Explicit Abstraction Barrier for Autoactive Verification

Paul Patault

supervised by Jean-Christophe Filliâtre and Andrei Paskevich

January 2026 @ Dafny Workshop

université PARIS-SACLAY

LMF — Laboratoire Méthodes Formelles

Deductive verification 101

```
method Maximum(values: seq<int>) returns (max: int)
{
  max := values[0];
  for idx := 0 to |values| {
    if max < values[idx] {
      max := values[idx];
    }
  }
}
```

Deductive verification 101

```
method Maximum(values: seq<int>) returns (max: int)
  requires |values| > 0
  ensures  max in values
  ensures  forall i :: 0 ⩽ i < |values| ⟹ values[i] ⩽ max
{
  max := values[0];
  for idx := 0 to |values| {
    if max < values[idx] {
      max := values[idx];
    }
  }
}
```

Deductive verification 101

```
method Maximum(values: seq<int>) returns (max: int)
  requires |values| > 0
  ensures  max in values
  ensures  forall i :: 0 ⩽ i < |values| ⟹ values[i] ⩽ max
{ ... }
```

→ we **assume** preconditions

→ we **prove** postconditions

Deductive verification 101

```
method Maximum(values: seq<int>) returns (max: int)
  requires |values| > 0
  ensures  max in values
  ensures  forall i :: 0 ⩽ i < |values| ⟹ values[i] ⩽ max
{ ... }
```

→ we **assume** preconditions

→ we **prove** postconditions

implementation

---

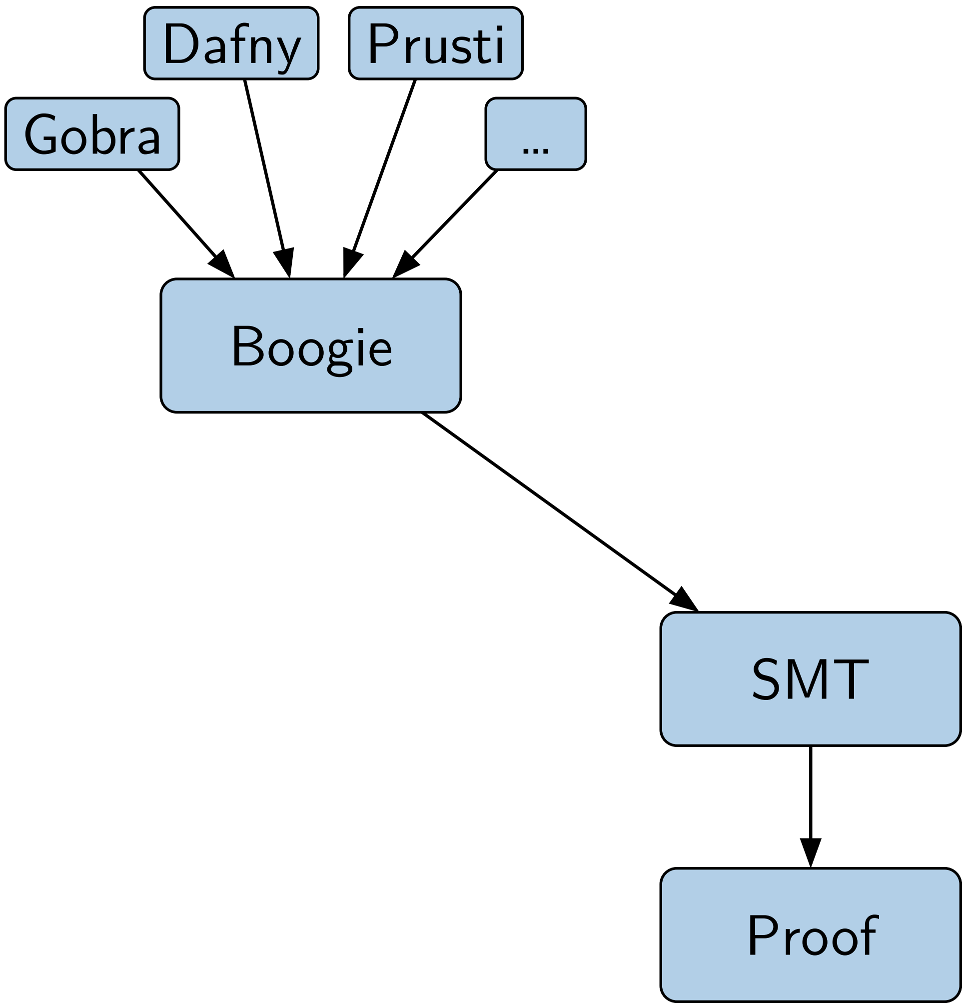
client

```
var m := Maximum(v);
...
```

