

Introduction

Semantics and applications to verification

Xavier Rival

École Normale Supérieure

February 7, 2020

Program of this first lecture

Introduction to the course:

- ① a study of some **examples** of **software errors**
 - ▶ what are the causes ? what kind of properties do we want to verify ?
- ② a panel of the main **verification methods**
with a fundamental limitation: **undecidability**
 - ▶ many techniques allow to **compute semantic properties**
 - ▶ each comes with **advantages** and **drawbacks**
- ③ an introduction to the **theory of ordered sets**
(or, most likely, mostly a refresher...)
 - ▶ **order relations** are pervasive in **semantics** and **verification**
 - ▶ **fixpoints** of operators are also very common

Outline

Ariane 5 – Flight 501

Ariane 5:

- a satellite launcher
- replacement of **Ariane 4**, a lot more powerful
- **first flight**, June, 4th, 1996: **failure!**

Flight story:

- nominal take-off, normal flight for 36 seconds
- **T + 36.7 s** : **angle of attack change**, trajectory lost
- **T + 39 s** : **disintegration** of the launcher



Consequences:

- **loss of satellites** : more than \$ 370 000 000...
- **launcher** unusable for more than a year (delay !)

Full report available online:

<http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf>

Trajectory control system design overview

Sensors: gyroscopes, inertial reference systems...

Calculators (hardware + software) :

- **“Inertial Reference System” (SRI) :**
integrates data about the trajectory (read on sensors)
- **“On Board Computer” (OBC) :**
computes the engine actuations that are required to follow the pre-determined theoretical trajectory

Actuators: **engines** of the launcher **follow orders** from the **OBC**

Redundant systems (failure tolerant system):

- **keep running** even in the presence of one or several system failures
- **traditional solution in embedded systems: duplication** of systems
aircraft flight system: 2 or 3 hydraulic circuits
launcher like Ariane 5 : 2 SRI units (SRI 1 and SRI 2)
- there is also a **control monitor**

The root cause: an unhandled arithmetic error

Processor registers

Each **register** has a size of 16, 32, 64 bits:

- **64-bits floating point**: values in range $[-3.6 \cdot 10^{308}, 3.6 \cdot 10^{308}]$
- **16-bits signed integers**: values in range $[-32768, 32767]$
- upon **copy of data**: conversions are performed such as **rounding**
- when the values are **too large**:
 - ▶ **interruption**: run error handling code if any, otherwise crash
 - ▶ or **unexpected behavior**: modulo arithmetic or other

Ariane 5:

- the SRI hardware runs in **interruption mode**
- it has **no error handling code** for arithmetic interruptions
- an **unhandled arithmetic conversion overflow crashes the SRI**

From the root cause to the failure

A **not so trivial** sequence of events:

- 1 a **conversion from 64-bits float to 16-bits signed int** is performed and **causes an overflow**
- 2 an **interruption** is raised
- 3 due to the lack of error handling code, the SRI **crashes**
- 4 the crash causes an **error return** (negative integer value) value be **sent to the OBC** (On-Board Computer)
- 5 the OBC interprets this illegal value as **flight data**
- 6 this causes the computation of an **absurd trajectory**
- 7 hence the **loss of control** of the launcher

Let us discuss **a few specific points**

A crash due to an unaddressed software case

Several solutions would have prevented this mishappening:

❶ **Deactivate interruptions on overflows:**

- ▶ then, an overflow may happen, and produce wrong values in the SRI
- ▶ but, these wrong values will not cause the computation to stop!
and most likely, the flight will not be impacted too much

❷ **Fix the SRI code**, so that **no overflow can happen**:

- ▶ all conversions must be **guarded against overflows**:

```
double x = /* ... */;  
short i = /* ... */;  
if( -32768. <= x && x <= 32767. )  
    i = (short) x;  
else  
    i = /* default value */;
```

- ▶ this may be costly (many tests), but redundant tests can be removed

❸ **Handle** conversion errors (not trivial):

- ▶ the handling code should **identify the problem** and **fix it** at run-time
- ▶ the OBC should **identify illegal input values**

A crash due to a useless task

Piece of code that generated the error:

- part of a gyroscope re-calibration process
- very useful to quickly restart the launch process after a short delay
- can only be done **before lift-off**...
- ... **but not after!**

Re-calibration task shut down:

- normally planned **50 seconds** after lift-off...
- no chance of a need for such a re-calibration after $T_0 + 3$ seconds
- the crash occurred at **36 seconds**

A crash due to legacy software

Software history:

- **already used in Ariane 4** (previous launcher, before Ariane 5)
- the software was tested and ran in real conditions many times yet never failed...
- but Ariane 4 was a **much less powerful** launcher

Software optimization:

- many conversions were **initially protected by a safety guard**
- but these tests were considered **expensive**
(a test and a branching take processor cycles, interact with the pipeline...)
- thus, conversions were ultimately **removed** for the sake of performance

Yet, Ariane 5 violates the assumptions that were valid with Ariane 4

- **higher values** of *horizontal bias* were generated
- those **were never seen** in Ariane 4, hence the failure

A crash not prevented by redundant systems

Principle of redundant systems: survive the failure of a component by the use of redundant systems

System redundancy in Ariane 5:

- one OBC unit
- **two SRI units**... yet **running the same software**

Obviously, physical redundancy does not address software issues

Other implementation of system redundancy (e.g., Airbus FBW):

- two independent set of controls
- three computing units per set of controls
- each computing unit comprises **two computers** with **distinct softwares** (design and implementation is also performed in **distinct teams**)

Ariane 501, a summary of the issues

A **long series of design errors**, all related to a lack of understanding of what the software does:

- ① **Non-guarded** conversion raising an **interruption** due to **overflow**
- ② **Removal of pre-existing guards**, too high confidence in the software
- ③ **Non revised assumptions** on the inputs when moving from Ariane 4 to Ariane 5
- ④ Redundant systems **running the same software**
- ⑤ Useless task **not shutdown** at the right time

Current status: such issues can be found by static analysis tools

Outline

High-speed runway overshoot at landing

Landing at Warsaw airport, Lufthansa A320:

- **bad weather conditions:** rain, high side wind
- **wet runway**
- landing (300 km/h) followed by **aqua-planing**, and **delayed braking**
- **runway overrun** at 132 km/h
- **impact** against a hillside at about 100 km/h

Consequences:

- **2 fatalities, 56 injured** (among 70 passengers + crew)
- **aircraft completely destroyed** (impact + fire)

Full report available online:

<http://www.rvs.uni-bielefeld.de/publications/Incidents/DOCS/ComAndRep/Warsaw/warsaw-report.html>

Causes of the accident

① Root cause:

- ▶ **bad weather conditions** not well assessed by the crew
- ▶ side wind **exceeding** aircraft certification specification
- ▶ **wrong action from the crew:**
 - a “Go Around” (missed landing, acceleration + climb) should have been done

② Contributing factor: **delayed action of the brake system**

time (seconds)	distance (meters) from runway threshold	events
T_0	770 m	main landing gear landed
$T_0 + 3 \text{ s}$	1030 m	nose landing gear landed brake command activated
$T_0 + 12 \text{ s}$	1680 m	spoilers activated
$T_0 + 14 \text{ s}$	1800 m	thrust reversers activated
$T_0 + 31 \text{ s}$	2700 m	end of runway

