UNIVERSITAT DE BARCELONA

Joost J. Joosten

Associate Professor

University of Barcelona

21 OCTOBER 2020

GURETRUCK

FV Time Manager powered by Formally Verified Time Library

EUROPEAN UNION
EUROPEAN REGIONAL DEVELOPMENT FUND

GOBIERNO DE ESPAÑA
MINISTERIO DE CIENCIA, INNOVACIÓN Y UNIVERSIDADES

Bosch i Gimpera
UNIVERSITAT DE BARCELONA

DOCTORATS INDUSTRIALS
talent alliance!

Generalitat de Catalunya


WELCOME

THE REGULATIONS ARE LOCALLY STRICTLY STIPULATED BUT:

- Driving data is corrupted;
- Legal enforcement software flawed;
- Certain legal texts at the best ambiguous;
- And possibly not consistent or coherent.

Could logic be of any help?

Mathematical logic can check laws with clear-cut ontologies for

- Consistency;

Mathematical logic can check laws with clear-cut ontologies for

- Consistency;
- Tractability;

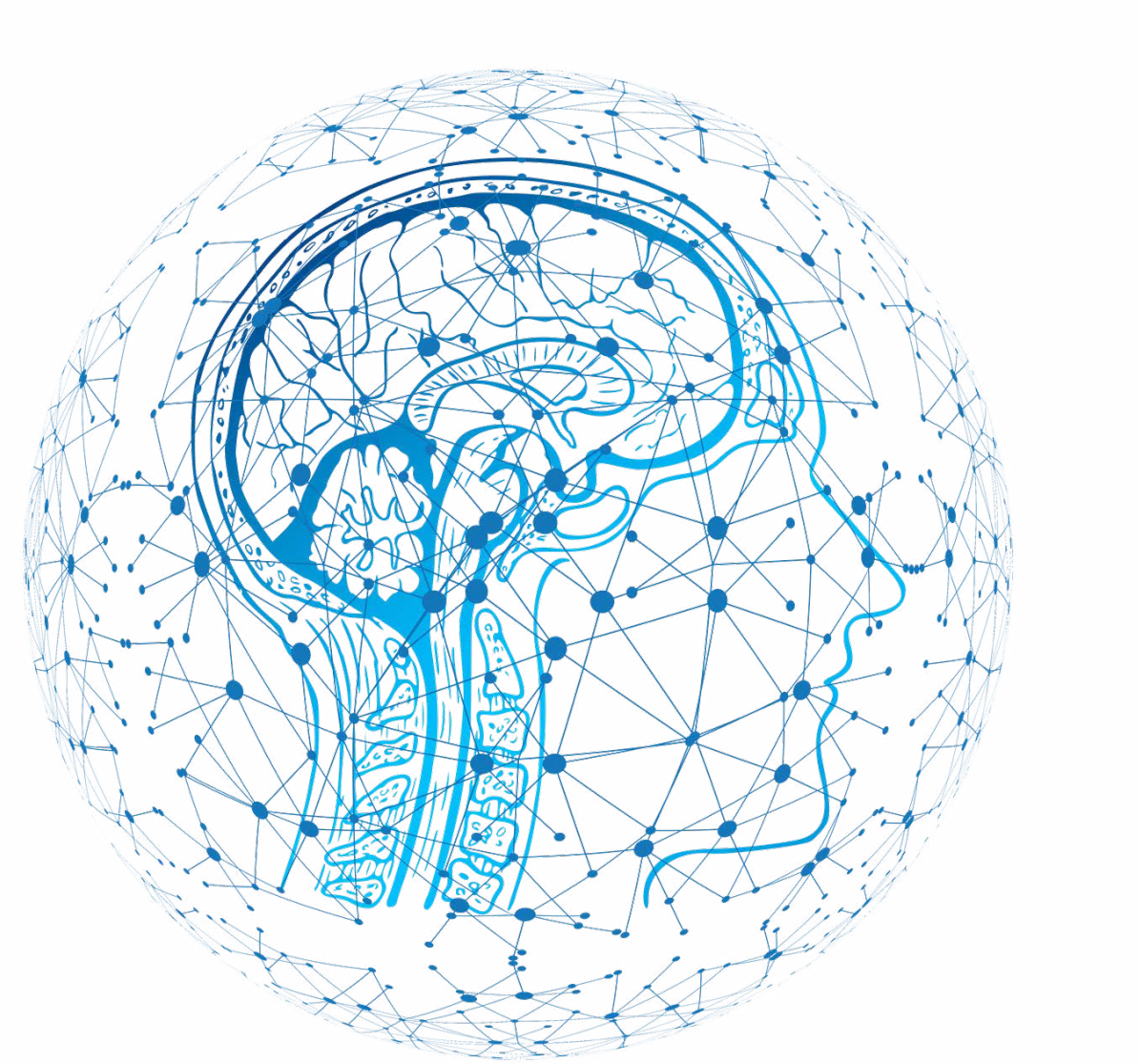
Mathematical logic can check laws with clear-cut ontologies for

- Consistency;
- Tractability;
- Meeting physics constraints;

Mathematical logic can check laws with clear-cut ontologies for

- Consistency;
- Tractability;
- Meeting physics constraints;
- Certificate that software meets the specification.

OUTLINE OF MY TALK:



OUTLINE OF MY TALK:

- Intro

OUTLINE OF MY TALK:

- Intro
- Bugs in Software

OUTLINE OF MY TALK:

- Intro
- Bugs in Software
- Our programming paradigm

OUTLINE OF MY TALK:

- Intro
- Bugs in Software
- Our programming paradigm
- Our first application:
  Formally Verified Time Library

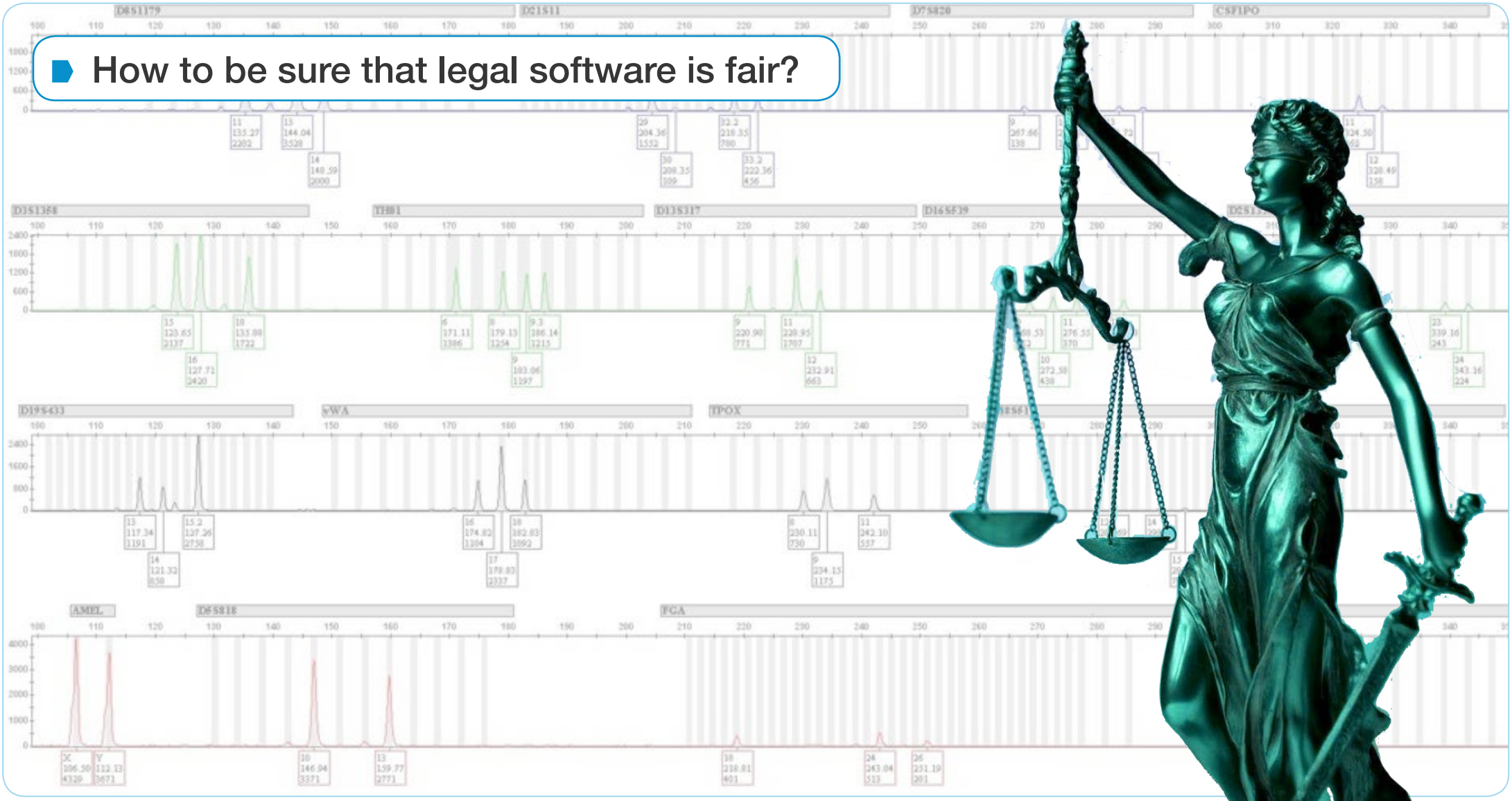
OUTLINE OF MY TALK:

- Intro
- Bugs in Software
- Our programming paradigm
- Our first application:
  Formally Verified Time Library
- The future ahead of us


Critical software should be error-free!