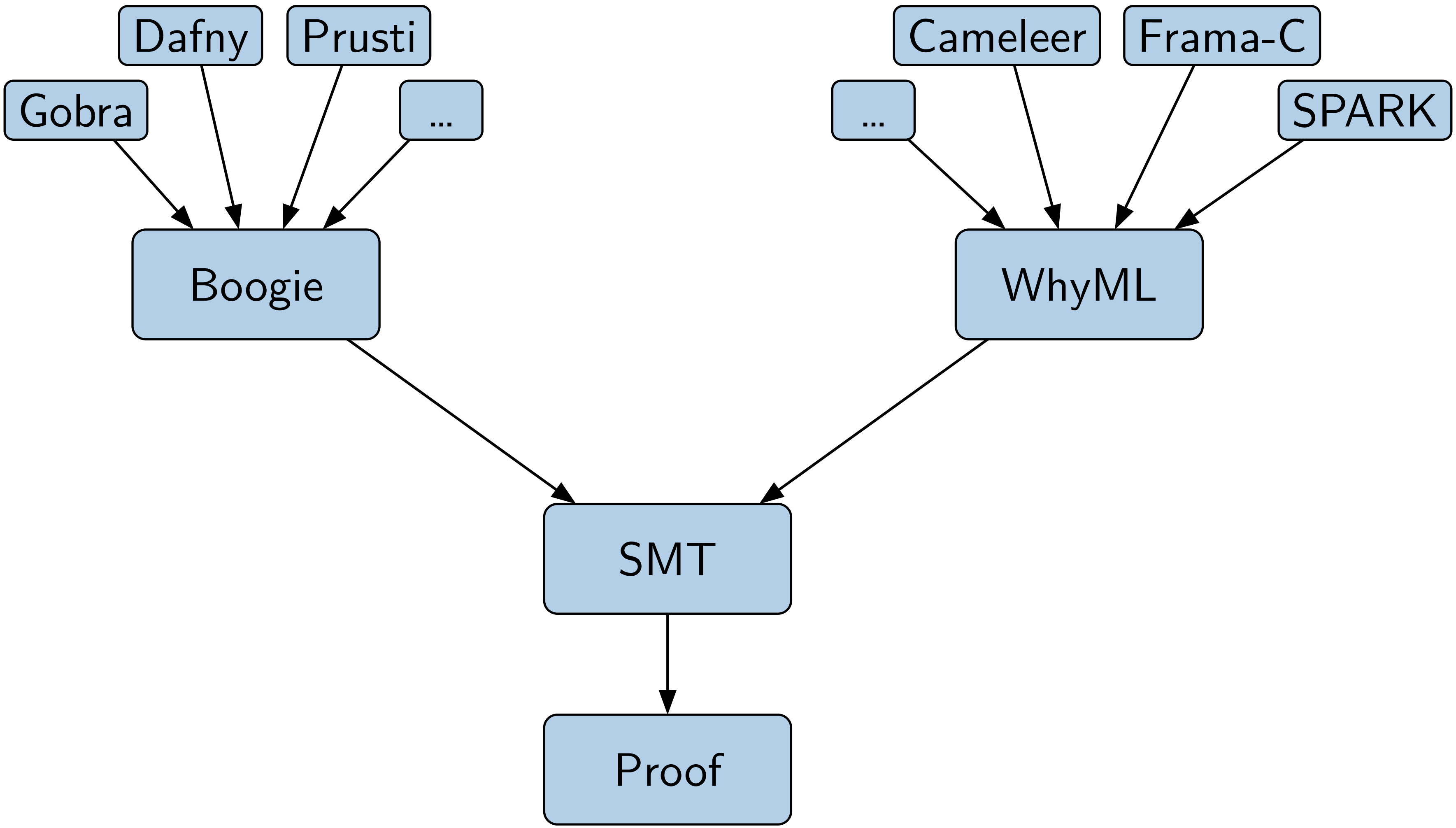
→ we **prove** preconditions

→ we **assume** postconditions