Protection of aircraft brake systems

- **Braking systems inhibition: Prevent in-flight activation !**

- ▶ **spoilers:** increase in aerodynamic load (drag)
- ▶ **thrust reversers:** could destroy the plane if activated in-flight !
(ex : crash of a B 767-300 ER Lauda Air, 1991, 223 fatalities; thrust reversers in-flight activation, electronic circuit issue)

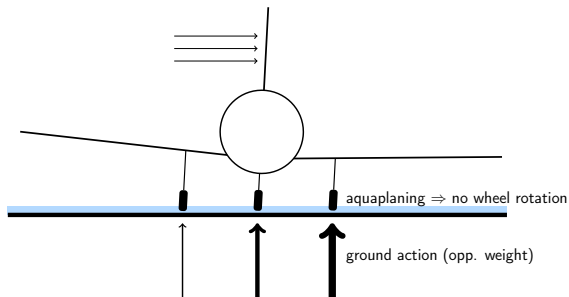
- **Braking software specification:**

DO NOT activate spoilers and thrust reverse **unless the following condition is met:**

- ▶ thrust lever should be set to **minimum** by the flight crew
 - ▶ **AND** either of the following conditions:
 - ★ **weight** on the main gear should be at least **12 T**
i.e., **6 T** for each side
 - ★ **OR wheels should be spinning**, with a speed of at least 130 km/h
- [Minimum Thrust] **AND** ([Weight] **OR** [Wheels spinning])

Understanding the braking delay

- **Landing configuration:**



- **Braking systems: inhibited**

- ▶ **thrust command** properly set to minimum
- ▶ **no weight** on the left landing gear due to **the wind**
- ▶ **no speed on wheels** due to **aquaplaning**

[Minimum Thrust] AND ([Weight] OR [Wheels spinning])

Flight 2904, a summary of the issues

Main factor is human (landing in weather conditions the airplane is not certified for), but the specification of the software is a contributing factor:

- **Old condition** that failed to be satisfied:

$$(P_{\text{left}} > 6T) \textbf{ AND } (P_{\text{right}} > 6T)$$

- **Fixed condition** (used in the new version of the software):

$$(P_{\text{left}} + P_{\text{right}}) > 12T$$

- The fix can be understood **only with knowledge of the environment**
 - ▶ conditions which the airplane will be used in
 - ▶ behavior of the sensors

Outline

The anti-missile “Patriot” system

- **Purpose:** destroy foe missiles before they reach their target
- **Use in wars:**
 - ▶ **first Gulf war** (1991)
protection of towns and military facilities in Israël and Saudi Arabia (against “Scud” missiles launched by Irak)
 - ▶ **success rate:**
 - ★ around 50 % of the “Scud” missiles are successfully destroyed
 - ★ almost all launched Patriot missiles destroy their target
 - ★ **failures** are due to **failure to launch a Patriot missile**
- **Constraints on the system:**
 - ▶ **hit very quickly moving targets:**
“Scud” missiles fly at around 1700 m/s ; travel about 1000 km in 10 minutes
 - ▶ **not to destroy a friendly target** (it happened at least twice!)
 - ▶ very high cost: about **\$1 000 000** per launch

System components

Detection / trajectory identification:

- **detection** using radar systems
- **trajectory confirmation** (to make sure a foe missile is tracked):
 - ① **trajectory identification** using a sequence of points at various instants
 - ② **trajectory confirmation**
computation of a predictive window (from position and speed vector)
+ confirmation of the predicted trajectory
 - ③ **identification of the target** (friend / foe)

Guidance system:

- **interception trajectory** computation
- **launch** of a Missile, and control until it hits its target
high precision required (both missiles travel at more than 1500 m/s)

Very short process: about ten minutes

Dahran failure (1991)

- ① **Launch of a “Scud” missile**
- ② **Detection** by the radars of the Patriot system
but failure to confirm the trajectory:
 - ▶ **imprecision** in the computation of the **clock** of the detection system
 - ▶ computation of **a wrong confirmation window**
 - ▶ the “Scud” cannot be found **in the predicted window**
failure to confirm the trajectory
 - ▶ the detection computer concludes it is a **false alert**
- ③ **The “Scud” missile hits its target:**
28 fatalities and around 100 people injured

Fixed precision arithmetic

- **Fixed precision numbers** are of the form $\epsilon N 2^{-p}$ where:
 - ▶ p is fixed
 - ▶ $\epsilon \in \{-1, 1\}$ is the **sign**
 - ▶ $N \in [-2^n, 2^n - 1]_{\mathbb{Z}}$ is an **integer** ($n > p$)
- In **32 bits fixed precision**, with one sign bit, $n = 31$;
thus we may let $p = 20$
- **A few examples:**

decimal value	sign	truncated value	fractional portion
2	0	00000000010	00000000000000000000
-5	1	00000000101	00000000000000000000
0.5	0	00000000000	10000000000000000000
-9.125	1	00000001001	00100000000000000000

- **Range of values that can be represented:**

$$\pm 2^{12}(1 - 2^{-32})$$

Rounding errors in fixed precision computations

- Not all real numbers in the right range can be represented
rounding is unavoidable
may happen both for **basic operations** and for **program constants**...
- **Example:** fraction $1/10$
 - ▶ $1/10$ **cannot be represented exactly** in fixed precision arithmetic
 - ▶ let us decompose $1/10$ as a sum of terms of the form $\frac{1}{2^i}$:

$$\begin{aligned}\frac{1}{10} &= \frac{1}{2} \cdot \frac{1}{5} \\ \frac{1}{5} &= \frac{1}{8} + \frac{1}{16} + \frac{1}{16} \cdot \frac{1}{5} = \frac{1}{8} + \frac{1}{16} + \frac{1}{16} \cdot \left(\frac{1}{8} + \frac{1}{16} + \frac{1}{16} \cdot \frac{1}{5} \right) = \dots\end{aligned}$$

- ▶ **infinite binary representation:** $0.00011001100110011001100\dots$
 - ▶ if $p = 24$:
representation: "0.000110011001100110011001"
rounding error is $9.5 \cdot 10^{-8}$
- **Floating precision numbers** (more commonly used today) have the same limitation

The root cause: a clock drift

Trajectory confirmation algorithm (summary):

- **hardware clock** T_d ticks every **tenth of a second**
- time T_c is computed **in seconds**: $T_c = \frac{1}{10} \times T_d$
- in **binary**: $T_c = 0.000110011001100110011001b \times_b T_d$!
- **relative error is 10^{-6}**
- after the computer has been running for **100 h** :
 - ▶ the absolute error is **0.34 s**
 - ▶ as a “Scud” travels at 1700 m/s : the predicted window is about **580 m** from where it should be
this explains **the trajectory confirmation failure!**

Remarks:

- the issue was **discovered** by israeli users, who noticed the clock drift
their solution: **frequently restart the control computer...** (daily)
- this was not done in Dahran... the system had been running for 4 days

Patriot missile failure, a summary of the issues

Precision issues in the fixed precision arithmetic:

- A scalar **constant** used in the code was **invalid** i.e., bound to be rounded to an approximate value, incurring a significant approximation the designers were unaware of
- There was **no adequate study of the precision** achieved by the system, although precision is clearly critical here !

