow s_n \mid \forall i \in [0, n-1]\ (s_i, s_{i+1}) \in f_{op} \text{ and } s_0 \in Init\}$$

where f_{op} is the set of transitions from state to state.

We can also use **collecting semantics**, i.e. be only interested in reachable states.
This semantics is useful as it can be used to guarantee that a particular property holds for all reachable states (an invariant for instance).

Axiomatic semantics

In axiomatic semantics, the semantics of the program is defined with **Hoare triples**:

$$\{\varphi\} \;\; \texttt{P} \;\; \{\psi\}$$

φ and ψ (resp. the precondition and the postcondition of P) are mathematical formulas.

Rules are expressed using these triples:

$$\frac{\{\varphi \land C\} \;\; \texttt{P} \;\; \{\psi\} \qquad \{\varphi \land \neg C\} \;\; \texttt{Q} \;\; \{\psi\}}{\{\varphi\} \;\; \texttt{if C then P else Q fi} \;\; \{\psi\}} \text{ (Cond.)}$$

This formal system can be used to derive proofs about programs.

Outline of part 1 - Introduction on formal methods

Model checking

In model checking, we have:

- a model of the system/the program
- a property to verify

We want to verify exhaustively that the model verifies the property.

For instance, the model can be the collecting traces and the property can expressed in temporal logic.

Abstract interpretation

Abstract interpretation is a sound approximation of the semantic of a program.
The idea is to "encompass" the traces of the program into an more abstract domain.

For instance, if you want to proof that there is no division-by-zero in your program, you may restrict the integers to two values, `0` and `others` and **statically** verify the property.

Abstract interpretation is also used in compilers for optimizations purposes.

Deductive methods: what is that?

Definition (simple but efficient...)

Deductive program verification is the art of turning the correctness of a program into a mathematical statement and then proving it.

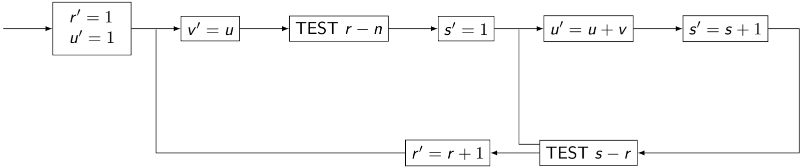
Filliâtre, Jean-Christophe (2011).
"Deductive Program Verification".
Habilitation à diriger les recherches. Université Paris-Sud 11.

We have thus to answer the following questions:

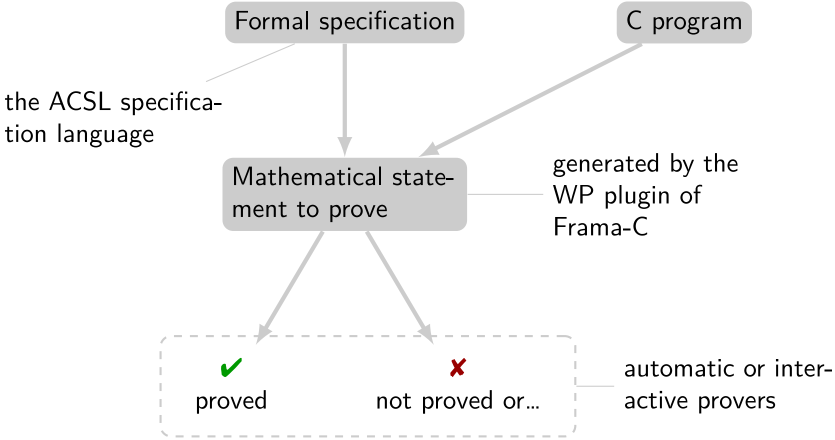
- what is a **proof**?
- how can we **turn the correctness of a program into a mathematical statement**?
- can we **automatically prove** the correctness of a program?

Deductive methods: is it old?

Turing, Alan Mathison (1949).
Checking a large routine.
Report of a Conference on High Speed Automatic Calculing Machines.
Cambridge: Mathematical Laboratory,
Pp. 67–69.



Deductive methods: the big picture



Outline of part 1 - Introduction on formal methods

1. **Why formal methods?**
2. **Programming languages semantics**
3. **Some techniques**
4. **Agenda**

FITR304: formal methods

During the lecture, you will choose to study deeper **one** of the following formal methods:

1. **deductive methods** (C. Garion, ISAE/DMIA)
   - ➥ how can we prove that imperative programs are correct w.r.t. to a specification?
2. **model checking** (J. Brunel, ONERA/DTIM)
   - ➥ given a model of a system, check if a given property is respected or not (mostly temporal properties)
3. **abstract interpretation** (P.-L. Garoche, ONERA/DTIM)
   - ➥ a theory of sound approximation of the semantics of computer programs

FITR304: agenda and evaluation

- 6×2 hours sessions are dedicated to the track you have chosen (groups of 3-4 students by track)
- a global miniproject on a rover: each FM will study one part of the rover architecture (50% of the final note)
- final presentation (50% of the final note) + MQC on **02/27/2017**
- industrial feedback conference

2 - Formal proof

5. **Formal systems**
6. **Natural deduction for PL:** $\mathcal{NK}$
7. **Natural deduction for FOL:** $\mathcal{NK}$

What is a proof?

Definition (informal, from Wikipedia...)

A proof is sufficient evidence or an argument for the truth of a proposition.

Nice, but:

- what is an argument?
- what is truth?
- what is a proposition?

All those notions are formally defined in **mathematical logic**.

What is a proof?

Definition (informal, from Wikipedia...)

A proof is sufficient evidence or an **argument** for the **truth** of a **proposition**.

Nice, but:

- what is an argument?
- what is truth?
- what is a proposition?

All those notions are formally defined in **mathematical logic**.

What is mathematical logic?

Informal definition

Mathematical logic is the study of the validity of an **argument** as a mathematical object.

First question: what is an argument?

An argument is composed of:

- a set of **declarative** sentences called **premises**
- a word, **therefore**
- a **declarative** sentence called **conclusion**

What is mathematical logic?

Informal definition

Mathematical logic is the study of the **validity** of an argument as a mathematical object.

Second question: what is validity?

Validity of an argument can be defined:

- in **model theory**: is the conclusion true when premises are?
- in **proof theory**: does the argument respect some rules?

A multi-disciplinary field

Philosophy ⟷ Mathematics ⟷ Computer Science

- ~~what is true?~~
- ~~what is false?~~

- what is a proof ?
- what mathematical structures do we need to define a proof?
- is this proof correct?

- is this program correct?
- can I automatically produce code that respect those specifications?
- can we prove automatically that this theorem is true?

Outline of part 2 - Formal proof

5. **Formal systems**
6. **Natural deduction for PL:** $\mathcal{NK}$
7. **Natural deduction for FOL:** $\mathcal{NK}$

What is a formal system?

Definition (formal system)

A formal system is composed of two elements:

- a **formal language** (grammar) defining a set of expressions E
- a **deductive system** or **deductive apparatus** on E

We have thus to define:

- what is a grammar
- what is a deductive system

Grammar

A **formal grammar** is a set of rules describing a **formal language** using a **finite alphabet**.

For instance, the grammar $\{X = \{a, b\}, V = \{S\}, S, \{S \to aS, S \to b\}\}$ describe the language $\{a^n b \mid n \in \mathbb{N}\}$.

There are other formalisms to describe (somme categories of) formal languages: regular expressions, EBNF, inductive definitions etc.

In the following, we will use **inductive definitions**.

Inductive definition

Definition (inductive or recursive definition)

An **inductive definition** of a set E is composed of:

- a **base case** of the definition which defines elementary elements of E
- an **inductive clause** of the definition which defines elements of E using other elements of E defined with a **finite number of steps** n and **operations**
- an **extremal clause** that says that E is the **smallest set** built using the base case and the inductive clause.

Some definitions by induction ✎

Exercise

Define $\mathbb{N}$ by induction.

Exercise

Define binary trees by induction.

Structural induction

Given a set E defined inductively, we can prove properties on elements of E using **structural induction**.