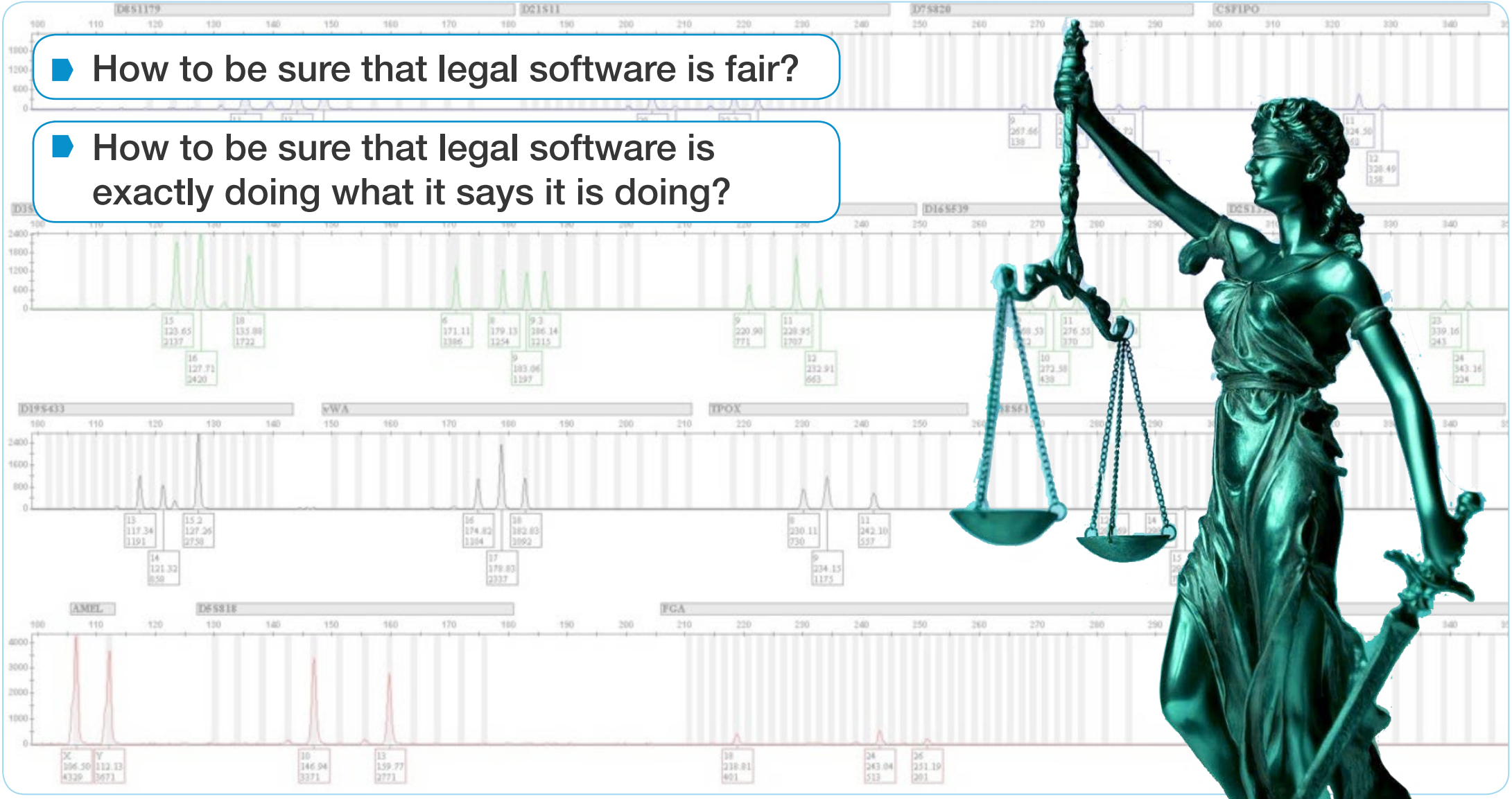
AI in society and law

- How to be sure that legal software is fair?



AI in society and law

- How to be sure that legal software is fair?
- How to be sure that legal software is exactly doing what it says it is doing?



AI in society and law

- How to be sure that legal software is fair?
- How to be sure that legal software is exactly doing what it says it is doing?
- How to provide full legal transparency to legal software without necessarily disclosing the proprietary source code?

IMPERATIVE PARADIGM.

Imperative programming was the first paradigm to be used when computers where invented.

It is close to the machine and it gives sequential imperative instructions:

IMPERATIVE PARADIGM.

Imperative programming was the first paradigm to be used when computers where invented.
It is close to the machine and it gives sequential imperative instructions:

- Store this number in the memory.

IMPERATIVE PARADIGM.

Imperative programming was the first paradigm to be used when computers where invented.
It is close to the machine and it gives sequential imperative instructions:

- Store this number in the memory.
- Move this number to another location in the memory.

IMPERATIVE PARADIGM.

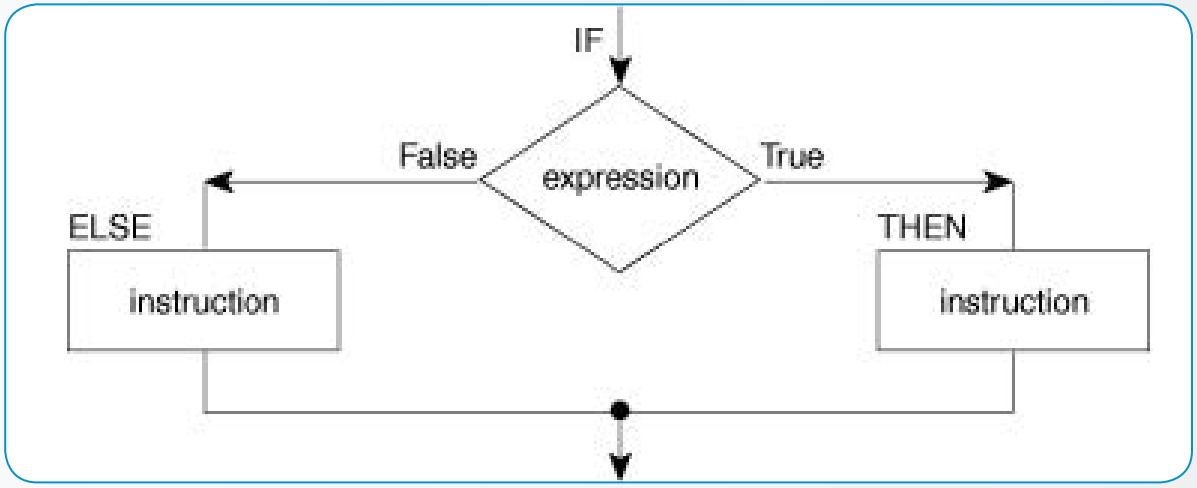
Imperative programming was the first paradigm to be used when computers where invented.
It is close to the machine and it gives sequential imperative instructions:

- Store this number in the memory.
- Move this number to another location in the memory.
- Repeat process while some condition holds.

IMPERATIVE PARADIGM.

Imperative programming was the first paradigm to be used when computers where invented.
It is close to the machine and it gives sequential imperative instructions:

- Store this number in the memory.
- Move this number to another location in the memory.
- Repeat process while some condition holds.
- Etc.



FUNCTIONAL PARADIGM.

Functional programming is a more advanced paradigm that is close to the mathematical language.

We do not tell the computer how to operate.

Rather, we tell the computer what the mathematical definitions are.

FUNCTIONAL PARADIGM.

Functional programming is a more advanced paradigm that is close to the mathematical language.
We do not tell the computer how to operate.
Rather, we tell the computer what the mathematical definitions are.

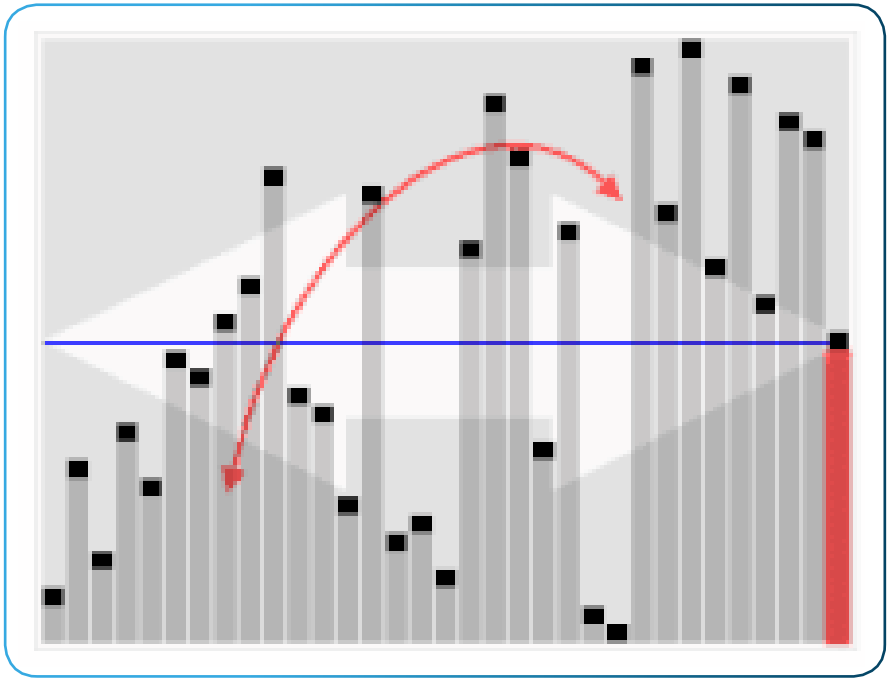
FUNCTIONAL PARADIGM.