Current status: such issues can be found by static analysis tools

Outline

Common issues causing software problems

The examples given so far **are not isolated cases**

See for instance:

`www.cs.tau.ac.il/~nachumd/horror.html`

(not up-to-date)

Typical reasons:

- **Improper specification** or understanding of the environment, conditions of execution...
- **Incorrect implementation of a specification**
e.g., the code should be free of runtime errors
e.g., the software should produce a result that meets some property
- **Incorrect understanding of the execution model**
e.g., generation of too imprecise results

New challenges to ensure embedded systems do not fail

Complex software architecture: e.g. **parallel softwares**

- single processor **multi-threaded**, **distributed** (several computers)
- more and more common: multi-core architectures
- **very hard to reason about**
 - ▶ other kinds of issues: **dead-locks**, **races**...
 - ▶ very complex execution model: **interleavings**, **memory models**

Complex properties to ensure: e.g., **security**

- the system should resist even in the presence of an **attacker** (agent with malicious intentions)
- attackers may try to **access sensitive data**, to **corrupt** critical data...
- security properties are often even **hard to express**

Techniques to ensure software safety

Software development techniques:

- **software engineering**, with a focus on specification, and software quality (may be more or less formal...)
- **programming rules** for specific areas (e.g., DO 178 c in avionics)
- usually do not guarantee any strong property, but make softwares “cleaner”

Formal methods:

- should have **sound mathematical foundations**
- should allow to **guarantee** softwares meet some complex properties
- should be **trustable** (is a paper proof ok ???)
- **increasingly used in real life applications**, but still a lot of open problems

What is to be verified ?

What do the C programs below do ?

What do these C programs do ?

P0.c

```
int x = 0
int f0( int y ){
    return x * y;
}
int f1( int y ){
    x = y;
    return 0;
}
void main( ){
    int z = f0( 10 ) +
           f1( 100 );
}
```

P1.c

```
void main( ){
    int i;
    int t[100] = { 0, 1, 2,
                  ..., 99 };
    while( i < 100 ){
        t[i]++;
        i++;
    }
}
```

P2.c

```
void main( ){
    float f = 0.;
    for( int i = 0;
         i < 1000000;
         i++ )
        f = f + 0.1;
}
```

Semantic subtleties...

P0.c

```
int x = 0
int f0( int y ){
    return x * y;
}
int f1( int y ){
    x = y;
    return 0;
}
void main( ){
    int z = f0( 10 ) + f1
        ( 100 );
}
```

Execution order:

- not specified in C
- specified in Java
- if left to right, $z = 0$
- if right to left, $z = 1000$

Semantic subtleties...

P1.c

```
void main( ){
    int i;
    int t[100] = { 0, 1, 2,
                  ..., 99 };
    while( i < 100 ){
        t[i]++;
        i++;
    }
}
```

P2.c

```
void main( ){
    float f = 0.;
    for( int i = 0;
        i < 1000000;
        i++ )
        f = f + 0.1;
}
```

Initialization:

- **runtime error in Java**
- **read of a random value in C** (the value that was stored before)

Floating point semantics:

- **0.1 is not representable exactly**; what is it rounded to by the compiler ?
- **rounding errors**; what is the rounding mode at runtime ?

The two main parts of this course

① Semantics

- ▶ allow to **describe precisely the behavior of programs**
should account for execution order, initialization, scope...
- ▶ allow to **express the properties to verify**
several important families of properties: safety, liveness, security...
- ▶ also important to **transform** and **compile** programs

② Verification

- ▶ aim at **proving** semantic properties of programs
- ▶ a very strong limitation: **undecidability**
- ▶ **several approaches**, that make various compromises around undecidability

Outline

The termination problem

Termination

Program P terminates on input X if and only if any execution of P , with input X eventually reaches a final state

- **Final state:** final point in the program (i.e., not error)
- **We may want to ensure termination:**
 - ▶ processing of a task, such as, e.g., printing a document
 - ▶ computation of a mathematical function
- **We may want to ensure *non-termination*:**
 - ▶ operating system
 - ▶ device drivers

The termination problem

Can we find a program P_t that **takes as argument a program P and data X and that returns “TRUE” if P terminates on X and “FALSE” otherwise ?**

The termination problem is not computable

- **Proof by reductio ad absurdum**, using a *diagonal argument*

We assume **there exists a program P_a such that:**

- ▶ P_a always terminates
- ▶ $P_a(P, X) = 1$ **if P terminates** on input X
- ▶ $P_a(P, X) = 0$ **if P does not terminate** on input X

- We consider the following program:

```
void P0( P ){  
    if( Pa( P, P ) == 1 ){  
        while( 1 ){  
            // loop forever  
        }  
    } else {  
        return; // do nothing  
    }  
}
```

- **What is the return value of $P_a(P_0, P_0)$?**
i.e., **does P_0 terminate on input P_0 ?**

The termination problem is not computable

- **What is the return value of $P_a(P_0, P_0)$?**

We know P_a always terminates and returns either 0 or 1 (assumption). Therefore, we need to consider only two cases:

- ▶ if $P_a(P_0, P_0)$ returns 1, then $P_0(P_0)$ **loops forever**, thus $P_a(P_0, P_0)$ should return 0, so we have reached a **contradiction**
- ▶ if $P_a(P_0, P_0)$ returns 0, then $P_0(P_0)$ **terminates**, thus $P_a(P_0, P_0)$ should return 1, so we have reached a **contradiction**

- In both cases, we **reach a contradiction**

- Therefore we conclude **no such a P_a exists**

The termination problem is not decidable

There exists no program P_t that always terminates and always recognizes whether a program P terminates on input X

Absence of runtime errors

- Can we find a program P_c that takes a program P and input X as arguments, always terminates and returns
 - ▶ 1 if and only if P runs safely on input X , i.e., without a runtime error
 - ▶ 0 if P crashes on input X
- Answer: **No**, the same diagonal argument applies
if $P_c(P, X)$ decides whether P will run safely on X , consider

```
void P1( P ){  
    if( P_c( P, P ) == 1 ){  
        0 / 0; // deliberately crash  
            (unsafe)  
    } else {  
        return; // do nothing  
    }  
}
```

Non-computability result

The absence of runtime errors is not computable

Rice theorem

- **Semantic specification:** set of *correct* program executions
 - **“Trivial” semantic specifications:**
 - ▶ empty set
 - ▶ set of all possible executions
- ⇒ intuitively, the non interesting verification problems...

Rice theorem (1953)

**Considering a Turing complete language,
any non trivial semantic specification is not computable**

- **Intuition:** there is no algorithm to decide non trivial specifications, starting with only the program code
- Therefore **all interesting properties are not computable** :
 - ▶ **termination**,
 - ▶ **absence of runtime errors**,
 - ▶ **absence of arithmetic errors**, etc...