Definition (structural induction)

Let E be a set defined inductively and $\mathcal{P}$ a property on elements of E to be proved. If:

- $\mathcal{P}$ can be proved to be true on each base case
- if we suppose that $\mathcal{P}$ is true on elements built with n steps then $\mathcal{P}$ is true on elements that can be built with $n + 1$ steps

then $\mathcal{P}$ is true for every element of E.

A proof by structural induction ✎

Exercise

Prove the following property of binary trees: "the number n of nodes in a binary tree of height h is at least $n = h$ and at most $n = 2^h - 1$ where h is the depth of the tree".

Induction example: alphabet of $\mathcal{L}_{PL}$

Definition (alphabet of $\mathcal{L}_{PL}$)

The alphabet of $\mathcal{L}_{PL}$is composed of:

- an infinite and enumerable set of **propositional variables** noted $Var = \{p, q, r, \ldots\}$
- two **constants** noted $\top$ (top/true) and $\bot$ (bottom/false)
- **logical connectors**:

$\neg$	negation
$\vee$	or/disjunction
$\wedge$	and/conjunction
$\rightarrow$	implication
$\leftrightarrow$	logical equivalence

- **parentheses** ()

Induction example: wff of $\mathcal{L}_{PL}$

Definition (well formed formulas)

- if p is a propositional variable, then p is a wff. p is an **atomic formula** or **atom**.
- $\top$ and $\bot$ are wff.
- if φ is a wff, alors $(\neg\varphi)$ is a wff.
- if φ and ψ are wff, then $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$, $(\varphi \rightarrow \psi)$ and $(\varphi \leftrightarrow \psi)$ are wff.

Modelling exercise ✎

Exercise

Use propositional language to model the following declarative sentences.

1. it is raining and it is cold.
2. if he eats too much, he will be sick.
3. it is sunny but it is cold.
4. if it is cold, I take my jacket.
5. I take either a jacket, either an umbrella.
6. it is not raining.
7. in autumn, if it is cold then I take a jacket.
8. in winter, I take a jacket only if it is cold.
9. if Peter does not forget to book tickets, we will go to theater.
10. if Peter does not forget to book tickets and if we find a baby-sitter, we will go to theater.
11. he went, although it was very hot, but he forgot his water bottle.
12. when I am nervous, I practise yoga or relaxation. Someone practising yoga also practises relaxation. So when I do not practise relaxation, I am calm.
13. my sister wants a black and white cat.

Definition (deductive system)

A **deduction system** (or **inference system**) on a set E is composed of a set of rules used to derive elements of E from other elements of E. They are called **inference rules**.

If an inference rule allows to derive e_{n+1} (conclusion) from $\mathcal{P} = \{e_1, \ldots, e_n\}$ (premises), it will be noted as follows:

$$\frac{e_1 \;\; e_2 \;\; \ldots \;\; e_n}{e_{n+1}}$$

When an inference rule is such that $\mathcal{P} = \emptyset$ it is called an **axiom**.

If e_1 is an axiom, it is either noted $\frac{}{e_1}$ or simply e_1.

Deductive system

Intuition

A rule $\dfrac{e_1 \;\; e_2}{e_3}$ means:

- from e_1 and e_2 you can deduce e_3
- to prove e_3, it is sufficient to prove e_1 and to prove e_2

If e can be produced only from axioms using inference rules, then e is called a **theorem** of $\mathcal{F}$ (same as in maths!). This is noted $\vdash_{\mathcal{F}} e$.

To represent a proof, we will use **trees**. For instance, considering the classical Hilbert system with Modus Ponens rule, here is a proof of $p \to p$:

$$\dfrac{\dfrac{(p \to (p \to p)) \to ((p \to ((p \to p) \to p)) \to (p \to p)) \qquad p \to (p \to p)}{(p \to ((p \to p) \to p)) \to (p \to p)} \qquad p \to ((p \to p) \to p)}{p \to p}$$

Outline of part 2 - Formal proof

5. **Formal systems**

6. **Natural deduction for PL: $\mathcal{NK}$**
   - Deductive system
   - A new language: sequents for NK

7. **Natural deduction for FOL: $\mathcal{NK}$**

5 **Formal systems**

6 **Natural deduction for PL: $\mathcal{NK}$**

- Deductive system
- A new language: sequents for NK

7 **Natural deduction for FOL: $\mathcal{NK}$**

Natural deduction is a formal system that has evolved from axiomatic formal systems developped by 19th century mathematicians like Hilbert or Russell.

G. Gentzen has proposed a more “intuitive” formal system, **natural deduction** (*natürliches Schließen*).

Gentzen, Gerhard (1934).
“Untersuchungen über das logische Schließen I”.
In: **Mathematische Zeitschrift** 39.2,
Pp. 176–210.

— (1935).
“Untersuchungen über das logische Schließen II”.
In: **Mathematische Zeitschrift** 39.3,
Pp. 405–431.

Rules for natural deduction

Definition (introduction and elimination rules)

$$\frac{A \qquad B}{A \wedge B}\ (I_\wedge) \qquad \frac{A \wedge B}{A}\ (E^1_\wedge) \qquad \frac{A \wedge B}{B}\ (E^2_\wedge)$$

$$\frac{A}{A \vee B}\ (I^1_\vee) \qquad \frac{B}{A \vee B}\ (I^2_\vee) \qquad \frac{A \vee B \qquad \begin{matrix}[A] \\ \vdots \\ C\end{matrix} \qquad \begin{matrix}[B] \\ \vdots \\ C\end{matrix}}{C}\ (E_\vee)$$

$$\frac{\begin{matrix}[A] \\ \vdots \\ B\end{matrix}}{A \rightarrow B}\ (I_\rightarrow) \qquad \frac{A \rightarrow B \qquad A}{B}\ (E_\rightarrow)$$

What are those [ ] everywhere?

Some premises in rules $(E_\vee)$ and $(I_\rightarrow)$ are between brackets. What does that mean?

The hypotheses between brackets are used for **hypothetical derivation** and are **discharged** when using the rule. They are not **real hypothesis** for the derivation.

For instance,

$$\dfrac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \rightarrow B}\;(I_\rightarrow)$$

means: "if assuming A you can deduce that B, then you can deduced $A \rightarrow B$".

What are those [ ] everywhere?

Some premises in rules $(E_\vee)$ and $(I_\rightarrow)$ are between brackets. What does that mean?

The hypotheses between brackets are used for **hypothetical derivation** and are **discharged** when using the rule. They are not **real hypothesis** for the derivation.

N.B. (important)

The discharged hypothesis are only valid in the rule context and cannot be used for instance below the rule application.

N.B.

When introducing hypothesis (not premises of the argument), you have to discharge them to obtain a valid proof.

How to discharge hypotheses

In order to remember where hypotheses are discharged, rule numbering can be used:

$$\dfrac{\dfrac{\dfrac{[a]^1 \qquad [b]^2}{a \wedge b}\,(I_\wedge)}{b \rightarrow (a \wedge b)}\,(I_\rightarrow)^2}{a \rightarrow (b \rightarrow (a \wedge b))}\,(I_\rightarrow)^1$$

Subdeductions are hypothetical: in the previous example, $b \rightarrow (a \wedge b)$ can be deduced under the assumption a.

From minimal system to classical system

The previous system is **minimal**: it does not correspond to classical logic. The following rules have to be added.

Definition (rules for intuitionist system)

$$\frac{\perp}{A}\,(E_\perp) \qquad\qquad \neg A \equiv A \rightarrow \perp$$

Definition (rules for classical system)

$$\frac{}{A \vee \neg A}\,(EM) \qquad\qquad \frac{\begin{array}{c}[\neg A]\\ \vdots\\ \perp\end{array}}{A}\,(A)$$

Let's prove some formulae! ✎

Exercise

Prove the following PL formulas in $\mathcal{NK}$:

$$(a \to (b \to c)) \to ((a \to b) \to (a \to c))$$
$$((a \lor b) \to c) \to (b \to c)$$
$$((a \lor b) \land (a \to c) \land (b \to c)) \to c$$
$$a \to \neg\neg a$$

Try it on your computer?

Adopt a Panda!