Functional programming is a more advanced paradigm that is close to the mathematical language.
We do not tell the computer how to operate.
Rather, we tell the computer what the mathematical definitions are.

- Advantage: We can easily reason about our programs and prove their correctness.

TEST CASE: SORTING ALGORITHM

- Quicksort is an efficient algorithm which sorts lists of numbers in lexicographical order.



FOR INSTANCE, APPLYING QUICKSORT TO [2,6,1,7] GIVES [1,2,6,7]

IMPERATIVE VS FUNCTIONAL

IMPERATIVE CODE:

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[⌊(hi + lo) / 2⌋]
    i := lo - 1
    j := hi + 1
    loop forever
        do
            i := i + 1
        while A[i] < pivot
        do
            j := j - 1
        while A[j] > pivot
        if i ≥ j then
            return j
        swap A[i] with A[j]
```

IMPERATIVE VS FUNCTIONAL

IMPERATIVE CODE:

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[⌊(hi + lo) / 2⌋]
    i := lo - 1
    j := hi + 1
    loop forever
        do
            i := i + 1
        while A[i] < pivot
        do
            j := j - 1
        while A[j] > pivot
        if i ≥ j then
            return j
        swap A[i] with A[j]
```

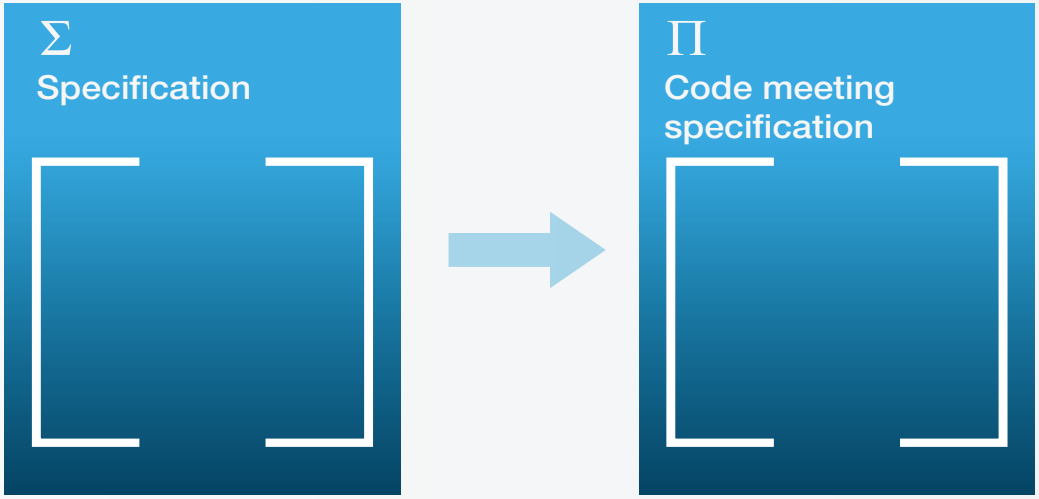
FUNCTIONAL CODE:

```
algorithm quicksort (l : list) :=
          case l of
            [] -> []
            [a] -> [a]
            (a :: as) -> quicksort [x ∈ l : x ≤ a] ++
                         quicksort [a ∈ l: x > a]
```

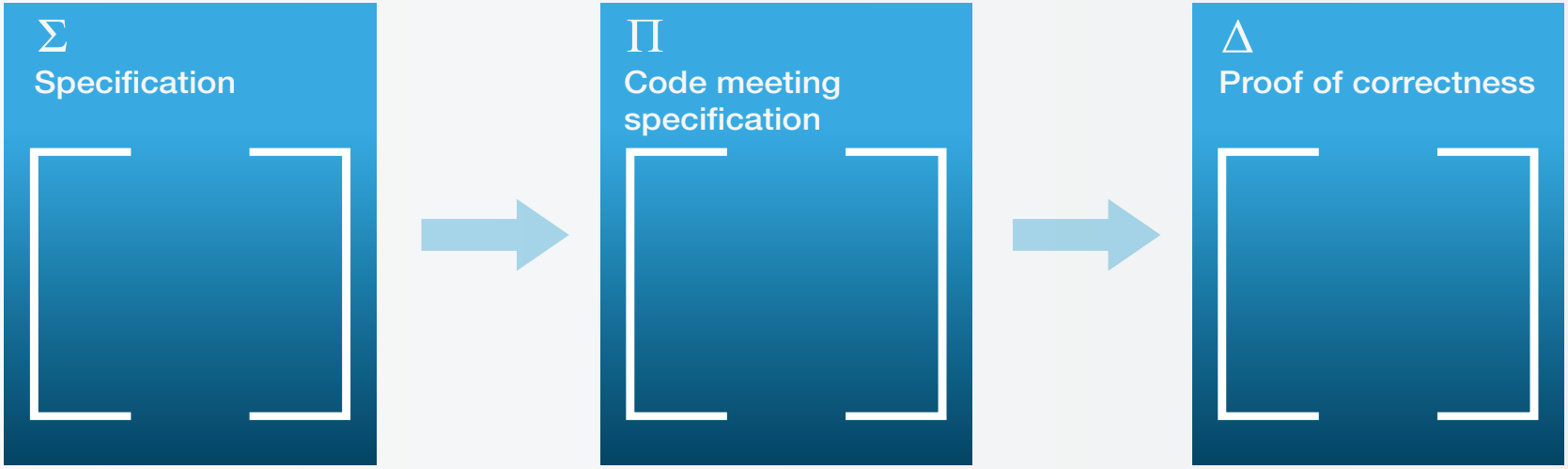
DEPENDENT TYPE THEORY CAN REASON ABOUT FUNCTIONAL CODE AND CAN BE USED TO PROVE CORRECTNESS OF THE CODE