Outline

Towards partial solutions

The initial verification problem is **not computable**

Solution: solve a weaker problem

Several compromises can be made:

- **simulation / testing:** observe only **finitely many finite executions** infinite system, but only finite exploration (no proof beyond that)
- **assisted theorem proving:** we **give up on automation** (no proof inference algorithm in general)
- **model checking:** we consider only **finite systems** (with finitely many states)
- **bug-finding:** search for “patterns” indicating “likely errors” (may miss real program errors, and report non existing issues)
- **static analysis with abstraction: attempt at automatic correctness proofs** (yet, may fail to verify some correct programs)

Safety verification method characteristics

Safety verification problem

- **Semantics** $\llbracket P \rrbracket$ of program P : set of behaviors of P (e.g., states)
- **Property to verify** \mathcal{S} : set of admissible behaviors (e.g., safe states)

Goal: establish $\llbracket P \rrbracket \subseteq \mathcal{S}$

- **Automation:** **existence of an algorithm**
- **Scalability:** should allow to **handle large softwares**
- **Soundness:** identify **any wrong program**
- **Completeness:** accept **all correct programs**
- **Apply to program source code**, i.e., not require a **modelling phase**

1. Testing by simulation

Principle

Run the program on **finitely many finite inputs**

- maximize **coverage**
- **inspect erroneous traces** to fix bugs
- **Very widely used:**
 - ▶ **unit testing**: each function is tested separately
 - ▶ **integration testing**: with all surrounding systems, hardware
e.g., **iron bird** in avionics
- **Automated**
- **Complete**: will never raise a false alarm
- **Unsound** unless exhaustive: **may miss program defects**
- **Costly**: needs to be re-done when software gets updated

2. Machine assisted proof

Principle

Have a **machine checked** proof, that is partly **human written**

- **tactics / solvers** may help in the inference
- the **hardest invariants** have to be user-supplied

- **Applications**

- ▶ software industry (rare): Line 14 in Paris Subway
- ▶ hardware: ACL 2
- ▶ academia: CompCert compiler, SEL4 verified micro-kernel
- ▶ also for math: four colour theorem, Feith-Thomson theorem

- **Not fully automated**

often turns out **costly** as complex proof arguments have to be found

- **Sound** and quasi-**complete** (in practice fine...)

3. Model-Checking

Principle

Consider **finite systems** only, using algorithms for

- **exhaustive exploration**,
- **symmetry reduction**...

- **Applications:**

- ▶ **hardware** verification
- ▶ **driver protocols** verification (Microsoft)

- Applies on **a model**: a model extraction phase is needed

- ▶ for infinite systems, this is **necessarily approximate**
- ▶ not always automated

- **Automated, sound, complete with respect to the model**

4. “Bug finding”

Principle

Identify “**likely**” **issues**, i.e., patterns known to often indicate an error

- use **bounded symbolic execution**, **model exploration**...
- **rank "defect" reports** using heuristics

- Intuition: **model checking made unsound**
- **Example**: Coverity
- **Automated**
- **Not complete**: may report false alarms
- **Not sound**: may accept false programs
thus **inadequate** for safety-critical systems

5. Static analysis with abstraction (1/4)

Principle

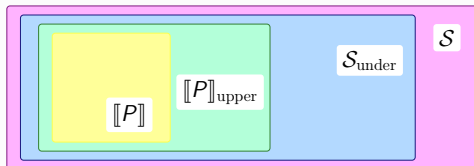
Use some approximation, but **always in a conservative manner**

- **Under-approximation** of the property to verify: $\mathcal{S}_{\text{under}} \subseteq \mathcal{S}$
- **Over-approximation** of the semantics: $\llbracket P \rrbracket \subseteq \llbracket P \rrbracket_{\text{upper}}$
- We let an automatic static analyzer attempt to prove that:

$$\llbracket P \rrbracket_{\text{upper}} \subseteq \mathcal{S}_{\text{under}}$$

If it succeeds, $\llbracket P \rrbracket \subseteq \mathcal{S}$

- In practice, the static analyzer **computes** $\llbracket P \rrbracket_{\text{upper}}, \mathcal{S}_{\text{under}}$



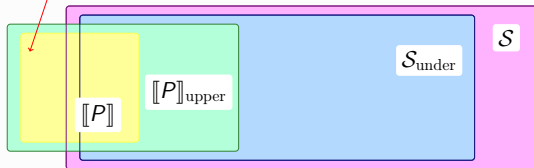
5. Static analysis with abstraction (2/4)

Soundness

The abstraction will catch **any incorrect program**

- If $\llbracket P \rrbracket \not\subseteq \mathcal{S}$, then $\llbracket P \rrbracket_{\text{upper}} \not\subseteq \mathcal{S}_{\text{under}}$
 since $\begin{cases} \mathcal{S}_{\text{under}} \subseteq \mathcal{S} \\ \llbracket P \rrbracket \subseteq \llbracket P \rrbracket_{\text{upper}} \end{cases}$

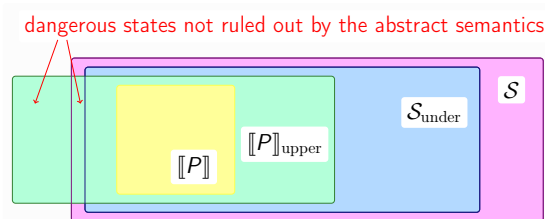
error found



5. Static analysis with abstraction (3/4)

Incompleteness

The abstraction may fail to certify **some correct programs**



Case of a false alarm:

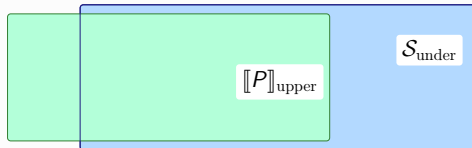
- program P is **correct**
- but the static analysis **fails**

5. Static analysis with abstraction (4/4)

Incompleteness

The abstraction may fail to certify **some correct programs**

- In the following case, the analysis cannot conclude anything



- One goal of the static analyzer designer is to avoid such cases

Static analysis using abstraction

- **Automatic**: $\llbracket P \rrbracket_{upper}$, S_{under} computed automatically
- **Sound**: reports any incorrect program
- **Incomplete**: may reject correct programs

A summary of common verification techniques

	Automatic	Sound	Complete	Source level	Scalable
Simulation	Yes	No ¹	Yes	Yes	sometimes ²
Assisted proving	No	Yes	Almost	Partially	sometimes ³
Model-checking	Yes	Yes	Partially ⁴	No	sometimes
Bug-finding	Yes	No	No	Yes	sometimes
Static analysis	Yes	Yes	No	Yes	sometimes

- Obviously, no approach checks all characteristics
- Scalability is a challenge for all

¹unless full testing is doable

²full testing usually not possible except for small programs with finite state space

³quickly requires huge manpower

⁴only with respect to the finite models... but not with respect to infinite semantics

Outline

Order relations

Very useful in semantics and verification:

- **logical ordering**, expresses **implication** of logical facts
- **computational ordering**, useful to establish well-foundedness of fixpoint definitions, and for termination

Definition: partially ordered set (poset)