The panda (Ailuropoda melanoleuca, lit. "black and white cat-foot"), also known as the giant panda to distinguish it from the unrelated red panda, is a bear native to central-western and south western China. (Wikipedia, 2012.)

Gasquet, Olivier, François Schwarzentruber, and Martin Strecker (2011).
Panda: Proof Assistant for Natural Deduction for All.
http://www.irit.fr/panda/.

Outline of part 2 - Formal proof

5. **Formal systems**

6. **Natural deduction for PL: $\mathcal{NK}$**
   - Deductive system
   - A new language: sequents for NK

7. **Natural deduction for FOL: $\mathcal{NK}$**

Sequent

Gentzen also proposed a new language based on $\mathcal{L}_{PL}$in order to make proof in $\mathcal{NK}$ easier (in particular for discharged hypotheses).

The main idea of this new language is to “embark” the hypotheses you are using in the “formulas”.

Definition (sequent)

A **sequent** is composed of a finite set of wff Γ and a wff φ and is denoted by $\Gamma \vdash \varphi$.

The intuition behind sequent is the following: $\Gamma \vdash \varphi$ means “φ can be deduced from hypotheses Γ”.
Γ is also called **the context**.

Some (false) notations are used: for instance $\Gamma, \psi \vdash \varphi$ is used for $\Gamma \cup \{\psi\} \vdash \varphi$.

Rules for sequent-based $\mathcal{NK}$

Definition (axiom and structural rule)

$$\frac{}{A \vdash A}\ (Hyp) \qquad\qquad \frac{\Gamma \vdash A}{\Gamma, B \vdash A}\ (Aff)$$

Rules for sequent-based $\mathcal{NK}$

Definition (logical rules)

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,(I_\wedge) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\,(E^1_\wedge) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\,(E^2_\wedge)$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\,(I^1_\vee) \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\,(I^2_\vee)$$

$$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}\,(E_\vee)$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\,(I_\to) \qquad \frac{\Gamma, A \vdash B \qquad \Gamma \vdash A}{\Gamma \vdash B}\,(E_\to)$$

$$\frac{\Gamma, A \to \bot \vdash \bot}{\Gamma \vdash A}\,(TE) \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash A}\,(E_\bot)$$

NK with sequents: example

With the previous example:

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{a \vdash a}\,(Hyp)}{a, b \vdash a}\,(Aff) \qquad \dfrac{\dfrac{}{b \vdash b}\,(Hyp)}{a, b \vdash b}\,(Aff)}{\dfrac{a, b \vdash a \wedge b}{a \vdash b \rightarrow (a \wedge b)}\,(I_{\rightarrow})}\,(I_{\wedge})}{\vdash a \rightarrow (b \rightarrow (a \wedge b))}\,(I_{\rightarrow})
$$

Automatic proof of the previous wffs?

Building proofs of the previous formulas is not automatic and can be fastidious. Is there an algorithm to prove that a wff is a theorem?

This field of study is called **automated theorem proving**. Some theorem provers:

- The E Theorem Prover (`http://www.eprover.org`)
- Vampire (`http://www.vprover.org`)
- SPASS (`http://www.spass-prover.org`)

Notice that:

- theorem proving is decidable for PL
- this problem is strongly related to the SAT problem
- the provers presented here also work with First-Order Logic

Use SPASS on our examples

Let us try SPASS on our examples.

The SPASS team (2014).
SPASS: An Automated Theorem Prover for First-Order Logic with Equality.
http://www.spass-prover.org.

Use SPASS on our examples

Let us try SPASS on our examples.

```
begin_problem(pl_1).

list_of_descriptions.
  name({*(A -> (B -> C)) -> ((A -> B) -> (A -> C))*}).
  author({*Christophe Garion*}).
  status(satisfiable).
  description({*Prove (A -> (B -> C)) -> ((A -> B) -> (A -> C))...*}).
end_of_list.

list_of_symbols.
  predicates[(A,0), (B,0), (C,0)].
end_of_list.

list_of_formulae(conjectures).
  formula(implies(implies(A, implies(B, C)), implies(implies(A, B),
                                                    implies(A, C)))).
end_of_list.

end_problem.
```

Outline of part 2 - Formal proof

5. Formal systems
6. Natural deduction for PL: $\mathcal{NK}$
7. **Natural deduction for FOL: $\mathcal{NK}$**
   - First-order logic language
   - Deductive system

Alphabet

Definition (alphabet)

The alphabet of $\mathcal{L}_{FOL}$ is composed of:

- logical symbols
  - an inifinite and enumerable set $\mathcal{V}$ of individual variables $x, y, \ldots$
  - connectors: $\top, \bot, \neg, \rightarrow, \wedge, \vee, \leftrightarrow$
  - quantifiers: $\exists$, $\forall$
  - , ( )
- non-logical symbols
  - an enumerable set $\mathcal{P}$ of predicate symbols $P, Q, R, \ldots$
  - an enumerable set $\mathcal{F}$ of functions $f, g, h, \ldots$
  - an enumerable set $\mathcal{C}$ of individual constants $a, b, c, \ldots$

Signature of a first-order language

Like in the propositional case, $\mathcal{V}$, $\top$, $\bot$, $\neg$, $\vee$, $\rightarrow$, $\leftrightarrow$, $($, $)$ and are called **logical symbols** because their logical meaning is already defined.

On the contrary, $\mathcal{P}$, $\mathcal{F}$ and $\mathcal{C}$ depend on the problem to be modelled and thus the predicate, function and constant symbols are called **non-logical symbols**. It is also called the **signature** $\mathcal{S}$ of the language.

So, when you want to model a problem using $\mathcal{L}_{FOL}$, you first have to define the signature of your language, i.e. $\mathcal{S} = \langle \mathcal{P}, \mathcal{F}, \mathcal{C} \rangle$.

When defining predicates and functions, the arity is often denoted using the $/$ notation:

$P/2$	a predicate P of arity 2
$f/3$	a function f of arity 3

$\mathcal{L}_{FOL}$ terms

An expression is a sequence of symbols.

Some expressions, called **terms**, represents **objects**.

ex: Socrates, John's father, $3+(2+5)$, ...

Definition (term)

The set of terms of $\mathcal{L}_{FOL}$ is defined inductively by:

- a variable is a term
- a constant is a term
- if f is a function symbol with arity m and if $t_1, \ldots, t_m$ are terms, then $f(t_1, \ldots, t_m)$ is a term

Well-formed formulas

Some expressions are interpreted as **assertions**. Those expressions are **well formed formulas** (wffs).

Definition (atomic formula)

If P is a predicate symbol with arity n and if $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is an **atomic formula** of $\mathcal{L}_{FOL}$.

Definition (well formed formula)

The set of wff of $\mathcal{L}_{FOL}$ is defined inductively as follows:

- an atomic formula is a wff
- $\top$ and $\bot$ are wffs
- if φ and ψ are wffs, then $(\neg\varphi)$, $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$, $(\varphi \rightarrow \psi)$ and $(\varphi \leftrightarrow \psi)$ are wffs
- if φ is a wff and x is a variable, then $(Qx\ \varphi)$ where $Q \in \{\forall, \exists\}$ is a wff
  φ is called the **scope of** Qx (cf. later).

Some conventions (as in the PL case)

To simplify the writing, some conventions can be used:

- removing of external parentheses: $(a \wedge b) \rightsquigarrow a \wedge b$
- $\neg$ is written without parentheses: $(\neg a) \rightsquigarrow \neg a$
- connectors are associative from left to right: $((a \wedge b) \wedge c) \rightsquigarrow a \wedge b \wedge c$
- quantifiers sequences can be simplified: $Q_1x(Q_2y\ \varphi) \rightsquigarrow Q_1xQ_2y\ \varphi$

Connectors and quantifiers can be ordered by growing priority like in the PL case:

$$\forall \quad \exists \quad \leftrightarrow \quad \rightarrow \quad \vee \quad \wedge \quad \neg$$

Some remarks on $\mathcal{L}_{FOL}$

Constants can also be viewed as **0-ary functions**, i.e. functions that does not take parameters. We use a distinct set $\mathcal{C}$ to simplify the presentation of FOL semantics.