VERIFICATION OF FUNCTIONAL CODE

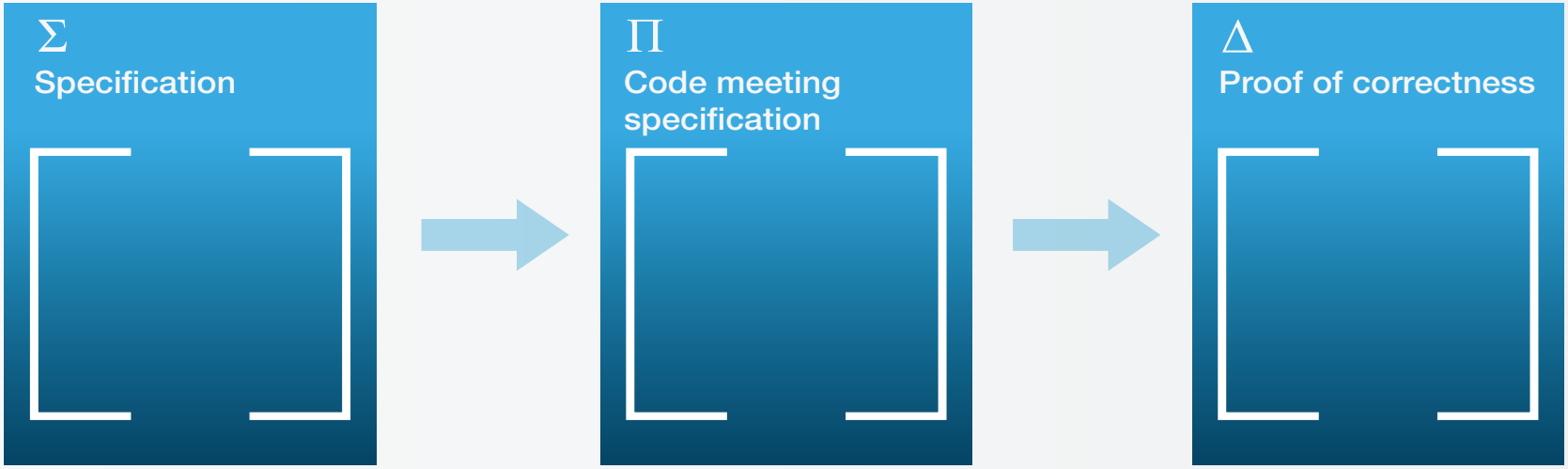
VERIFICATION OF FUNCTIONAL CODE



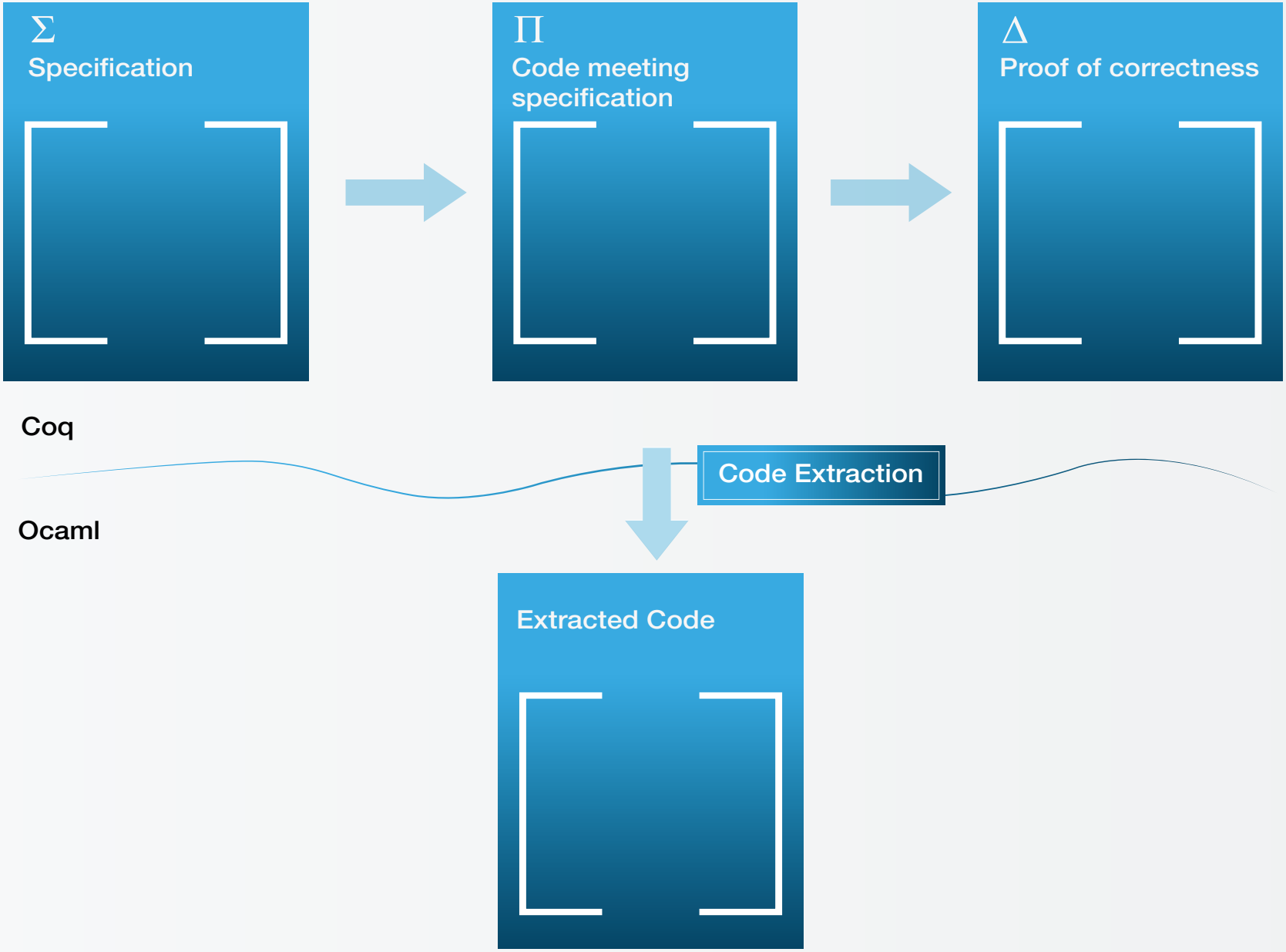
VERIFICATION OF FUNCTIONAL CODE



VERIFICATION OF FUNCTIONAL CODE



VERIFICATION OF FUNCTIONAL CODE



EXAMPLE THEOREMS

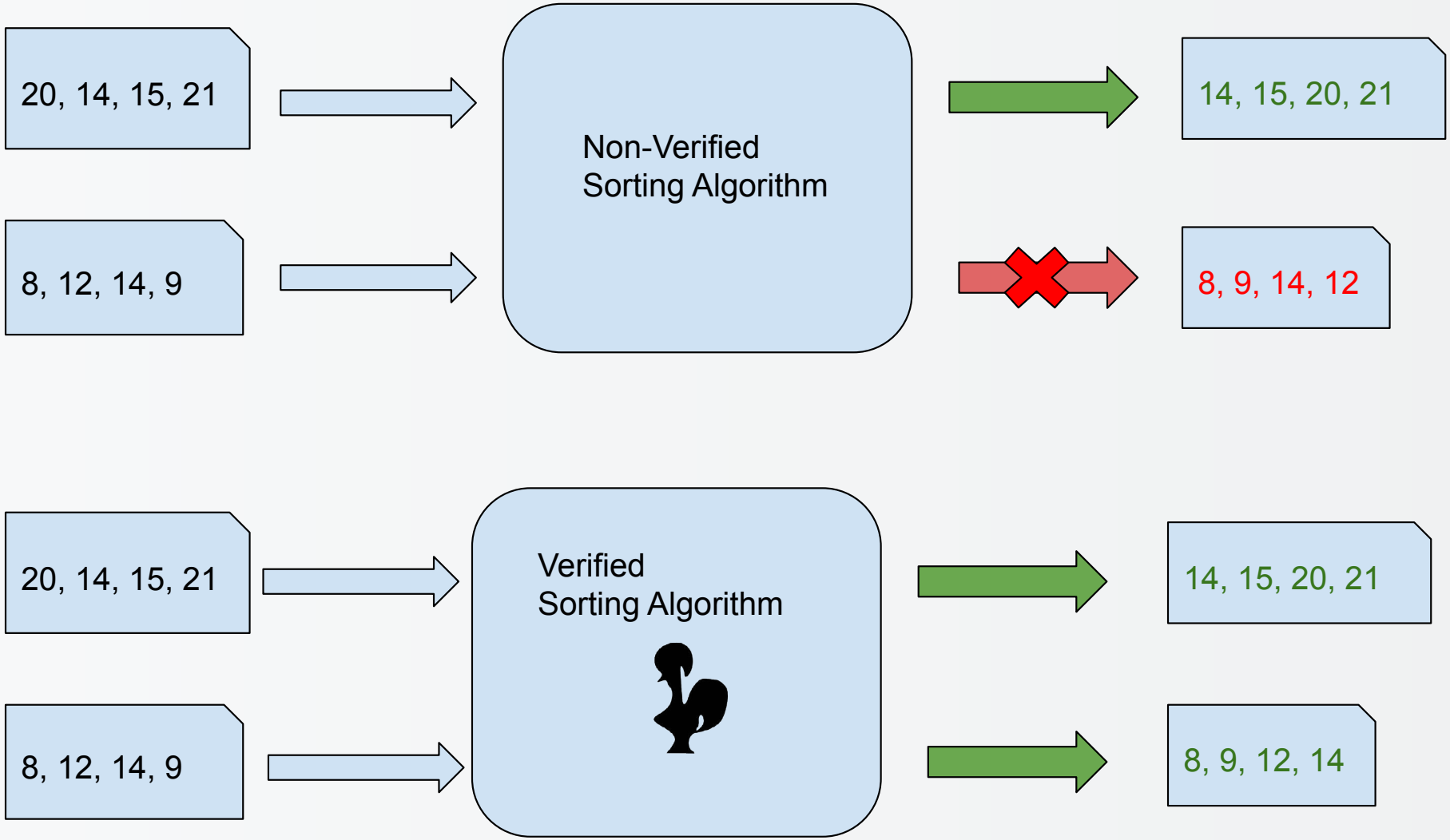
```
Lemma leq_trans a b c : a <= b -> b <= c -> a <= c.
Proof. by elim: a b c => [|i IHn] [|b] [|c] //;
      apply: IHn b c. Qed.
```

EXAMPLE THEOREMS

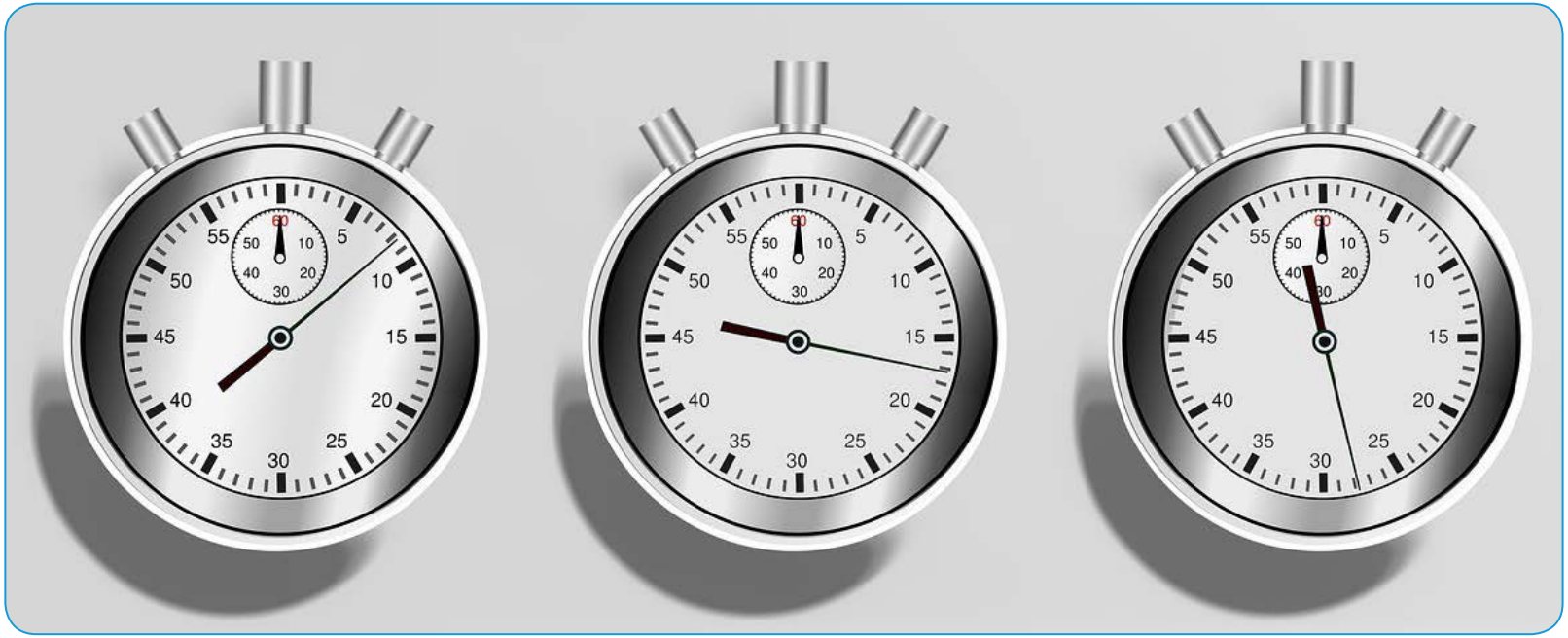
```
Lemma leq_trans a b c : a <= b -> b <= c -> a <= c.
Proof. by elim: a b c => [|i IHn] [|b] [|c] //;
      apply: IHn b c. Qed.
```

```
Lemma quicksort_correct :
      forall l,
      permutation l (quicksort l)
      /\ is_sorted (quicksort l).
```