Let a set \mathcal{S} and a binary relation $\sqsubseteq \subseteq \mathcal{S} \times \mathcal{S}$ over \mathcal{S} . Then, \sqsubseteq is an **order relation** (and $(\mathcal{S}, \sqsubseteq)$ is called a **poset**) if and only if it is

- **reflexive**: $\forall x \in \mathcal{S}, x \sqsubseteq x$
 - **transitive**: $\forall x, y, z \in \mathcal{S}, x \sqsubseteq y \wedge y \sqsubseteq z \implies x \sqsubseteq z$
 - **antisymmetric**: $\forall x, y \in \mathcal{S}, x \sqsubseteq y \wedge y \sqsubseteq x \implies x = y$
-
- notation: $x \sqsubset y ::= (x \sqsubseteq y \wedge x \neq y)$

Graphical representation

We often use **Hasse diagrams** to represent posets:

Extensive definition:

- $\mathcal{S} = \{x_0, x_1, x_2, x_3, x_4\}$
- \sqsubseteq defined by:

$$x_0 \sqsubseteq x_1$$

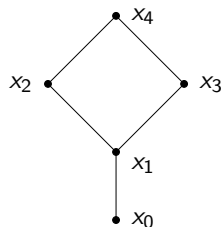
$$x_1 \sqsubseteq x_2$$

$$x_1 \sqsubseteq x_3$$

$$x_2 \sqsubseteq x_4$$

$$x_3 \sqsubseteq x_4$$

Diagram:



- By reflexivity, we have, e.g., $x_1 \sqsubseteq x_1$
- By transitivity, we have, e.g., $x_1 \sqsubseteq x_4$

Order relations are very useful in semantics...

Example: semantics of automata

In the following, we **illustrate order relations** and their **usefulness in semantics** using **word automata**.

We consider the classical notion of **finite word automata** and let

- L be a finite set of **letters**
- Q be a finite set of **states**
- $q_i, q_f \in Q$ denote the **initial** state and **final** state
- $\rightarrow \subseteq Q \times L \times Q$ be a **transition relation**

Semantics of an automaton

The set of words recognized by $\mathcal{A} = (Q, q_i, q_f, \rightarrow)$ is defined by:

$$\mathcal{L}[\mathcal{A}] = \{a_0 a_1 \dots a_n \mid \exists q_0 \dots q_{n-1} \in Q, q_i \xrightarrow{a_0} q_0 \xrightarrow{a_1} q_1 \dots q_{n-1} \xrightarrow{a_n} q_f\}$$

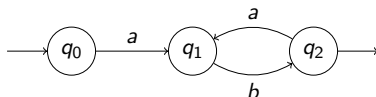
Example: automata and semantic properties

A simple automaton:

$$L = \{a, b\} \quad Q = \{q_0, q_1, q_2\}$$

$$q_i = q_0 \quad q_f = q_2$$

$$q_0 \xrightarrow{a} q_1 \quad q_1 \xrightarrow{b} q_2 \quad q_2 \xrightarrow{a} q_1$$



A few semantic properties:

- \mathcal{P}_0 : no recognized word contains two consecutive b

$$\mathcal{L}[\mathcal{A}] \subseteq L^* \setminus L^*bbL^*$$

- \mathcal{P}_1 : all recognized words contain at least one occurrence of a

$$\mathcal{L}[\mathcal{A}] \subseteq L^*aL^*$$

- we could also consider under-approximation properties (of the form $\mathcal{P}_2 \subseteq \mathcal{L}[\mathcal{A}]$), but do not in this lecture

Total ordering

Definition: total order relation

Order relation \sqsubseteq over \mathcal{S} is a **total** order if and only if

$$\forall x, y \in \mathcal{S}, x \sqsubseteq y \vee y \sqsubseteq x$$

Examples:

- **real numbers:**

(\mathbb{R}, \leq) is a total ordering

- **powerset:**

if set \mathcal{S} has at least two distinct elements x, y then its powerset

$(\mathcal{P}(\mathcal{S}), \subseteq)$ is **not** a total order

indeed $\{x\}, \{y\}$ cannot be compared

Most of the order relations we will use are **not be total**

indeed: very often, powerset or similar

Minimum and maximum elements

Definition: extremal elements

Let $(\mathcal{S}, \sqsubseteq)$ be a poset and $\mathcal{S}' \subseteq \mathcal{S}$. Then x is

- **minimum element** of \mathcal{S}' if and only if $x \in \mathcal{S}' \wedge \forall y \in \mathcal{S}', x \sqsubseteq y$
- **maximum element** of \mathcal{S}' if and only if $x \in \mathcal{S}' \wedge \forall y \in \mathcal{S}', y \sqsubseteq x$

- maximum and minimum elements **may not exist**
example: $\{\{x\}, \{y\}\}$ in the powerset, where $x \neq y$
- **infimum** \perp (“**bottom**”): minimum element of \mathcal{S}
- **supremum** \top (“**top**”): maximum element of \mathcal{S}

Exercise:

what are the logical interpretations of infimum / supremum elements ?

Upper bounds and least upper bound

Definition: bounds

Given poset $(\mathcal{S}, \sqsubseteq)$ and $\mathcal{S}' \subseteq \mathcal{S}$, then $x \in \mathcal{S}$ is

- an **upper bound** of \mathcal{S}' if

$$\forall y \in \mathcal{S}', y \sqsubseteq x$$

- the **least upper bound** (lub) of \mathcal{S}' (noted $\sqcup \mathcal{S}'$) if

$$\forall y \in \mathcal{S}', y \sqsubseteq x \wedge \forall z \in \mathcal{S}, (\forall y \in \mathcal{S}', y \sqsubseteq z) \implies x \sqsubseteq z$$

- if it exists, the least upper bound is **unique**: if x, y are least upper bounds of \mathcal{S} , then $x \sqsubseteq y$ and $y \sqsubseteq x$, thus $x = y$ by antisymmetry
- notation: $x \sqcup y ::= \sqcup \{x, y\}$
- upper bounds and least upper bounds **may not exist**
- **dual notions**: lower bound, greatest lower bound (glb, noted $\sqcap \mathcal{S}'$)

Exercise: logical interpretations ?