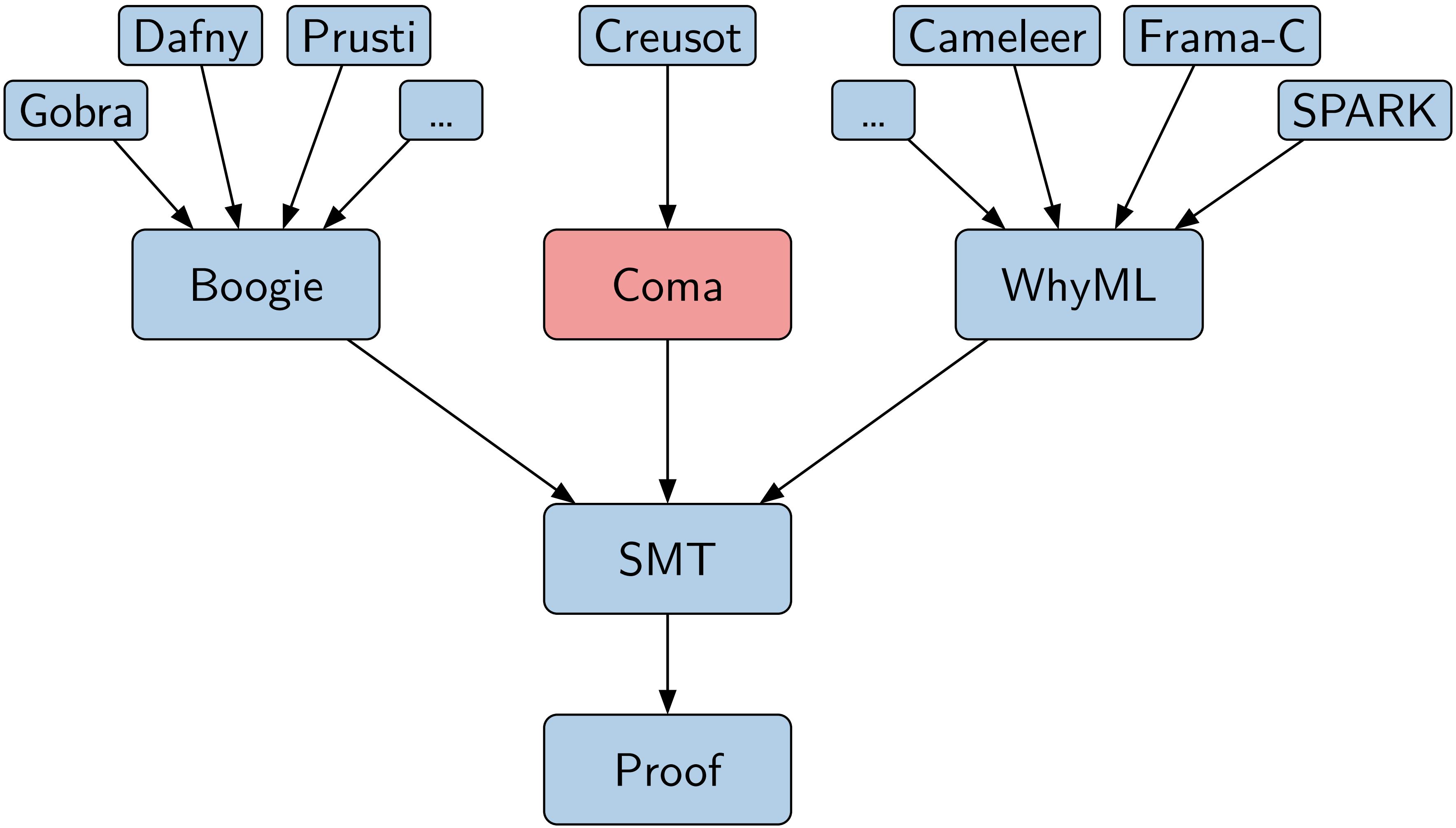
Intermediate Verification Languages (IVL)