If you consider a FO language whose signature is the following:

- $\mathcal{C} = \emptyset$
- $\mathcal{F} = \emptyset$
- every predicate symbol P in $\mathcal{P}$ is a 0-ary symbol, i.e. it does not take parameters

then you obtain **propositional logic**. Thus, PL is a subset of FOL.

Exercise

Let E be a set. Model the following mathematical notions using a first-order language. Define precisely the signature of the language.

- $=$ define the "classical" equality relation on E (not easy!)
- $\leq$ is a preorder on E
- $(E, .)$ is a monoid

A correct definition of scope

We have defined the **scope** of a formula $Qx\ \varphi$ to be φ, but is it really the case?

Consider for instance $\forall x\ (P(x) \rightarrow (\exists x\ Q(x)))$. If the intuitive meaning of the scope of $\forall x$ is to define the formula in which you can replace x by "what you want", it is false.

Using the syntax tree, we can define scope in a better way:

Definition (scope)

Let $Qx\ \varphi$ be a wff with $Q \in \{\forall, \exists\}$. The **scope** of Qx in $Qx\ \varphi$ is the subtree of Qx in $\mathcal{ST}(Qx\ \varphi)$ minus the subtrees in $\mathcal{ST}(Qx\ \varphi)$ reintroducing a new quantifier for x.

With this definition the scope of $\forall x$ in $\forall x\ (P(x) \rightarrow (\exists x\ Q(x)))$ is only $P(x)$.

A correct definition of scope

We have defined the **scope** of a formula $Qx\ \varphi$ to be φ, but is it really the case?

Consider for instance $\forall x\ (P(x) \to (\exists x\ Q(x)))$. If the intuitive meaning of the scope of $\forall x$ is to define the formula in which you can replace x by "what you want", it is false.

N.B. (important)

Avoid reintroducing new quantifiers for a previously quantified variable in wff!

For instance, rewrite the previous formula as $\forall x\ (P(x) \to (\exists y\ Q(y)))$ which is unambiguous.

Definition (free and bound variables)

The set BV of **bound variables** and FV of **free variables** of a wff φ are defined inductively as follows:

- if φ is an atomic formula $P(t_1, \ldots, t_n)$, then $BV(\varphi) = \emptyset$ and $FV(\varphi) = \{t_i | i \in \{1, \ldots, n\}$ and t_i is a variable$\}$
- if $\varphi \equiv \neg\varphi_1$ then $BV(\varphi) = BV(\varphi_1)$ and $FV(\varphi) = FV(\varphi_1)$
- if $\varphi \equiv \varphi_1$ conn φ_2 where conn $\in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$ then $BV(\varphi) = BV(\varphi_1) \cup BV(\varphi_2)$ and $FV(\varphi) = FV(\varphi_1) \cup FV(\varphi_2)$
- if $\varphi \equiv Qx\ \varphi_1$ where $Q \in \{\forall, \exists\}$, then $BV(\varphi) = BV(\varphi_1) \cup \{x\}$ and $FV(\varphi) = FV(\varphi_1) - \{x\}$

Definition (closed formula)

A **closed** formula is a formula φ such that $FV(\varphi) = \emptyset$.

Free and bound variables: examples

- free
- bound

$(\exists x\ P(\boxed{x})) \wedge (\forall y\ \neg Q(\boxed{y})) \wedge R(\boxed{z})$

$(\exists x\ P(\boxed{x})) \wedge Q(\boxed{x})$

N.B.

When modelling "real" notions, it is very difficult to use open formulas (i.e. non closed formulas).

Substitutions

As variables are placeholders, we should be able to **replace** them with concrete (or not) **terms**.

Definition (substitution)

Let φ be a wff, x a variable and t a term. $\varphi[x/t]$ denotes the formula obtained by replacing all **free occurrences** of x in φ by t.

You will sometimes find the "contrary" in some textbook, i.e. $[t/x]$ meaning "replace x by t".

Examples:

$$P(x)[x/y] \equiv P(y)$$
$$P(x)[x/x] \equiv P(x)$$
$$(P(x) \to \forall x\ P(x))[x/y] \equiv (P(y) \to \forall x\ P(x))$$

Using the syntax tree of φ, it means replacing all x nodes by the syntax tree of t.

Free substitutions

Substitution should preserve validity in semantics.

Let us consider $\exists y \; P(x, y)$. Can x be substituted by y in this formula?

➥ no, as you change the meaning of the formula!

Definition (free substitution)

A term t is **freely substitutable** to x in φ if

- φ is an atomic formula
- $\varphi \equiv \neg\varphi_1$ and t is freely substituable to x in φ_1
- $\varphi \equiv \varphi_1$ conn φ_2 where conn $\in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$ and t is freely substituable to x in φ_1 and φ_2
- $\varphi \equiv Qy \; \varphi_1$ where $Q \in \{\forall, \exists\}$ and
  - x and y are the same variable
  - y is not a variable of t and t is freely substitutable for x in φ_1

Rules for natural deduction for FOL

As PL is a subset of FOL, all rules defined for PL are also valid for PL.

Rules have to be added for quantifiers (x is supposed to be free in A):

Definition (intr. and elim. rules for quantifiers)

$$\frac{A}{\forall x\ A}\ (I_\forall) \qquad \frac{\forall x\ A}{A[x/t]}\ (E_\forall)$$

$$\frac{A[x/t]}{\exists x\ A}\ (I_\exists) \qquad \frac{\exists x\ A \qquad \begin{array}{c}[A]\\ \vdots\\ B\end{array}}{B}\ (E_\exists)$$

Rules for natural deduction for FOL: sequent view

Definition (intr. and elim. rules for quantifiers)

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x\ A}\ (I_\forall) \qquad \frac{\Gamma \vdash \forall x\ A}{\Gamma \vdash A[x/t]}\ (E_\forall)$$

$$\frac{\Gamma \vdash A[x/t]}{\Gamma \vdash \exists x\ A}\ (I_\exists) \qquad \frac{\Gamma \vdash \exists x\ A}{\Gamma \vdash A[x/f(y_1, \ldots, f_n)]}\ (E_\exists)$$

where $x \notin FV(\Gamma)$ in $(I_\forall)$ and $FV(\exists x\ A) = \{y_i \mid i \in \{1, \ldots, n\}\}$.

Exercise

Prove the following FOL formulas in $\mathcal{NK}$:

$$(\forall x\ \varphi \land \psi) \rightarrow (\forall x\ \varphi \land \forall x\ \psi)$$
$$\exists x \forall y\ \varphi \rightarrow \forall y \exists x\ \varphi$$

Automatic proof of FOL wffs?

We can ask ourselves again if it is possible to build automatically proofs of the previous formulas.

Unfortunately, as **First-Order Logic is not decidable** (but ony semi-decidable), it is not possible to automatically prove all the possible theorems of FOL.

The previously presented theorem provers (E, Vampire, SPASS) can although be used to prove the previous formulas.

Use SPASS on our examples

```
begin_problem(fol_1).

list_of_descriptions.
  name({*(forall x Phi(x) /\ Psi(x)) ->
       (forall x Phi(x)) /\ (forall x Psi(x))*}).
  author({*Christophe Garion*}).
  status(satisfiable).
  description({*Prove (forall x Phi(x) /\ Psi(x)) ->
                      (forall x Phi(x)) /\ (forall x Psi(x))...*}).
end_of_list.

list_of_symbols.
  predicates[(Phi,1), (Psi,1)].
end_of_list.

list_of_formulae(conjectures).
  formula(implies(forall([X], and(Phi(X), Psi(X))),
                  and(forall([X], Phi(X)), forall([X], Psi(X))))).
end_of_list.

end_problem.
```

3 - The Floyd-Hoare logic

Outline of part 3 - The Floyd-Hoare logic

8. **Imperative programs**

9. **The Floyd-Hoare deductive system**

What kind of program do we want to "prove"?

Definition (imperative kernel)

The imperative kernel of a programming language is defined by the five following constructs: **declaration**, **assignment**, **sequence**, **conditional**, **loop**.