Duality principle

So far all definitions admit a symmetric counterpart

- **dual relation**: given an order relation \sqsubseteq , \mathcal{R} defined by

$$x\mathcal{R}y \iff y \sqsubseteq x$$

is also an order relation

- thus all properties that can be proved about \sqsubseteq also have a symmetric property that also holds

This is the **duality principle**:

minimum element	maximum element
infimum	supremum
lower bound	upper bound
greatest lower bound	least upper bound

... more to follow

Complete lattice

Definition: complete lattice

A **complete lattice** is a tuple $(\mathcal{S}, \sqsubseteq, \perp, \top, \sqcup, \sqcap)$ where:

- $(\mathcal{S}, \sqsubseteq)$ is a poset
- \perp is the infimum of \mathcal{S}
- \top is the supremum of \mathcal{S}
- any subset \mathcal{S}' of \mathcal{S} has a lub $\sqcup \mathcal{S}'$ and a glb $\sqcap \mathcal{S}'$

Properties:

- $\perp = \sqcup \emptyset = \sqcap \mathcal{S}$
- $\top = \sqcap \emptyset = \sqcup \mathcal{S}$

Example:

the **powerset** $(\mathcal{P}(\mathcal{S}), \subseteq, \emptyset, \mathcal{S}, \cup, \cap)$ of set \mathcal{S} is a complete lattice

Lattice

The existence of lubs and glbs for all subsets is often a very strong property, that may not be met:

Definition: lattice

A **lattice** is a tuple $(\mathcal{S}, \sqsubseteq, \perp, \top, \sqcup, \sqcap)$ where:

- $(\mathcal{S}, \sqsubseteq)$ is a poset
 - \perp is the infimum of \mathcal{S}
 - \top is the supremum of \mathcal{S}
 - any **pair** $\{x, y\}$ of \mathcal{S} has a lub $x \sqcup y$ and a glb $x \sqcap y$
-
- let $\mathcal{Q} = \{q \in \mathbb{Q} \mid 0 \leq q \leq 1\}$;
then (\mathcal{Q}, \leq) is a **lattice** but **not a complete lattice**
indeed, $\{q \in \mathcal{Q} \mid q \leq \frac{\sqrt{2}}{2}\}$ has no lub in \mathcal{Q}
 - property: a **finite** lattice is also a complete lattice

Chains

Definition: increasing chain

Let $(\mathcal{S}, \sqsubseteq)$ be a poset and $\mathcal{C} \subseteq \mathcal{S}$.

It is an **increasing chain** if and only if

- it has an infimum (thus it is not empty)
- poset $(\mathcal{C}, \sqsubseteq)$ is total (i.e., any two elements can be compared)

Example, in the powerset $(\mathcal{P}(\mathbb{N}), \subseteq)$:

$$\mathcal{C} = \{c_i \mid i \in \mathbb{N}\} \quad \text{where} \quad c_i = \{2^0, 2^2, \dots, 2^i\}$$

Definition: increasing chain condition

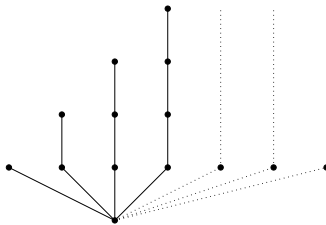
The poset $(\mathcal{S}, \sqsubseteq)$ **satisfies the increasing chain condition** if and only if any increasing chain $\mathcal{C} \subseteq \mathcal{S}$ is finite.

Complete partial orders

Definition: complete partial order

A **complete partial order** (cpo) is a poset $(\mathcal{S}, \sqsubseteq)$ such that any increasing chain \mathcal{C} of \mathcal{S} has a least upper bound. A **pointed cpo** is a cpo with an infimum \perp .

- clearly, any complete lattice is a cpo
- the opposite is not true:



Outline

Towards a constructive definition of the automata semantics

We now look for a **constructive version of the automaton semantics** as hinted by the following observations

Observation 1: $\mathcal{L}[\mathcal{A}] = \llbracket \mathcal{A} \rrbracket(q_f)$ where

$$\begin{aligned} \llbracket \mathcal{A} \rrbracket : Q &\longrightarrow \mathcal{P}(L^*) \\ q &\longmapsto \{w \in L^* \mid \exists n, w = a_0 a_1 \dots a_n \\ &\quad \exists q_0 \dots q_{n-1} \in Q, q_i \xrightarrow{a_0} q_0 \xrightarrow{a_1} \dots q_{n-1} \xrightarrow{a_n} q\} \end{aligned}$$

Observation 2: $\llbracket \mathcal{A} \rrbracket = \dot{\bigcup}_{n \in \mathbb{N}} \llbracket \mathcal{A} \rrbracket_n$ where

$$\begin{aligned} \llbracket \mathcal{A} \rrbracket_n : Q &\longrightarrow \mathcal{P}(L^*) \\ q &\longmapsto \{a_0 a_1 \dots a_{n-1} \mid \\ &\quad \exists q_0 \dots q_{n-2} \in Q, q_i \xrightarrow{a_0} q_0 \xrightarrow{a_1} \dots q_{n-1} \xrightarrow{a_{n-1}} q\} \end{aligned}$$

Observation 3: $\llbracket \mathcal{A} \rrbracket_{n+1}$ can be computed directly from $\llbracket \mathcal{A} \rrbracket_n$

$$\llbracket \mathcal{A} \rrbracket_{n+1}(q) = \bigcup_{q' \in Q} \{wa \mid w \in \llbracket \mathcal{A} \rrbracket_n(q') \wedge q' \xrightarrow{a} q\}$$

Towards a constructive definition of the automata semantics

Alternate approach:

- 1 Let $\llbracket \mathcal{A} \rrbracket_n$ denote **recognized words of length at most n** :

$$\llbracket \mathcal{A} \rrbracket(q) = \{w \in \llbracket \mathcal{A} \rrbracket(q) \mid \text{length}(w) \leq n\}$$

- 2 Compute $\llbracket \mathcal{A} \rrbracket_{n+1}$ from $\llbracket \mathcal{A} \rrbracket_n$
- 3 Define the semantics of the automaton as the union of the iterates of this sequence:

$$\llbracket \mathcal{A} \rrbracket = \bigcup_{n \in \mathbb{N}} \llbracket \mathcal{A} \rrbracket_n$$

In the following, we **study such a way of defining semantics**, based on **general mathematical tools**, that we will use throughout the course

Operators over a poset

Definition: operators and orderings

Let $(\mathcal{S}, \sqsubseteq)$ be a poset and $\phi : \mathcal{S} \rightarrow \mathcal{S}$ be an operator over \mathcal{S} . Then, ϕ is:

- **monotone** if and only if $\forall x, y \in \mathcal{S}, x \sqsubseteq y \implies \phi(x) \sqsubseteq \phi(y)$
- **continuous** if and only if, for any chain $\mathcal{S}' \subseteq \mathcal{S}$ then:

$$\begin{cases} \text{if } \sqcup \mathcal{S}' \text{ exists, so does } \sqcup \{\phi(x) \mid x \in \mathcal{S}'\} \\ \text{and } \phi(\sqcup \mathcal{S}') = \sqcup \{\phi(x) \mid x \in \mathcal{S}'\} \end{cases}$$
- **\sqcup -preserving** if and only if:

$$\forall \mathcal{S}' \subseteq \mathcal{S}, \begin{cases} \text{if } \sqcup \mathcal{S}' \text{ exists, then } \sqcup \{\phi(x) \mid x \in \mathcal{S}'\} \text{ exists} \\ \text{and } \phi(\sqcup \mathcal{S}') = \sqcup \{\phi(x) \mid x \in \mathcal{S}'\} \end{cases}$$

Notes:

- “monotone” in English means “*croissante*” in French ; “*décroissante*” translates into “anti-monotone” and “monotone” into “*isotone*”
- the dual of “monotone” is “monotone”

Operators over a poset

A few interesting properties:

Continuity implies monotonicity

If ϕ is continuous, then it is also **monotone**

We assume ϕ is continuous, and $x, y \in \mathcal{S}$ are such that $x \sqsubseteq y$:
Then $\{x, y\}$ is a chain with lub y , thus $\phi(x) \sqcup \phi(y)$ exists and is equal to $\phi(\sqcup\{x, y\}) = \phi(y)$. Therefore $\phi(x) \sqsubseteq \phi(y)$.