Intermediate Verification Languages (IVL)



Intermediate Verification Languages (IVL)



Coma *[Paskevich, Patault, Filliâtre (ESOP 2025)]*

- "minimal"
  - function definition and application
  - logical assertions

- continuation-passing-style (CPS)
  - enable the encoding of many control structures
  - simpliflies *verification conditions* generation (VCgen)

- **explicit abstraction barrier**

Abstraction barrier

```
let rec fib (n: int) (out: int ⟶ ⊥): ⊥
= if n < 0 then fail () else
  if n < 2 then out n    else
  fib (n-2) (fun x ⟶
  fib (n-1) (fun y ⟶
  out (x+y) ))
```

Abstraction barrier

```
let rec fib (n: int) (out: int ⟶ ⊥): ⊥
= hide if n < 0 then fail () else
       if n < 2 then out n   else
       fib (n-2) (fun x ⟶
       fib (n-1) (fun y ⟶
       out (x+y) ))
```

Abstraction barrier

```
let rec fib (n: int) (out: int ⟶ ⊥): ⊥
= assert { n ⩾ 0 }
  hide if n < 0 then fail () else
       if n < 2 then out n else
       fib (n-2) (fun x ⟶
       fib (n-1) (fun y ⟶
       out (x+y) ))
```

Abstraction barrier

```
let rec fib (n: int) (out: int ⟶ ⊥): ⊥
= let out r = assert { r = F(n) } hide out r in
  assert { n ⩾ 0 }
  hide if n < 0 then fail () else
       if n < 2 then out n   else
       fib (n-2) (fun x ⟶
       fib (n-1) (fun y ⟶
       out (x+y) ))
```

Abstraction barrier

```
let rec fib (n: int) (out: int ⟶ ⊥): ⊥
= let out r = assert { r = F(n) } hide out r in
  assert { n ⩾ 0 }
  if n < 0 then fail () else
  hide if n < 2 then out n   else
       fib (n-2) (fun x ⟶
       fib (n-1) (fun y ⟶
       out (x+y) ))
```

Abstraction barrier

```
let rec fib (n: int) (out: int ⟶ ⊥): ⊥
= let out r = assert { r = F(n) } hide out r in
  if n < 0 then fail () else
  hide if n < 2 then out n   else
       fib (n-2) (fun x ⟶
       fib (n-1) (fun y ⟶
       out (x+y) ))
```

Abstraction barrier

```
let rec fib (n: int) (out: int ⟶ ⊥): ⊥
= let out r = assert { r = F(n) } hide out r in
  if n < 0 then fail () else
  if n < 2 then out n   else
  hide fib (n-2) (fun x ⟶
       fib (n-1) (fun y ⟶
       out (x+y) ))
```

Abstraction barrier

```
let rec fib (n: int) (out: int ⟶ ⊥): ⊥
= let out r = assert { r = F(n) } hide out r in
  if n < 0 then fail () else
  if n < 2 then out n   else
  hide fib (n-2) (fun x ⟶
       fib (n-1) (fun y ⟶
       out (x+y) ))
```

implementation

---

client

```
fib 42 (fun r ⟶ assert { r > 10^8 } halt ())
```

Abstraction barrier

```
let rec fib (n: int) (out: int ⟶ ⊥): ⊥
  = let out r = assert { r = F(n) } hide out r in
    if n < 0 then fail () else
    if n < 2 then out n   else
    hide fib (n-2) (fun x ⟶
         fib (n-1) (fun y ⟶
         out (x+y) ))
```

implementation

VC of the client

```
fib 42 (fun r ⟶ assert { r > 10^8 } halt ())
```

```
(42 < 0 ⟶ false) ∧
(0 ⩽ 42 < 2 ⟶ 42 = F(42)) ∧
(∀r. r = F(42) ⟶ r > 10^8)
```