Theorem (Böhm-Jacopini,1966)

Algorithms combining subprograms using only the three following control structures can compute any computable function:

- *sequence (denoted by "`P;Q`")*
- *selection using boolean expression (denoted by "`if C then P else Q fi`")*
- *iteration while a boolean condition is true (denoted by "`while C do P od`")*

where `P` and `Q` are subprograms and `C` is a boolean expression.

➥ we will use only those three control structures in the following.

What kind of program do we want to "prove"?

Definition (assignment)

The assignment operator is denoted by `:=`.

But no declaration operator...

➥ types of variables will be "obvious"

By convention, we will use uppercase latin letters for variable names (X, Y, etc.).

Usual operators on integers like $+$, $*$ etc. are available to build **expressions** that can be used on the right side of `:=`.

N.B.

Expressions used on the right side of `:=` (rvalues) cannot have side effects!

Outline of part 3 - The Floyd-Hoare logic

8. Imperative programs

9. **The Floyd-Hoare deductive system**
   - Rules for partial correctness
   - Rule for total correctness

Hoare triple

Definition (Hoare triple)

A **Hoare triple** is denoted by $\{\varphi\}\ \ \mathsf{P}\ \ \{\psi\}$ where:

- φ is a first-order logic wff called the **precondition**
- P is a program as defined previously
- ψ is a first-order logic wff called the **postcondition**

Intuition

$\{\varphi\}\ \ \mathsf{P}\ \ \{\psi\}$ is true iff when starting from a state where φ is true, executing P leads to a state where ψ is true.

The terms used in φ and ψ generally speak about the state of the program.

What do we want to prove?

The Hoare triple of a program P is given as a **specification** of P.

Floyd-Hoare logic provides a formal system $\mathcal{FH}$ to reason on Hoare triples for each primitive programming construct.

So, proving that P is correct wrt. its specifications φ and ψ is proving that $\{\varphi\}$ P $\{\psi\}$ is a theorem in $\mathcal{FH}$.

Hoare, C. A. R. (1969).
"An axiomatic basis for computer programming".
In: **Communications of the ACM** 12.10,
Pp. 576–580.

8 **Imperative programs**

9 **The Floyd-Hoare deductive system**

- Rules for partial correctness
- Rule for total correctness

Rule for assignment

Definition (rule for assignment)

$$\frac{}{\{\varphi[X/E]\}\ \ \texttt{X := E}\ \ \{\varphi\}}\ (\texttt{:=})$$

Exercise

Find φ such that:

$$\frac{}{\{\varphi\}\ \ \texttt{X := X + 1}\ \ \{X = 4\}}\ (\texttt{:=})$$

$$\frac{}{\{\varphi\}\ \ \texttt{F := F * K}\ \ \{F = K!\}}\ (\texttt{:=})$$

$$\frac{}{\{\varphi\}\ \ \texttt{K := K + 1}\ \ \{F = (K-1)!\}}\ (\texttt{:=})$$

Assignment rule: why?

You may feel the previous axiom to be “backwards” from what your intuition says. But, if the axiom were

$$\dfrac{}{\{\varphi\}\ \ \texttt{X := E}\ \ \{\varphi[X/E]\}}\ (\texttt{:=})$$

what is the postcondition ψ in $\{X = 0\}$ `X := 1` $\{\psi\}$?

There is in fact a assignment axiom (from Floyd) which is the following:

$$\dfrac{}{\{\varphi\}\ \ \texttt{X := E}\ \ \{\exists v\ ((X = E[X/v]) \land \varphi[X/v])\}}\ (\texttt{:=})$$

where v is a new variable.

This rule is more complicated to use due to the existentially quantified variable, but it works!

Rule for sequence

Definition (rule for sequence)

$$\frac{\{\varphi\}\ \texttt{P}\ \{\gamma\} \qquad \{\gamma\}\ \texttt{Q}\ \{\psi\}}{\{\varphi\}\ \texttt{P;Q}\ \{\psi\}}\ \text{(Seq)}$$

Example:

$$\dfrac{\dfrac{}{\{(A+X\geq 0)[A/0]\}\ \texttt{A := 0}\ \{A+X\geq 0\}}\ (\texttt{:=}) \qquad \dfrac{}{\{(A+B\geq 0)[B/X]\}\ \texttt{B := X}\ \{A+B\geq 0\}}\ (\texttt{:=})}{\{X\geq 0\}\ \texttt{A := 0; B := X}\ \{A+B\geq 0\}}\ \text{(Seq)}$$

From now on, we will annote programs instead of writing the proof tree.

Is the sequence rule sufficient?

Is it possible to prove the following program using only the affectation and the sequence rules?

$\{X \geq 0\}$
```
A := 1
B := X;
```
$\{A + B \geq 0\}$

Consequence rule

Definition (consequence rule)

$$\frac{\varphi \to \varphi' \qquad \{\varphi'\}\ \mathsf{P}\ \{\psi'\} \qquad \psi' \to \psi}{\{\varphi\}\ \mathsf{P}\ \{\psi\}}\ \text{(Cons)}$$

Two derived rules:

Definition (strenghtening of precondition)

$$\frac{\varphi \to \varphi' \qquad \{\varphi'\}\ \mathsf{P}\ \{\psi\}}{\{\varphi\}\ \mathsf{P}\ \{\psi\}}\ \text{(Str)}$$

Definition (weakening of postcondition)

$$\frac{\{\varphi\}\ \mathsf{P}\ \{\psi'\} \qquad \psi' \to \psi}{\{\varphi\}\ \mathsf{P}\ \{\psi\}}\ \text{(Weak)}$$

Consequence rule

Definition (consequence rule)

$$\frac{\varphi \to \varphi' \qquad \{\varphi'\}\ \mathsf{P}\ \{\psi'\} \qquad \psi' \to \psi}{\{\varphi\}\ \mathsf{P}\ \{\psi\}}\ \text{(Cons)}$$

$\varphi \to \varphi'$ and $\psi' \to \psi$ are called **proof obligations**.

They are often proved by an **external theorem prover**.

They are the most "difficult parts" of the proof, as they may involve (complex) mathematics.

Definition (rule for conditional)

$$\frac{\{\varphi \land C\}\ \texttt{P}\ \{\psi\} \qquad \{\varphi \land \neg C\}\ \texttt{Q}\ \{\psi\}}{\{\varphi\}\ \textbf{if}\ \texttt{C}\ \textbf{then}\ \texttt{P}\ \textbf{else}\ \texttt{Q}\ \textbf{fi}\ \{\psi\}}\ \text{(Cond.)}$$

Exercise

Prove the following Hoare triple:

$$\{\top\}\ \textbf{if}\ \texttt{Y=0}\ \textbf{then}\ \texttt{X := Y}\ \textbf{else}\ \texttt{X := 0}\ \textbf{fi}\ \{X = 0\}$$

Definition (rule for iteration)

$$\frac{\{\varphi \land C\}\ \ \mathtt{P}\ \ \{\varphi\}}{\{\varphi\}\ \ \mathbf{while}\ \mathtt{C}\ \mathbf{do}\ \mathtt{P}\ \mathbf{od}\ \ \{\varphi \land \neg C\}}\ (\text{It.})$$

Definition (invariant)

In the previous rule, φ is called the **invariant**.
φ is a FOL formula that is true **before the first call** to P and is true **at each iteration** and **at the end of the loop**.

Iteration

Definition (rule for iteration)

$$\frac{\{\varphi \wedge C\} \ \ \texttt{P} \ \ \{\varphi\}}{\{\varphi\} \ \ \texttt{while C do P od} \ \ \{\varphi \wedge \neg C\}} \ (\mathrm{It.})$$

Exercise

Prove the following Hoare triple:

$$\{X \geq 0\} \ \ \texttt{while X<B do X := X+1 od} \ \ \{X \geq 0 \wedge \neg(X < B)\}$$

Partial vs. total correctness

What does happen if P does not terminate?

➥ we have to prove also that P terminates (loops...)