\sqcup -preserving implies monotonicity

If ϕ preserves \sqcup , then it is also **monotone**

Same argument.

Fixpoints

Definition: fixpoints

Let $(\mathcal{S}, \sqsubseteq)$ be a poset and $\phi : \mathcal{S} \rightarrow \mathcal{S}$ be an operator over \mathcal{S} .

- a **fixpoint** of ϕ is an element x such that $\phi(x) = x$
- a **pre-fixpoint** of ϕ is an element x such that $x \sqsubseteq \phi(x)$
- a **post-fixpoint** of ϕ is an element x such that $\phi(x) \sqsubseteq x$
- the **least fixpoint** $\text{lfp } \phi$ of ϕ (if it exists, it is unique) is the smallest fixpoint of ϕ
- the **greatest fixpoint** $\text{gfp } \phi$ of ϕ (if it exists, it is unique) is the greatest fixpoint of ϕ

Note: the existence of a least fixpoint, a greatest fixpoint or even a fixpoint is *not guaranteed*; we will see several theorems that establish their existence under specific assumptions...

Tarski's Theorem

Theorem

Let $(\mathcal{S}, \sqsubseteq, \perp, \top, \sqcup, \sqcap)$ be a complete lattice and $\phi : \mathcal{S} \rightarrow \mathcal{S}$ be a monotone operator over \mathcal{S} . Then:

- ❶ ϕ has a least fixpoint **lfp** ϕ and **lfp** $\phi = \sqcap \{x \in \mathcal{S} \mid \phi(x) \sqsubseteq x\}$.
- ❷ ϕ has a greatest fixpoint **gfp** ϕ and **gfp** $\phi = \sqcup \{x \in \mathcal{S} \mid x \sqsubseteq \phi(x)\}$.
- ❸ the set of fixpoints of ϕ is a complete lattice.

Proof of point 1:

We let $X = \{x \in \mathcal{S} \mid \phi(x) \sqsubseteq x\}$ and $x_0 = \sqcap X$.

Let $y \in X$:

- $x_0 \sqsubseteq y$ by definition of the glb;
- thus, since ϕ is monotone, $\phi(x_0) \sqsubseteq \phi(y)$;
- thus, $\phi(x_0) \sqsubseteq y$ since $\phi(y) \sqsubseteq y$, by definition of X .

Therefore $\phi(x_0) \sqsubseteq x_0$, since $x_0 = \sqcap X$ and $\phi(x_0)$ is a lower bound.

Tarski's Theorem

We proved that $\phi(x_0) \sqsubseteq x_0$. We derive from this that:

- $\phi(\phi(x_0)) \sqsubseteq \phi(x_0)$ since ϕ is monotone;
- $\phi(x_0)$ is a post-fixpoint of ϕ , thus $\phi(x_0) \in X$;
- $x_0 \sqsubseteq \phi(x_0)$ by definition of the greatest lower bound

We have established both inclusions so $\phi(x_0) = x_0$.

If x_1 is another fixpoint, then $x_1 \in X$, so $x_0 \sqsubseteq x_1$.

Proof of point 2: similar, by duality.

Proof of point 3:

- if X is a set of fixpoints of ϕ , we need to consider ϕ over $\{y \in \mathcal{S} \mid y \sqsubseteq_{\mathcal{S}} \sqcap X\}$ to establish the existence of a **glb of X in the poset of fixpoints**
- the existence of **least upper bounds in the poset of fixpoints** follows by duality

Tarski's theorem: example (1)

A function over the powerset:

We consider a set \mathcal{E} , and a subset $\mathcal{A} \subseteq \mathcal{E}$

We let:

$$\begin{array}{ccc} f : \mathcal{P}(\mathcal{E}) & \longrightarrow & \mathcal{P}(\mathcal{E}) \\ X & \longmapsto & X \cup \mathcal{A} \end{array}$$

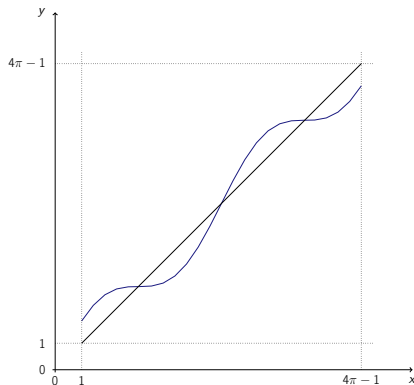
Exercise:

- apply Tarski's theorem, characterize the least and greatest fixpoints

Tarski's theorem: example (2)

Function:

$$\begin{array}{rcl}
 f : [1, 4\pi - 1] & \longrightarrow & [1, 4\pi - 1] \\
 x & \longmapsto & x + \sin x
 \end{array}$$



Exercise:

- apply Tarski's theorem, and derive the fixpoints of the function

Automata example, fixpoint definition

Lattice:

- $\mathcal{S} = Q \rightarrow \mathcal{P}(L^*)$
- the ordering is the pointwise extension $\dot{\sqsubseteq}$ of \sqsubseteq

Operator:

- we let $\phi_0 : \mathcal{S} \rightarrow \mathcal{S}$ be defined by

$$\phi_0(f) = \lambda(q \in Q) \cdot \bigcup_{q' \in Q} \{wa \mid w \in f(q') \wedge q' \xrightarrow{a} q\}$$
- we let $\phi : \mathcal{S} \rightarrow \mathcal{S}$ be defined by

$$\phi(f) = \lambda(q \in Q) \cdot \begin{cases} f(q) \cup \phi_0(f)(q_i) \cup \{\epsilon\} & \text{if } q = q_i \\ f(q) \cup \phi_0(f)(q) & \text{otherwise} \end{cases}$$

Proof steps to complete:

- **the existence of $\text{lfp } \phi$** follows from Tarski's theorem
- **the equality $\text{lfp } \phi = \llbracket \mathcal{A} \rrbracket$** can be established by induction and double inclusion... but there is a simpler way

Kleene's Theorem

Tarski's theorem guarantees existence of an lfp, but is not constructive.

Theorem

Let $(\mathcal{S}, \sqsubseteq, \perp)$ be a pointed cpo and $\phi : \mathcal{S} \rightarrow \mathcal{S}$ be a continuous operator over \mathcal{S} . Then ϕ has a least fixpoint, and

$$\text{lfp } \phi = \bigsqcup_{n \in \mathbb{N}} \phi^n(\perp)$$

First, we prove **the existence of the lub**:

Since ϕ is continuous, it is also monotone. We can prove by induction over n that $\{\phi^n(\perp) \mid n \in \mathbb{N}\}$ is a chain:

- $\phi^0(\perp) = \perp \sqsubseteq \phi(\perp)$ by definition of the infimum;
- if $\phi^n(\perp) \sqsubseteq \phi^{n+1}(\perp)$, then
$$\phi^{n+1}(\perp) = \phi(\phi^n(\perp)) \sqsubseteq \phi(\phi^{n+1}(\perp)) = \phi^{n+2}(\perp)$$

By definition of the cpo structure, the lub exists. We let x_0 denote it.