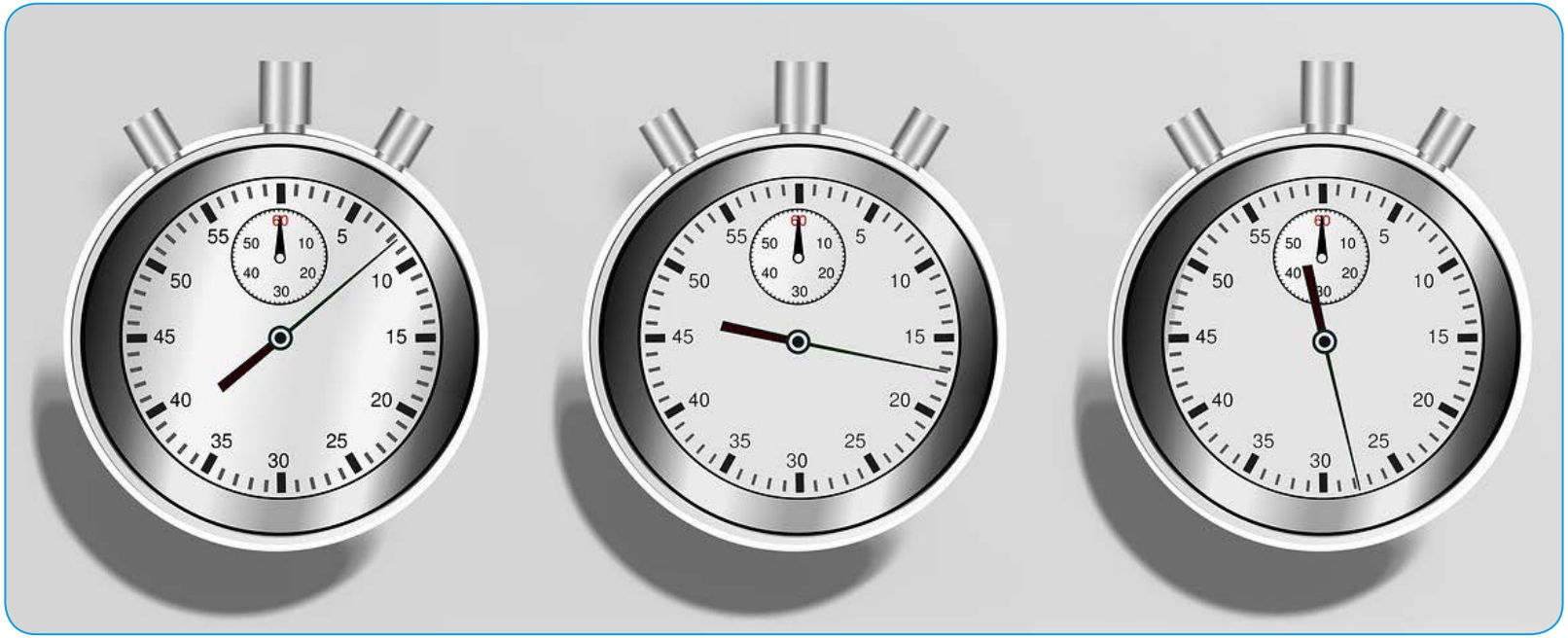
THE GOAL


20, 14, 15, 21
8, 12, 14, 9
Non-Verified
Sorting Algorithm
14, 15, 20, 21
8, 9, 14, 12
20, 14, 15, 21
8, 12, 14, 9
Verified
Sorting Algorithm
14, 15, 20, 21
8, 9, 12, 14