Definition (partial correctness)

A program P is **partially correct** wrt. to its specifications φ and ψ iff whenever starting from a state where φ is true and executing P, **if P terminates**, then the resulting state will satisfy ψ.

Definition (total correctness)

A program P is **totally correct** wrt. to its specifications φ and ψ iff P is partially correct wrt. to φ and ψ and P terminates.

Outline of part 3 - The Floyd-Hoare logic

8 **Imperative programs**

9 **The Floyd-Hoare deductive system**
- Rules for partial correctness
- Rule for total correctness

How to prove that the program terminate?

Intuition

In order to prove that a program terminate, find an expression e and a well-founded relation $\prec$ such that e decreases wrt $\prec$ during the execution.

In practise, e is often a function of the program variables returning a value in $\mathbb{N}$.

Only one rule has to be modified: the **iteration** rule.

Definition (rule for iteration)

$$\frac{\{\varphi \land C \land v = V\} \ \mathtt{P} \ \{\varphi \land v \prec V\} \qquad \prec \text{ is wf}}{\{\varphi\} \ \mathbf{while}\ \mathtt{C}\ \mathbf{do}\ \mathtt{P}\ \mathbf{od} \ \{\varphi \land \neg C\}} \ (\text{It.})$$

Definition (variant)

In the previous rule, v is called the **variant**.

Exercise: growing...

Prove the following program:

$\{N \geq 0\}$
```
K := 0
F := 1
while (K ≠ N) do
      K := K + 1;
      F := F * K
od
```
$\{F = N!\}$

Exercise: decreasing...

Prove the following program:

$\{N \geq 0\}$
```
K := N;
F := 1;
while (K ≠ 0) do
      F := F * K;
      K := K - 1
od
```
$\{F = N!\}$

Exercise: *divide ut imperes*

Prove the following program:

$\{X \geq 0 \wedge Y > 0\}$

```
Q := 0;
R := X;
while (Y ≤ R) do
      Q := Q + 1;
      R := R - Y
od
```

$\{X = Q \times Y + R \wedge 0 \leq Q \wedge 0 \leq R < Y\}$

Exercise: hello Euclid!

Prove the following program:

$\{A > 0 \wedge B > 0\}$

```
X := A;
Y := B;
while (X ≠ Y) do
      if (X > Y) then
            X := X - Y
      else
            Y := Y - X
      fi
od
```

$\{X = Y \wedge X > 0 \wedge X = gcd(A, B)\}$

4 - Automatic verification of imperative programs

Outline of part 4 - Aut. verification of imperative programs

Why automating verification?

Hoare logic is a formal system for proving imperative programs, but:

- proof obligations can be complicated and use complex theories
- for larger programs, you cannot do proofs "by hand"
- programming languages have (often) more constructs than those presented (e.g. pointers)

Conclusion

Program verification should be automated.

Automatizing verification: the principles

prog. language
+ annotations

↓

VC

↓

ATP

Automatizing verification: the principles

prog. language
+ annotations

↓

VC

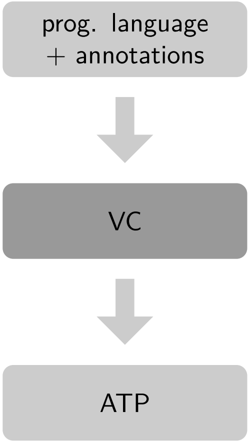
↓

ATP

Annotations

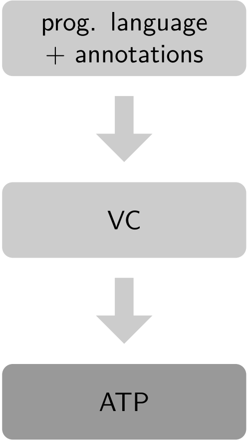
- extends prog. language syntax with pre/post-conditions, invariants etc.
- extension to first-order logic
- often represented by comments

Automatizing verification: the principles



Verification conditions

Annotated programs are translated by a verification conditions generator into **verification conditions (VC)**.
Verification conditions are **logical** properties that should hold for the program to be correct.
They are often used with several particular **domain theories**.



Automated theorem prover

An **automated theorem prover** is a software being able to prove that a given wff is a theorem.

Outline of part 4 - Aut. verification of imperative programs

Is it difficult to prove theorems?

Proving that a formula φ is a **theorem** is equivalent to proving that φ is **valid** (i.e. always true). This is also equivalent to proving that $\neg\varphi$ is not satisfiable...

Theorem (Cook, 1971)

*The SAT problem (i.e. verifying the satisfiability of a propositional formula) is **NP-complete**.*

➥ SAT is difficult to solve, but there are instances that can be solved efficiently

➥ **proving that a wff is a tautology/a theorem** is Co-NP-complete!

And for FOL?

Remember that proof obligations, invariants etc. use **first-order theories**, like arithmetics, linear inequalities etc.

Theorem

The decision problem for FOL, i.e. determining if a FOL wff is valid/a theorem or not, is ***not decidable****.*

- ➥ OK, so end of the story?
- ➥ No, some first-order theories are decidable:
  - Pressburger arithmetics
  - real numbers (!!!)
  - etc.
- ➥ We have seen that we can use theorem provers like SPASS to prove some wffs.

In the following we will use **SMT solvers**.

Definition (informal)

A **Satisfiability Modulo Theory** (**SMT**) problem is a decision problem based on SAT where the interpretation of some symbols is constrained by a background theory.

Barrett, Clark et al. (2009).
"Satisfiability Modulo Theories".
In:
Handbook of Satisfiability.
Ed. by Armin Biere et al.
Vol. 185.
Frontiers in Artificial Intelligence and Applications.
IOS Press.
Chap. 26, pp. 825–885.
ISBN: 978-1-58603-929-5.

Definition (informal)

A **Satisfiability Modulo Theory** (**SMT**) problem is a decision problem based on SAT where the interpretation of some symbols is constrained by a background theory.

For instance:

$$(3x + 2y \geq 3) \wedge (x - z < 2) \vee (z + y \leq x)$$

SMT solver used: Alt-Ergo

We will use Alt-Ergo, a SMT solver written in OCaml:

Conchon, Sylvain and Evelyne Contejean (2013).
Alt-Ergo, an OCaml SMT-solver for software verification.
http://alt-ergo.lri.fr/.

There are other interesting SMT solvers:

- Z3 (z3.codeplex.com)
- CVC4 (http://cvc4.cs.nyu.edu/web/)

SMT solver used: Alt-Ergo

Beware, Alt-Ergo cannot prove every theorem of FOL, for instance:

```
type E

logic phi : E -> prop
logic psi : E -> prop

logic phi2 : E, E -> prop

logic a: E

goal Th_1 : (forall x : E. phi(x) and psi(x)) ->
            (forall x : E. phi(x)) and (forall x : E. psi(x))

goal Th_2 : (exists x : E. forall y : E. phi2(x, y)) ->
            (forall y : E. exists x : E. phi2(x, y))

goal Th_3 : (forall y : E. exists x : E. phi2(x, y)) ->
            (exists x : E. forall y : E. phi2(x, y))

goal Th_4 : (forall y : E. phi2(a, y)) ->
            (exists x : E. forall y : E. phi2(x, y))
```

SMT solver used: Alt-Ergo

Alt-Ergo has been designed to be used for program verification, so it can solve problems we will have for proving programs:

```
goal arith_1 :
     forall x, y : int.
       2 * y - x <= 0 and -8 * y + x + 2 <= 0 and 2 * y + x - 3 <= 0
       -> false

goal arith_2 :
     forall x, y : int. x * (x + 1) = y -> x * x = y - x

goal arith_non_linear_1 :
     forall x, y : int.
       2 <= x <= 6 and -3 <= y < 0 ->
       -84 <= 3 * x + 2 * y + 4 * x * y + 2 * (x / y) <= 6

goal arith_non_linear_2 :
     forall x, y : int.
       2 <= x <= 6 and -3 <= y < 0 ->
       -64 <= 3 * x + 2 * y + 4 * x * y + 2 * (x / y) <= -8
```

Outline of part 4 - Aut. verification of imperative programs

Verification conditions: how to generate them?