Abstraction barrier

```
let rec fib (n: int) (out: int ⟶ ⊥): ⊥
= let out r = assert { r = F(n) } hide out r in
  if n < 0 then fail () else
  if n < 2 then out n   else
  hide fib (n-2) (fun x ⟶
       fib (n-1) (fun y ⟶
       out (x+y) ))
```

implementation

VC of the definition

```
∀n. not n < 0 ⟶ not n < 2 ⟶
  (n-2 < 0 ⟶ false) ∧
  (0 ⩽ n-2 < 2 ⟶ n-2 = F(n-2)) ∧
  (2 ⩽ n-2 ⟶ ∀x. x = F(n-2) ⟶
   (n-1 < 0 ⟶ false) ∧
   (0 ⩽ n-1 < 2 ⟶ n-1 = F(n-1)) ∧
   (2 ⩽ n-1 ⟶ ∀y. y = F(n-1) ⟶
    x + y = F(n)))
```

Modal VCgen

$\mathcal{C}$: verification at call site

$\mathcal{D}$: verification at definition site

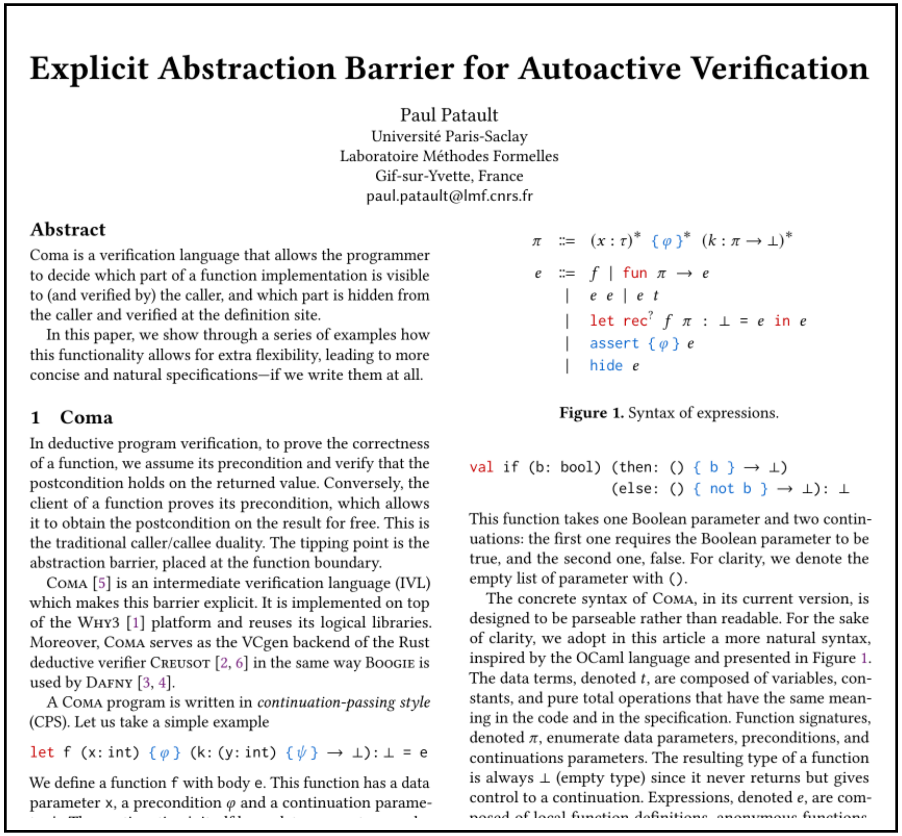
$$\mathcal{C}(\texttt{assert \{ ϕ \} e}) \triangleq \varphi \land \mathcal{C}(\texttt{e})$$

$$\mathcal{D}(\texttt{assert \{ ϕ \} e}) \triangleq \varphi \longrightarrow \mathcal{D}(\texttt{e})$$

$$\mathcal{C}(\texttt{hide e}) \triangleq \top$$

$$\mathcal{D}(\texttt{hide e}) \triangleq \mathcal{C}(\texttt{e}) \land \mathcal{D}(\texttt{e})$$