Kleene's Theorem

Secondly, we prove that **it is a fixpoint of ϕ** :

Since ϕ is continuous, $\{\phi^{n+1}(\perp) \mid n \in \mathbb{N}\}$ has a lub, and

$$\begin{aligned}
 \phi(x_0) &= \phi(\sqcup\{\phi^n(\perp) \mid n \in \mathbb{N}\}) \\
 &= \sqcup\{\phi^{n+1}(\perp) \mid n \in \mathbb{N}\} && \text{by continuity of } \phi \\
 &= \perp \sqcup (\sqcup\{\phi^{n+1}(\perp) \mid n \in \mathbb{N}\}) && \text{by definition of } \perp \\
 &= x_0 && \text{by simple rewrite}
 \end{aligned}$$

Last, we show that it is the **least** fixpoint:

Let x_1 denote another fixpoint of ϕ . We show by induction over n that $\phi^n(\perp) \sqsubseteq x_1$:

- $\phi^0(\perp) = \perp \sqsubseteq x_1$ by definition of \perp ;
- if $\phi^n(\perp) \sqsubseteq x_1$, then $\phi^{n+1}(\perp) \sqsubseteq \phi(x_1) = x_1$ by monotony, and since x_1 is a fixpoint.

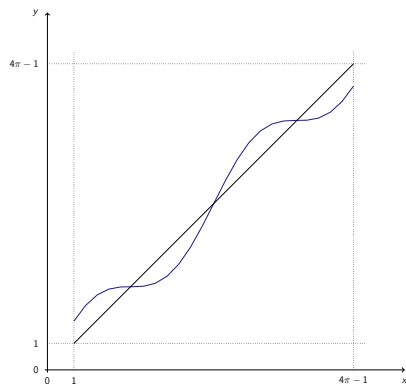
By definition of the lub, $x_0 \sqsubseteq x_1$

Kleene's theorem: example

Function:

$$f : [1, 4\pi - 1] \longrightarrow [1, 4\pi - 1]$$

$$x \longmapsto x + \sin x$$



Exercise:

- apply Kleene's theorem and sketch the iterations

Automata: constructive semantics

We can now state a **constructive definition** of the automaton semantics.
Operator ϕ is defined by

$$\phi(f) = \lambda(q \in Q) \cdot \begin{cases} f(q) \cup \phi_0(f)(q_i) \cup \{\epsilon\} & \text{if } q = q_i \\ f(q) \cup \phi_0(f)(q) & \text{otherwise} \end{cases}$$

Proof steps:

- ϕ is continuous
- thus, Kleene's theorem applies so **lfp** ϕ exists and
 $\text{lfp } \phi = \bigcup_{n \in \mathbb{N}} \phi^n(\perp) \dots$
 ... this actually saves the double inclusion proof to establish that
 $\llbracket \mathcal{A} \rrbracket = \text{lfp } \phi$

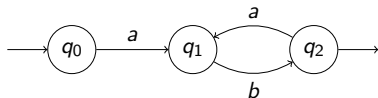
Furthermore, $\llbracket \mathcal{A} \rrbracket = \bigcup_{n \in \mathbb{N}} \phi^n(\perp)$.

This fixpoint definition will be very useful to infer or verify semantic properties.

Automata: constructive semantics iterates

A simple automaton:

$$\begin{aligned}
 L &= \{a, b\} & Q &= \{q_0, q_1, q_2\} \\
 q_i &= q_0 & q_f &= q_2 \\
 q_0 &\xrightarrow{a} q_1 & q_1 &\xrightarrow{b} q_2 & q_2 &\xrightarrow{a} q_1
 \end{aligned}$$



Iterates of function ϕ from \perp :

Iterate	0	1	2	3	4	5
q_0	\emptyset	$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon\}$
q_1	\emptyset	\emptyset	$\{a\}$	$\{a\}$	$\{a, aba\}$	$\{a, aba\}$
q_2	\emptyset	\emptyset	\emptyset	$\{ab\}$	$\{ab\}$	$\{ab, abab\}$

Duality principle

We can **extend the duality notion to fixpoints**:

monotone	monotone
anti-monotone	anti-monotone
post-fixpoint	pre-fixpoint
least fixpoint	greatest fixpoint
increasing chain	decreasing chain

Furthermore both Tarski's theorem and Kleene's theorem have a dual version (Tarski's theorem is its own dual).

On the topic of inductive reasoning...

Formalizing inductive definitions:

Definition based on
inference rules:

$$\frac{}{x_0 \in \mathcal{X}} \quad \frac{x \in \mathcal{X}}{f(x) \in \mathcal{X}}$$

Same property based on a
least-fixpoint:

$$\text{lfp}(Y \mapsto \{x_0\} \cup Y \cup \{f(x) \mid x \in Y\})$$

Proving the inclusion of a fixpoint in a given set:

- Let $\phi : \mathcal{S} \longrightarrow \mathcal{S}$ be a continuous operator
- Let $\mathcal{I} \in \mathcal{S}$ such that:

$$\forall x \in \mathcal{S}, x \sqsubseteq \mathcal{I} \implies \phi(x) \sqsubseteq \mathcal{I}$$

- We obviously have $\perp \sqsubseteq \mathcal{I}$
- We can prove that $\text{lfp } \phi \sqsubseteq \mathcal{I}$

Exercise: language of a grammar

Language of a grammar as a least-fixpoint

Assumptions:

- Alphabet \mathcal{A} , finite set of nodes \mathcal{N}
- Finite set of rules $\mathcal{R} \subseteq \mathcal{N} \times (\mathcal{A} \uplus \mathcal{N})^*$
- Starting node $S \in \mathcal{N}$

Questions:

- Define the set of words recognized by the grammar with inductive rules
- Do the same using a least-fixpoint

Hints:

- start with a function that maps each node into the set of words recognized by this node
- compute such a function by induction

Outline

Main points to remember

Foundations:

- **program semantics**: express program behaviors
- **target semantic property**: express proof goal
- **conservative approximation** usually required due to undecidability

Order relations:

- **counterpart for logical implication** (among other)
- will be pervasive in this course

Fixpoints and induction:

- **encode general iteration**
- will also be pervasive in this course

In the next lectures...

- Families of **semantics**, for a general model of programs
- Families of **semantic properties of programs**
- **Verification techniques:**
 - ▶ abstract interpretation based static analysis
 - ▶ machine assisted theorem proving
 - ▶ model checking

Next week: transition systems and operational semantics

Practical information about the course

The course will be taught by:

- **Marc Chevalier** (DIENS, TDs)
- **Sylvain Conchon** (LRI, Paris-Orsay, Model-Checking / SMT)
- **Jérôme Feret** (DIENS, Semantics, Abstract interpretation)
- **Xavier Rival** (DIENS, Semantics, Abstract interpretation, Coq)

Practical organization:

- 1h30 Cours + 2h00 TD or TP depending on week

Evaluation:

$$n = \frac{p + e}{2}$$

- **project** p : several projects will be proposed in a few weeks
- **exam** e : 30th of May, 2020