Verification conditions are **purely logical** formulas **automatically** generated from an **annotated** program.

Dijdkstra's seminal work on **predicate transformer semantics** gives a complete strategy (either by **weakest preconditions** or strongest postconditions) to build theorems in FH logic.

Dijkstra, Edger W. (1975).
"Guarded commands, nondeterminacy and formal derivation of program".
In: **Communications of the ACM** 18.8,
Pp. 453–457.

Condition on program for VC generation

In order to automatize VC generation, **the program should contain enough assertions**.

Definition (properly annotated program)

A program is properly annotated if **there is an assertion**:

- before each subprogram `Ci` ($i > 1$) in a sequence `C1;C2;...;Cn` which **is not an assignment command**
- for each **loop invariant**

N.B.

Generation of loop invariants is generally undecidable.

Completely annotated program: example

$\{X \geq 0 \wedge Y > 0\}$

```
Q := 0;
R := X;
```

$\{R = X \wedge R \geq 0 \wedge Q = 0\}$

while (Y $\leq$ R) **do**

  $\{X = R + (Q \times Y) \wedge R \geq 0 \wedge Q \geq 0\} \leftarrow$ `loop invariant`

```
    Q := Q + 1;
    R := R - Y
```

od

$\{X = Q \times Y + R \wedge 0 \leq Q \wedge 0 \leq R < Y\}$

Definition (generation of VC for assignment)

The VC generated by $\{\varphi\}$ `X := E` $\{\psi\}$ is

$$\varphi \rightarrow \psi[X/E]$$

For instance, the VC generated by

$$\{X = 0\} \texttt{ X := X + 1 } \{X = 1\}$$

is

$$(X = 0) \rightarrow (X + 1) = 1$$

Definition (generation of VC for conditional)

The VC generated by

$$\{\varphi\}\ \texttt{if C then P else Q}\ \{\psi\}$$

are

1. the VC generated from $\{\varphi \wedge C\}$ `P` $\{\psi\}$
2. the VC generated from $\{\varphi \wedge \neg C\}$ `Q` $\{\psi\}$

Generating VC for sequences

Definition (generation of VC for sequence (case 1))

The VC generated by

$$\{\varphi\}\texttt{C1; C2; ...;Cn-1;}\{\varphi'\}\texttt{Cn}\{\psi\}$$

where **`Cn` is not an assignment** are

1. the VC generated from $\{\varphi\}$ `C1; C2; ...;Cn-1` $\{\varphi'\}$
2. the VC generated from $\{\varphi'\}$ `Cn` $\{\psi\}$

Definition (generation of VC for sequence (case 2))

The VC generated by

$$\{\varphi\}\ \texttt{C1; C2; ...;Cn-1; X := E}\ \{\psi\}$$

are the VC generated from

$$\{\varphi\}\ \texttt{C1; C2; ...;Cn-1}\ \{\psi[X/E]\}$$

Generating VC for iterations

Definition (generation of VC for iteration)

The VC generated by

$$\{\varphi\}\ \texttt{while C do P od}\ \{\psi\}$$

with invariant φ_i are

1. $\varphi \rightarrow \varphi_i$
2. $\varphi_i \wedge \neg C \rightarrow \psi$
3. the VC generated by

$$\{\varphi_i \wedge C\}\ \texttt{P}\ \{\varphi_i\}$$

1 $\{X \geq 0 \land Y > 0\}$
2 `Q := 0;`
3 `R := X;`
4 $\{R = X \land R \geq 0 \land Q = 0\}$
5 **while** `(Y` $\leq$ `R)` **do**
6 $\quad\{X = R + (Q \times Y) \land R \geq 0 \land Q \geq 0\}$
7 $\quad$`Q := Q + 1;`
8 $\quad$`R := R - Y`
9 **od**
10 $\{X = Q \times Y + R \land 0 \leq Q \land 0 \leq R < Y\}$

Applying VC generation for sequence on line 5 gives...

Generating VC: example

$$\{X \geq 0 \land Y > 0\}$$

```
Q := 0;
R := X;
```

$$\{R = X \land R \geq 0 \land Q = 0\}$$

$$\{R = X \land R \geq 0 \land Q = 0\}$$

```
while (Y ≤ R) do
```

$$\{X = R + (Q \times Y) \land R \geq 0 \land Q \geq 0\}$$

```
        Q := Q + 1;
        R := R - Y
od
```

$$\{X = Q \times Y + R \land 0 \leq Q \land 0 \leq R < Y\}$$

Generating VC: example

$\{X \geq 0 \land Y > 0\}$

```
Q := 0;
R := X;
```

$\{R = X \land R \geq 0 \land Q = 0\}$

Applying VC generation for sequence two times gives:

$$(X \geq 0 \land Y > 0) \rightarrow (X = X \land X \geq 0 \land 0 = 0)$$

Generating VC: example

1 $\{R = X \land R \geq 0 \land Q = 0\}$
2 **while** $(Y \leq R)$ **do**
3 $\quad\{X = R + (Q \times Y) \land R \geq 0 \land Q \geq 0\}$
4 `    Q := Q + 1;`
5 `    R := R - Y`
6 **od**
7 $\{X = Q \times Y + R \land 0 \leq Q \land 0 \leq R < Y\}$

Applying VC generation for iteration gives:

$$(R = X \land R \geq 0 \land Q = 0) \rightarrow (X = R + (Q \times Y) \land R \geq 0 \land Q \geq 0)$$

$$(X = R + (Q \times Y) \land R \geq 0 \land Q \geq 0 \land \neg(Y \leq R)) \rightarrow \\ (X = Q \times Y + R \land 0 \leq Q \land 0 \leq R < Y)$$

and VC generated from the inner part of the loop (lines 4 et 5, cf. next slide).

Generating VC: example

$$\{X = R + (Q \times Y) \wedge R \geq 0 \wedge Q \geq 0 \wedge Y \leq R\}$$

```
Q := Q + 1;
R := R - Y
```

$$\{X = R + (Q \times Y) \wedge R \geq 0 \wedge Q \geq 0\}$$

Applying VC generation for this sequences gives:

$$(X = R + (Q \times Y) \wedge R \geq 0 \wedge Q \geq 0 \wedge Y \leq R) \rightarrow$$
$$(X = (R - Y) + ((Q + 1) \times Y) \wedge R - Y \geq 0 \wedge Q + 1 \geq 0)$$

Using Alt-Ergo on generated VC...

```
goal VC_1 :
     forall X, Y : int.
       (X >= 0) and (Y > 0) -> (X = X) and (X >= 0) and (0 = 0)

goal VC_2 :
     forall X, Y, R, Q : int.
       (R = X) and (R >= 0) and (Q = 0) ->
       (X = R + (Q * Y)) and (R >= 0) and (Q >= 0)

goal VC_3 :
     forall X, Y, R, Q : int.
       (X = R + (Q * Y)) and (R >= 0) and (Q >= 0) and not(Y <= R) ->
       (X = Q * Y + R) and (0 <= R < Y) and (0 <= Q)

goal VC_4 :
     forall X, Y, R, Q : int.
       (X = R + (Q * Y)) and (R >= 0) and (Q >= 0) and (Y <= R) ->
       (X = (R-Y) + ((Q+1) * Y)) and ((R-Y) >= 0) and (Q+1 >= 0)
```

Outline of part 4 - Aut. verification of imperative programs

10. **Introduction on automated verification**

11. **Automated theorem proving**

12. **Generating verification conditions**

13. **Annotation language for C programs**
    - Toolchain: Frama-C + WP + Alt-Ergo
    - ACSL presentation

Outline of part 4 - Aut. verification of imperative programs

The global toolchain for C programs

C prog. lang.
+ ACSL

↓

WP plugin
of Frama-C

↓

Alt-Ergo

ACSL

The ANSI/ISO C Specification Language (ACSL) is a behavioral specification language for C, inspired of JML.