To go further



Explicit Abstraction Barrier for Autoactive Verification

Paul Patault
Université Paris-Saclay
Laboratoire Méthodes Formelles
Gif-sur-Yvette, France
paul.patault@lmf.cnrs.fr

Abstract

Coma is a verification language that allows the programmer to decide which part of a function implementation is visible to (and verified by) the caller, and which part is hidden from the caller and verified at the definition site.

In this paper, we show through a series of examples how this functionality allows for extra flexibility, leading to more concise and natural specifications—if we write them at all.

1 Coma

In deductive program verification, to prove the correctness of a function, we assume its precondition and verify that the postcondition holds on the returned value. Conversely, the client of a function proves its precondition, which allows it to obtain the postcondition on the result for free. This is the traditional caller/callee duality. The tipping point is the abstraction barrier, placed at the function boundary.

Coma [5] is an intermediate verification language (IVL) which makes this barrier explicit. It is implemented on top of the Why3 [1] platform and reuses its logical libraries. Moreover, Coma serves as the VCgen backend of the Rust deductive verifier Creusot [2, 6] in the same way Boogie is used by Dafny [3, 4].

A Coma program is written in *continuation-passing style* (CPS). Let us take a simple example

```
let f (x: int) { φ } (k: (y: int) { ψ } → ⊥): ⊥ = e
```

We define a function f with body e. This function has a data parameter x, a precondition φ and a continuation parame-

$$\pi \ ::= \ (x:\tau)^* \ \{\varphi\}^* \ (k:\pi\to\bot)^*$$

$$\begin{array}{rcl} e & ::= & f \mid \texttt{fun}\ \pi \to e \\ & | & e\ e \mid e\ t \\ & | & \texttt{let rec}^?\ f\ \pi : \bot = e\ \texttt{in}\ e \\ & | & \texttt{assert}\ \{\varphi\}\ e \\ & | & \texttt{hide}\ e \end{array}$$

Figure 1. Syntax of expressions.

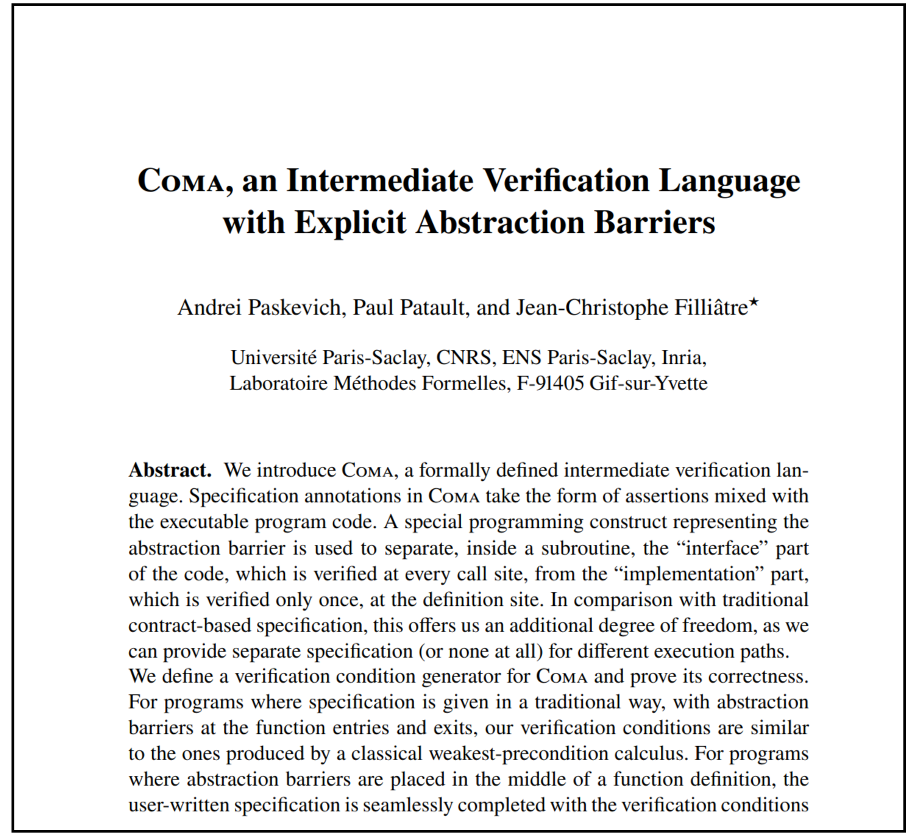
```
val if (b: bool) (then: () { b } → ⊥)
                 (else: () { not b } → ⊥): ⊥
```