TIME MANAGEMENT IS IMPORTANT

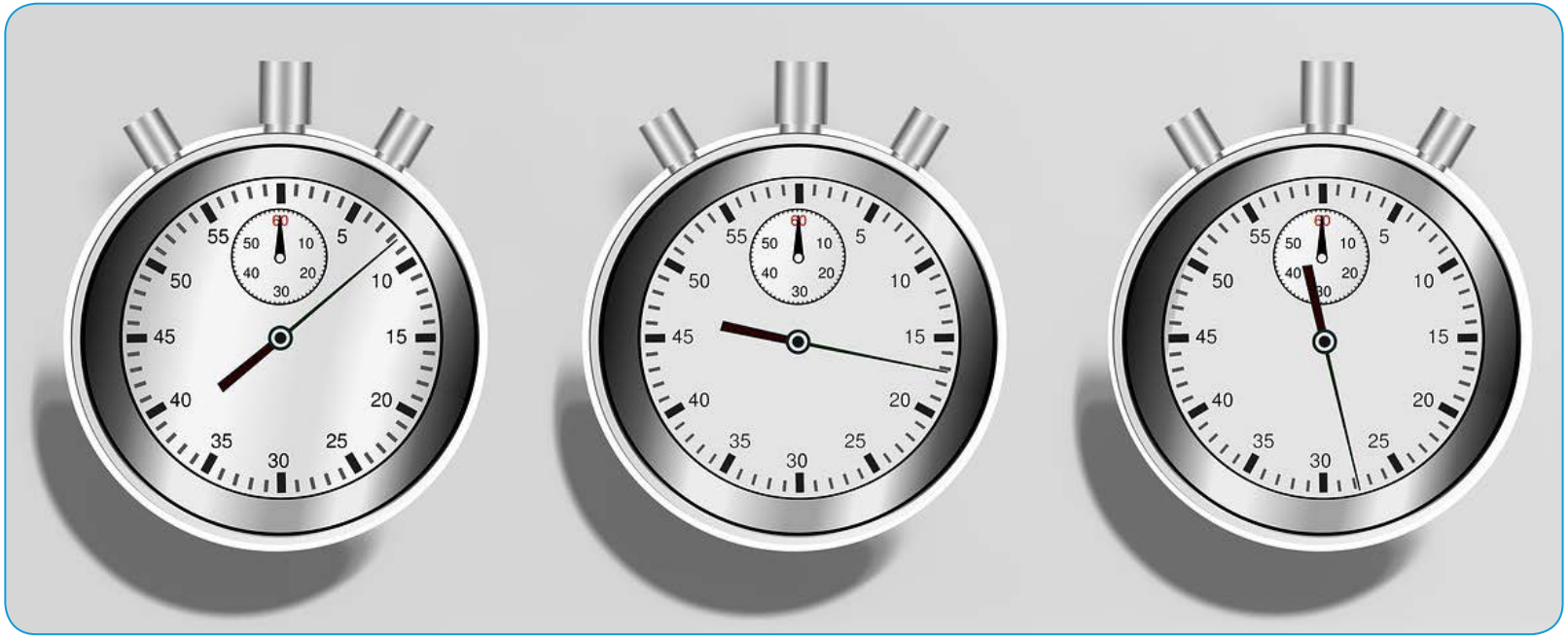


TIME MANAGEMENT IS IMPORTANT



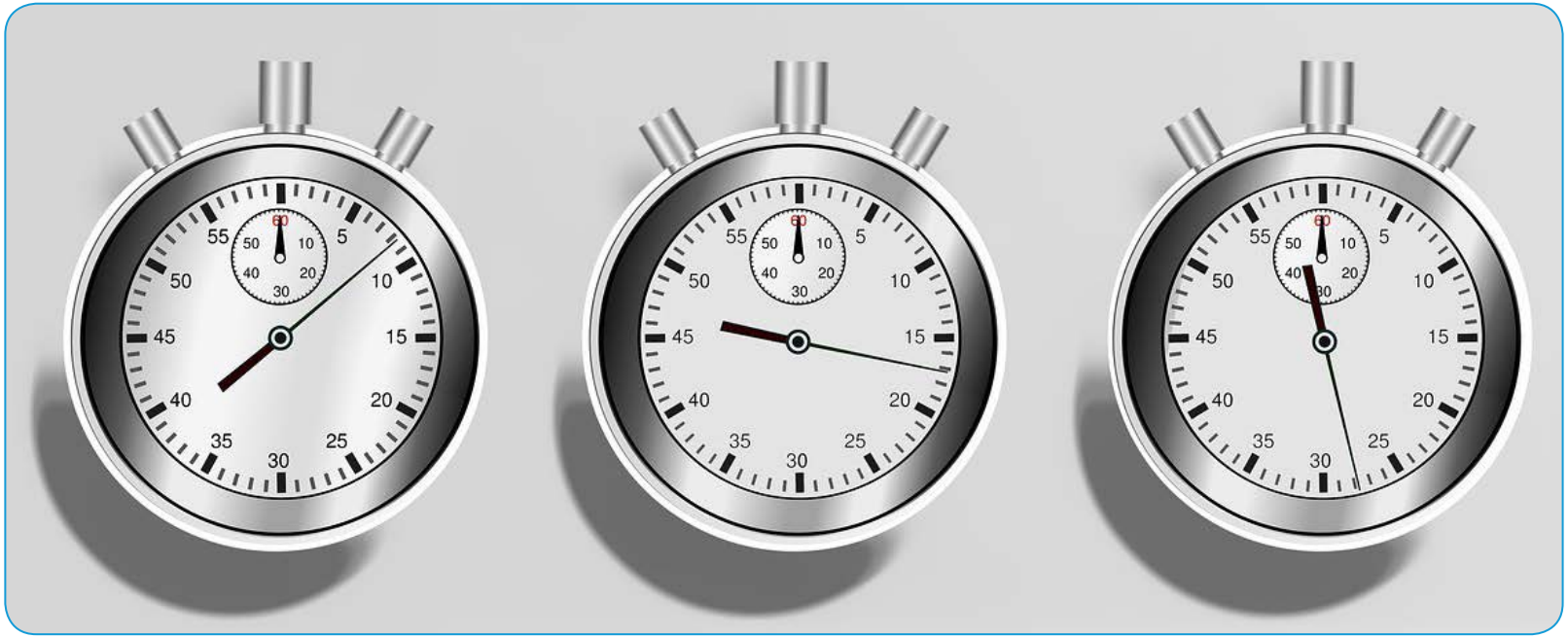
- Time management is a crucial part of many industry projects.

TIME MANAGEMENT IS IMPORTANT



- Time management is a crucial part of many industry projects.
- We have developed our own verified library to be integrated in a bigger project

TIME MANAGEMENT IS IMPORTANT



- Time management is a crucial part of many industry projects.
- We have developed our own verified library to be integrated in a bigger project.
- Bigger project: legal software involving time and durations

  for example

  full tachograph software.

SPECIFICATION

- Mathematical definitions
- Blueprint of the running sofware
- Unambiguosly defines sofware intended behaviour

```
| Rec.WordF n ⇒
| Rec.ArrayF t' n₀ ⇒
  (fix array_of_word n from :=
     match n with
     | O ⇒ Modify (InitSliceWithCapacity n₀) ^(vdst)
     | S n' ⇒
       Declare (go_rec_type t') (λ vt' ⇒
         (array_of_word n' (from + Rec.len t');
          go_of_word t' vt' vsrc from;
          Modify Append ^(vdst, vt')))
     end) n₀ from
| Rec.RecF fs ⇒
  (fix rec_of_word fs vdst from :=
     match fs with
     | [] ⇒ Modify (@SetConst (Struct []) tt) ^(vdst)
     | (_, f) :: fs' ⇒
       Declare (go_rec_type f) (λ vf ⇒
```

SPECIFICATION

- Mathematical definitions
- Blueprint of the running sofware
- Unambiguosly defines sofware intended behaviour

- Given a time in format Year- Month- Day- hour : minute : second, obtain the number of seconds elapsed since 1970-01-01-00:00:00 until then
- Mathematical definition:

$$timestamp(T) := |\{t : 1970\text{-}01\text{-}01\text{-}00{:}00{:}00 \leq t < T\}|$$

```
| Rec.WordF n ⇒
| Rec.ArrayF t' n0 ⇒
          Modify Append ^(vdst, vt')))
    end) n0 from
| Rec.RecF fs ⇒
  (fix rec_of_word fs vdst from :=
     match fs with
     | [] ⇒ Modify (@SetConst (Struct []) tt) ^(vdst)
     | (_, f) :: fs' ⇒
       Declare (go_rec_type f) (λ vf ⇒
```

SPECIFICATION

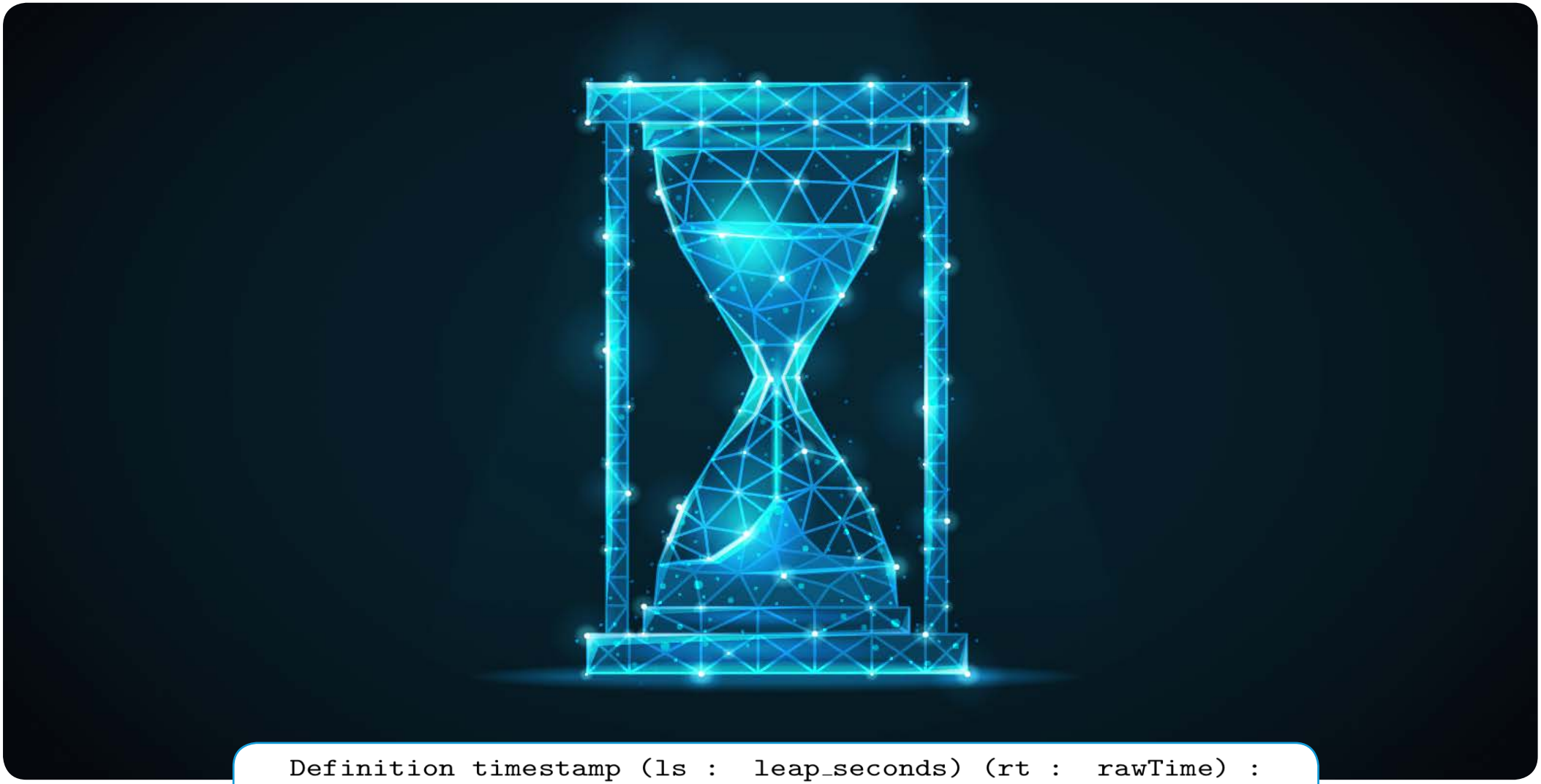
- Mathematical definitions
- Blueprint of the running sofware
- Unambiguosly defines sofware intended behaviour

- Given a time in format Year- Month- Day- hour : minute : second, obtain the number of seconds elapsed since 1970-01-01-00:00:00 until then
- Mathematical definition:

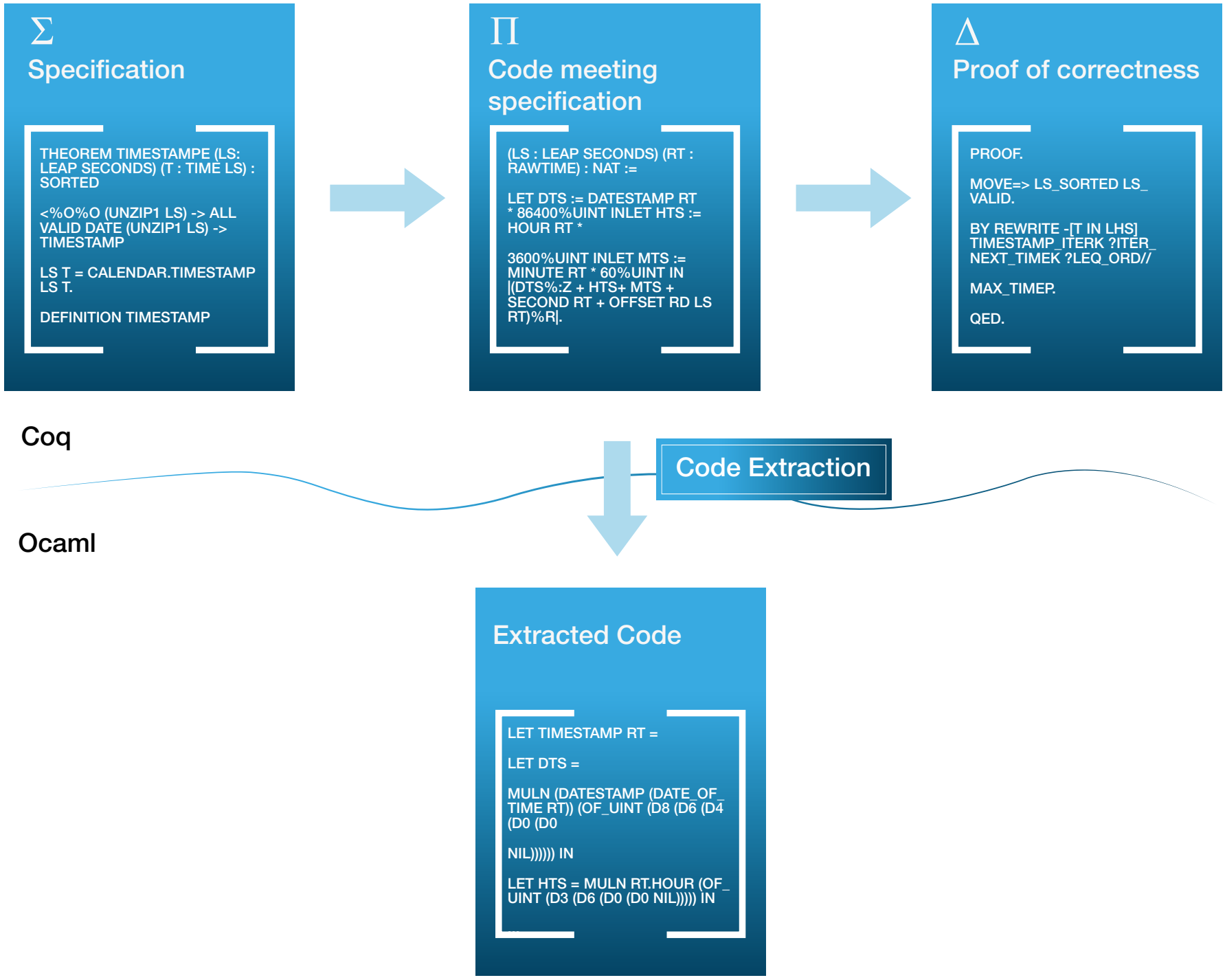
$$timestamp(T) := |\{t : 1970\text{-}01\text{-}01\text{-}00{:}00{:}00 \leq t < T\}|$$

- Different implementations might correspond to the same specification
- Is not obvious when the implementation satisfies its specification

```
match fs with
| [] ⇒ Modify (@SetConst (Struct []) tt) ~(vdst)
| (_, f) :: fs' ⇒
  Declare (go_rec_type f) (λ vf ⇒
```



```
Definition timestamp (ls :  leap_seconds) (rt :  rawTime) :
nat :=
  let Dts := datestamp rt * 86400%uint in
  let hts := hour rt * 3600%uint in
  let mts := minute rt * 60%uint in
  ⌊(Dts%:Z + hts + mts + second rt + offset_rd ls rt)%R⌋.
```


Σ
Specification
THEOREM TIMESTAMPE (LS: LEAP SECONDS) (T : TIME LS) : SORTED
<%O%O (UNZIP1 LS) -> ALL VALID DATE (UNZIP1 LS) -> TIMESTAMP
LS T = CALENDAR.TIMESTAMP LS T.
DEFINITION TIMESTAMP
Π
Code meeting specification
(LS : LEAP SECONDS) (RT : RAWTIME) : NAT :=
LET DTS := DATESTAMP RT * 86400%UINT INLET HTS := HOUR RT *
3600%UINT INLET MTS := MINUTE RT * 60%UINT IN |(DTS%:Z + HTS+ MTS + SECOND RT + OFFSET RD LS RT)%R|.
Δ
Proof of correctness
PROOF.
MOVE=> LS_SORTED LS_ VALID.
BY REWRITE -[T IN LHS] TIMESTAMP_ITERK ?ITER_ NEXT_TIMEK ?LEQ_ORD//
MAX_TIMEP.
QED.
Coq
Code Extraction
Ocaml
Extracted Code
LET TIMESTAMP RT =
LET DTS =
MULN (DATESTAMP (DATE_OF_ TIME RT)) (OF_UINT (D8 (D6 (D4 (D0 (D0
NIL)))))) IN
LET HTS = MULN RT.HOUR (OF_ UINT (D3 (D6 (D0 (D0 NIL))))) IN

TIME LIBRARY FUNCTIONALITY

Time Library is formally verified with an ample functionality:

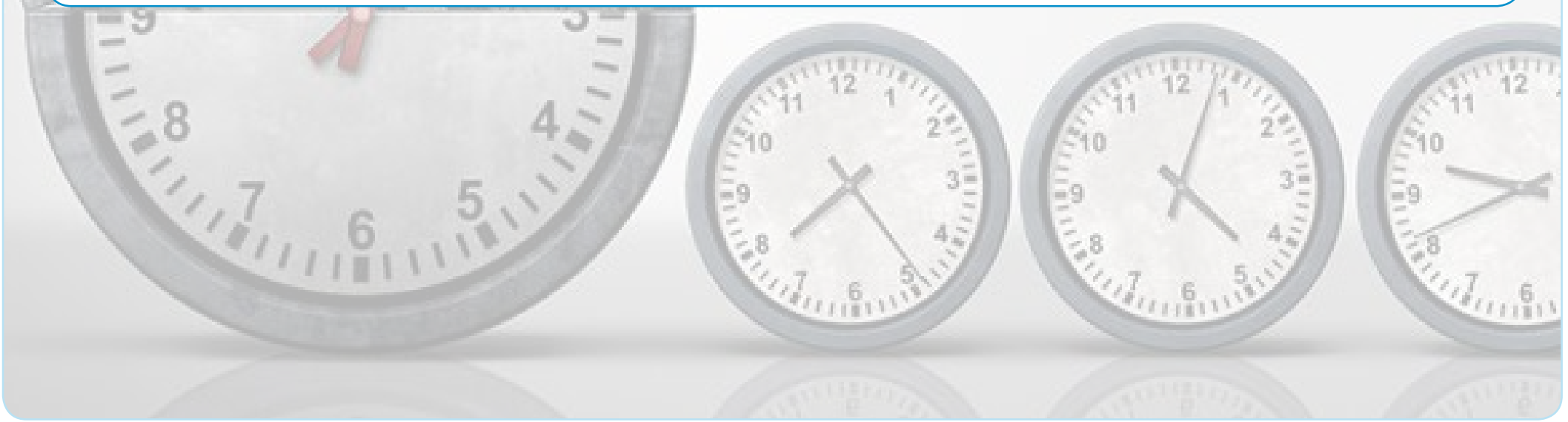
- Conversion functions (4);
- Shift functions (6);
- Add functions (6);
- Auxiliary functions (174).

TIME LIBRARY PROGRAMMING STATISTICS

Some statistics about the library and corresponding extraction:

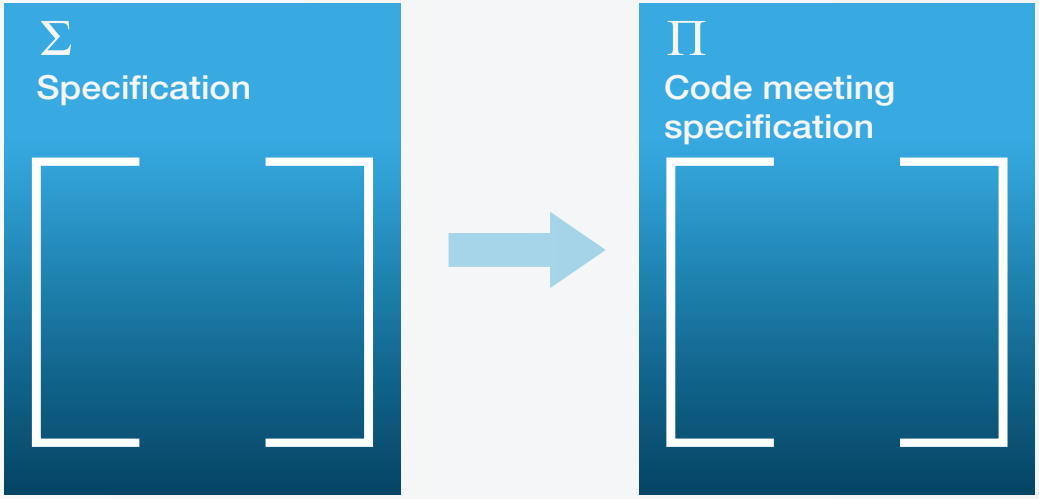
- Estimated programming hours 30.000 = 41 formal months;
- Nr of lines of Coq code and proofs= 4114 (4821 incl. extraction directives);
- Nr of lemmata/theorems = 190 (278 incl. extraction directives);
- Nr of lines of extracted code = 1363 (down from an earlier (±) 100 MB)