- function contract (pre/post-conditions)
- formal language
- granularity of assertions

Baudin, Patrick, Pascal Cuoq, et al. (2014).
ACSL: ANSI/ISO C Specification Language.
Version 1.8.
http://frama-c.com/download/acsl-implementation-Neon-

The global toolchain for C programs

C prog. lang.
+ ACSL

↓

WP plugin
of Frama-C

↓

Alt-Ergo

Frama-C

Frama-C is a suite of tools developed by CEA and INRIA dedicated to the analysis of C programs.
Analysis is made **statically**.

Baudin, Patrick, Richard Bonichon, et al. (2013).
Frama-C.
http://frama-c.com.

The global toolchain for C programs

C prog. lang.
+ ACSL

↓

WP plugin
of Frama-C

↓

Alt-Ergo

WP plugin

WP (for Weakest Precondition) is a VC generator for C programs annotated with ACSL.
It can be used with several provers (interactive or not).

Baudin, Patrick, François Bobot, et al. (2014).
WP Plug-in Manual.
http://www.frama-c.com/download/wp-manual-Neon-20140301.pdf.

Tools configuration at ISAE SI

At ISAE SI, tools configuration is done with the following commands (put them in your `.bashrc`):

shell

```
module load opam-softs
init-opam
```

Using Frama-C and WP

Suppose we have the following C program:

max.c

```c
int max(int i, int j) {
    if (i < j) {
        return j;
    } else {
        return i;
    }
}
```

Invoking WP on the program is done by the following command:

shell

```
frama-c -wp max.c
```

Using Frama-C and WP

Suppose we have the following C program:

max.c

```c
int max(int i, int j) {
    if (i < j) {
        return j;
    } else {
        return i;
    }
}
```

Invoking WP on the program with GUI is done by the following command:

shell

```
frama-c-gui -wp max.c
```

Try it on `max.c` (available in your repo).

An annotated version of max...

basic-annotated-max.c

```c
//@ ensures \result == (i < j ? j : i);
int max(int i, int j) {
    if (i < j) {
        return j;
    } else {
        return i;
    }
}
```

An (more) annotated version of max...

annotated-max.c

```c
/*@ requires \valid(i) && \valid(j);
  @ requires r == \null || \valid(r);
  @ assigns *r;
  @ behavior zero:
  @   assumes r == \null;
  @   assigns \nothing;
  @   ensures \result == -1;
  @ behavior normal:
  @   assumes \valid(r);
  @   assigns *r;
  @   ensures *r == (*i < *j ? *j : *j);
  @   ensures \result == 0;
  @*/
int max(int *r, int* i, int* j) {
    if (!r)
        return -1;

    if (*i < *j) {
        *r = *j;
        return 0;
    }

    *r = *i;
    return 0;
}
```

What is and can be verified with WP

Default behavior or user-defined behavior

Verification of postcondition, frame condition, loop invariants and assertions.

Safety verification

Verification of null-pointer dereferencing, buffer overflow, integer overflow...with special option.

Some useful options

- `-wp-help`: help ☺
- `-wp-split`: force splitting of conjunctions
- `-wp-fct f,g`: select only `f` and `g` functions
- `-wp-print`: pretty-print proof obligations
- `-wp-report`: generate a report
- `-wp-timeout n`: change provers timeout to `n` seconds
- `-wp-rte`: enable RTE checking

Try them on `annotated-max.c`.

Using Frama-C "for real"

If you want to completely develop an C application with Frama-C, use the following advises:

- verify that your C code is syntactically correct with `gcc` or `clang`
- beware of `/*@` comments blocks (you cannot separate `@` from `*` even with a space!)
- to specify contracts for functions or loops (cf. further), use blocks with `/*@` and `@*/` instead of several `//@` annotations

Outline of part 4 - Aut. verification of imperative programs

Where to write ACSL annotations?

Syntax (ACSL annotations)

ACSL annotations are written into special comments:

```
//@ ACSL annotation

/*@
  @ ACSL annotation
  @*/
```

Mathematical and logical operators

Every classical C operator (addition, multiplication, operators on bits) can be used in ACSL.
Some additional operations **may be** available: `\min`, `\max`, `\abs`, `\exp`, `\cos`, `\sqrt` etc.

Syntax (logical operators)

`&&`	and connector ($\wedge$)
`||`	or connector ($\vee$)
`==>`	implies connector ($\rightarrow$)
`<==>`	equivalence connector ($\leftrightarrow$)
`\forall`	universal quantifier ($\forall$)
`\exists`	existential quantifier ($\exists$)

Typing

The language of logic expressions is typed. Types are either **C types** or **logic types**.

Syntax (logic types)

Logic types are the following:

`integer`	**unbounded** integers
`real`	real numbers
`boolean`	booleans (different from integers...)

Specification writers can also introduce logic types.

Basic assertions

You can ask WP to check simple assertions inside your program by using the `assert` construct.

Syntax (assertion)

```
//@ assert logical_assertion;
```

Do not forget the “;” !

Write and verify some assertions in `basics.c`.

Specifying contracts for functions

Syntax (built-in constructs)

`\old(e)`	the value of `e` (predicate or exp.) in the pre-state of the function
`\result`	the returned value

Syntax (pre/post-conditions)

`//@ requires P;`	`P` is a precondition of the function
`//@ ensures Q;`	`Q` is a postcondition of the function

Loop annotations

Syntax (loop invariant and variant)

`//@ loop invariant P;`	`P` must hold before entering the loop and at each loop iteration
`//@ loop variant E;`	**expression** `E` is the variant of the loop (classical meaning)

Assign statements

`assigns` statement can (should!) be used to help provers. They precise which parameters/local variables are modified during the execution of the function/the loop.

Syntax (assign clause)

`//@ assigns a;`	parameter `a` is assigned during the execution of the function
`//@ loop assigns a;`	variable `a` is assigned during the execution of the loop

`\nothing` can be used with `assigns` clauses.

Complete specification of a function

A complete specification of a function consists of (**in this order**):

- (eventually) `requires` clauses
- (eventually) `assigns` clauses
- (eventually) `ensures` clauses

Notice that if you do not provide `assigns` clauses, WP will consider `assigns \everything` by default.

You can also use `behavior` to define different behaviours (more readable than big implications for instance).

Exercise: prove factorial function ✎

Exercise

Prove the factorial function defined in `fact.c`. Beware, it is not written as previously!

Predicates

Users can define their own theories, for instance predicates.

Syntax (predicate)

```
//@ predicate predicate_name(par.) = definition
```

Example:

```
//@ predicate is_positive(integer x) = x > 0;
```

Lemmas

Users can define **lemmas** and **axioms** in order to help ATP to establish validity of specifications.

Syntax (lemma and axioms)

```
//@ lemma lemma_name: wff
//@ axiom axiom_name: wff
```

Example:

```
/*@ axiom div_mul_identity:
  @     \forall real x, real y: y != 0.0 ==> y*(x/y) == x;
  @*/
```

N.B.

Lemmas have to be proved...

Exercise

Specify and prove GCD algorithm:

- first, define a theory for GCD in `gcd.c`
- then, specify and prove `gcd.c`

Working with pointers

If you want to work with pointers, you can use several build-in predicates.

Syntax (validity of a pointer)

`\valid(p)`	pointer `p` is valid both for reading and writing
`\valid(p+(n..m))`	memory regions from `p+n` to `p+m` are valid both for reading and writing

Syntax (memory regions overlapping)

`\separated(p, q)`	memory region pointed by `p` and `q` are separated
`\separated(p+(n..m), q+(i..j))`	

Divide ut imperes

✎

Exercise

Specify and prove the euclidean algorithm defined in `division_1.c` and `division_2.c`.

Built-in construct `\at`

Specification writers need sometimes to refer to a value of an expression at a particular state. The built-in construct `\at` can be used to refer to such a value.

Syntax (`\at`)

`\at(e,id)` refers to the value of expression `e` at label `id`.

`id` can be a regular C label or a label added with a ghost statement or one of the predefined labels:

`Pre`	pre-state of the function
`Here`	the state where the annotation appears
`Old`	pre-state of the function (visible in `ensures` clauses)
`Post`	post-state of the function (visible in `ensures` clauses)
`LoopEntry`	state just before entering the loop
`LoopCurrent`	state of the current iteration