A particularly hard function to deal with was yoe_of_doe

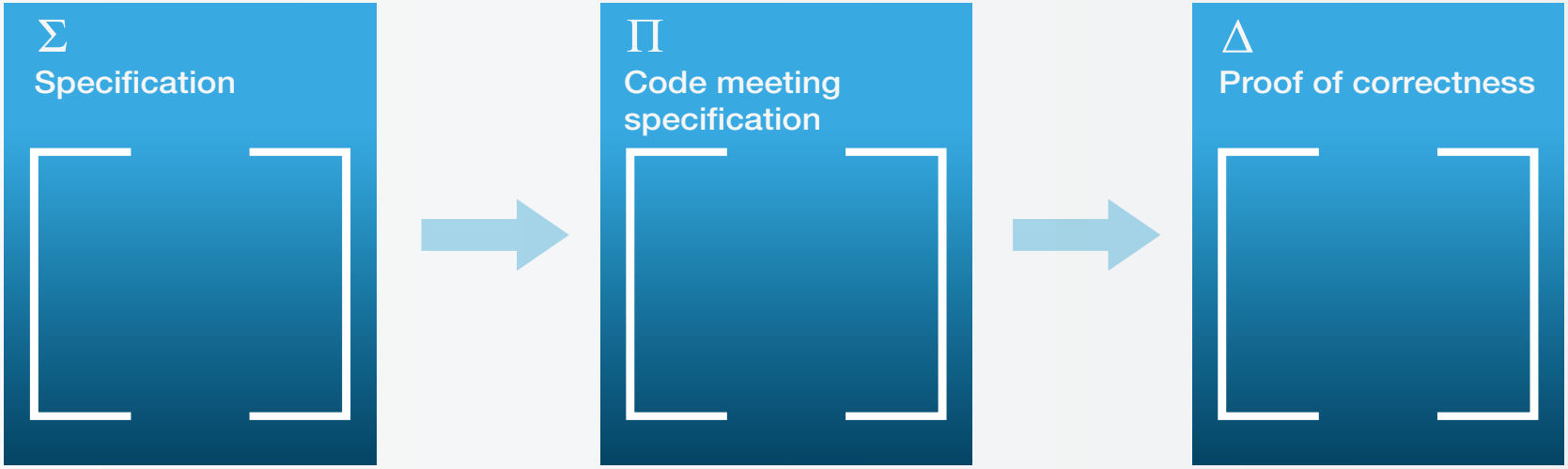


VERIFICATION OF FUNCTIONAL CODE

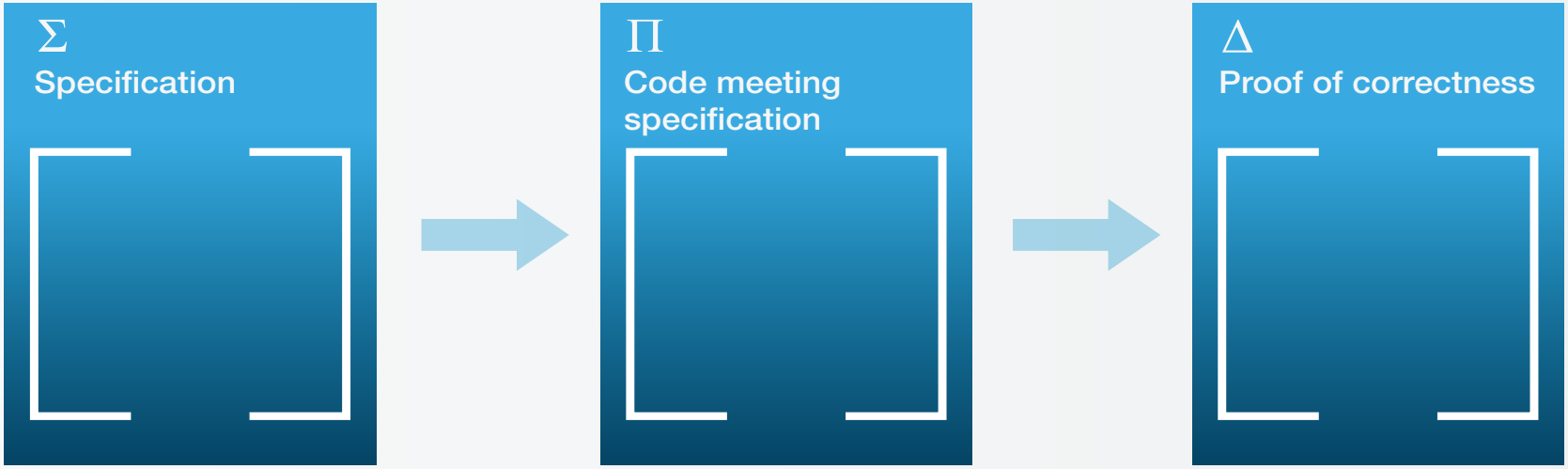
VERIFICATION OF FUNCTIONAL CODE



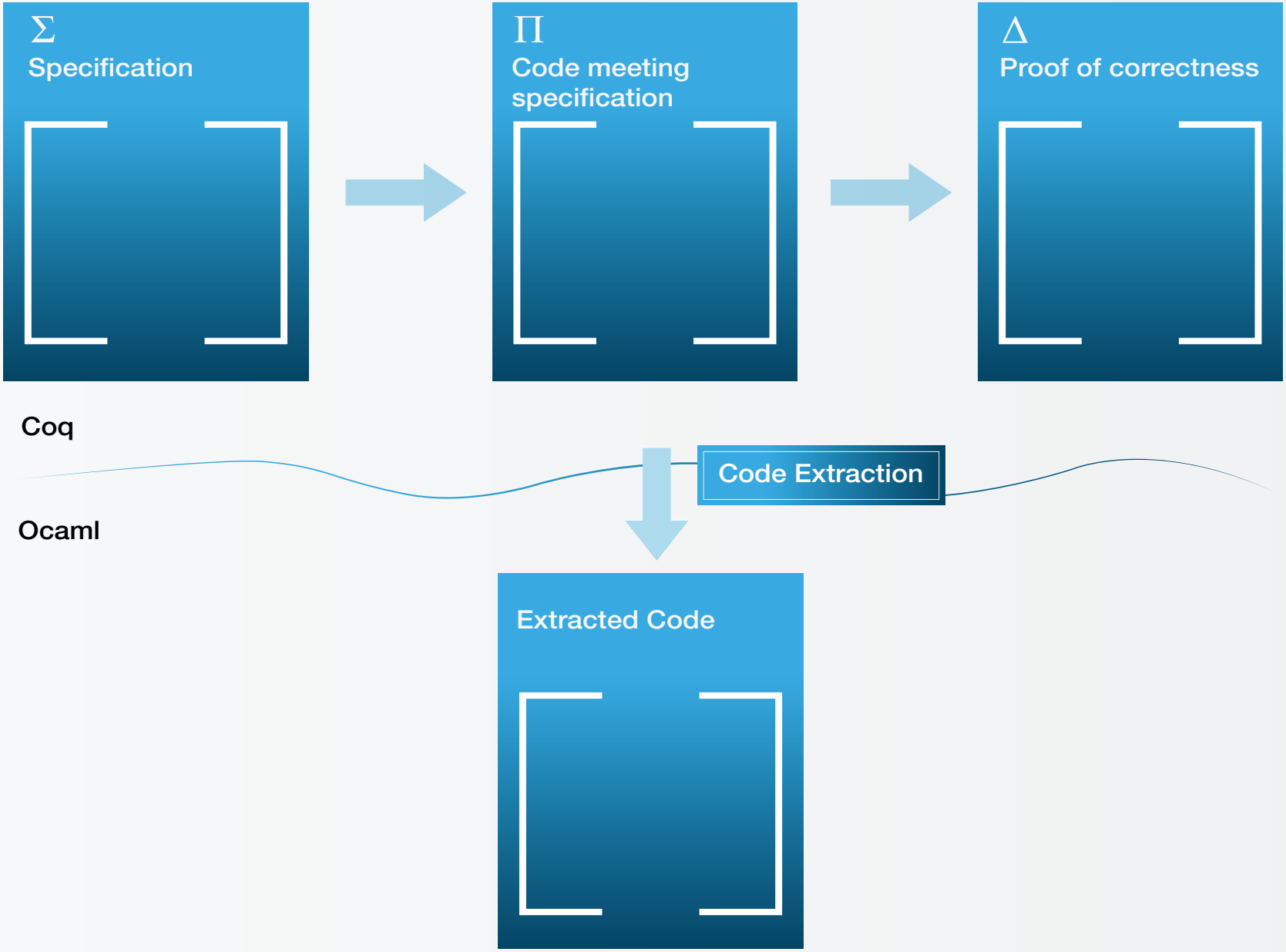
VERIFICATION OF FUNCTIONAL CODE



VERIFICATION OF FUNCTIONAL CODE



VERIFICATION OF FUNCTIONAL CODE




THANK YOU!