This function takes one Boolean parameter and two continuations: the first one requires the Boolean parameter to be true, and the second one, false. For clarity, we denote the empty list of parameter with ().

The concrete syntax of Coma, in its current version, is designed to be parseable rather than readable. For the sake of clarity, we adopt in this article a more natural syntax, inspired by the OCaml language and presented in Figure 1. The data terms, denoted t, are composed of variables, constants, and pure total operations that have the same meaning in the code and in the specification. Function signatures, denoted π, enumerate data parameters, preconditions, and continuations parameters. The resulting type of a function is always $\bot$ (empty type) since it never returns but gives control to a continuation. Expressions, denoted e, are com-

Dafny 2026

→ more *examples*



Coma, an Intermediate Verification Language with Explicit Abstraction Barriers

Andrei Paskevich, Paul Patault, and Jean-Christophe Filliâtre*

Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria,
Laboratoire Méthodes Formelles, F-91405 Gif-sur-Yvette

Abstract. We introduce Coma, a formally defined intermediate verification language. Specification annotations in Coma take the form of assertions mixed with the executable program code. A special programming construct representing the abstraction barrier is used to separate, inside a subroutine, the "interface" part of the code, which is verified at every call site, from the "implementation" part, which is verified only once, at the definition site. In comparison with traditional contract-based specification, this offers us an additional degree of freedom, as we can provide separate specification (or none at all) for different execution paths. We define a verification condition generator for Coma and prove its correctness. For programs where specification is given in a traditional way, with abstraction barriers at the function entries and exits, our verification conditions are similar to the ones produced by a classical weakest-precondition calculus. For programs where abstraction barriers are placed in the middle of a function definition, the user-written specification is seamlessly completed with the verification conditions

ESOP 2025

→ more *details*

Application: Creusot *[Denis, Jourdan, Marché, Golfouse]*

- Creusot: deductive verifier for Rust
- uses Coma IVL
- barrier allows specification inference of closures

```
let o = Some(42);

let a = o.map(
  #[requires(x@ + 1 ⩽ i32::MAX@)]
  #[ensures(result@ == x@ + 1)]
  |x| x + 1,
);
let b = o.map(
  #[requires(2 * x@ ⩾ i32::MIN@)]
  #[requires(2 * x@ ⩽ i32::MAX@)]
  #[ensures(result.0@ == 2 * x@)]
  #[ensures(result.1  == x)]
  |x| (2 * x, x),
);
```

Application: Creusot *[Denis, Jourdan, Marché, Golfouse]*

- Creusot: deductive verifier for Rust
- uses Coma IVL
- barrier allows specification inference of closures

```
let o = Some(42);

let a = o.map(
  #[requires(x@ + 1 ⩽ i32::MAX@)]
  #[ensures(result@ == x@ + 1)]
  |x| x + 1,
);
let b = o.map(
  #[requires(2 * x@ ⩾ i32::MIN@)]
  #[requires(2 * x@ ⩽ i32::MAX@)]
  #[ensures(result.0@ == 2 * x@)]
  #[ensures(result.1  == x)]
  |x| (2 * x, x),
);
```

```
let o = Some(42);

let a = o.map(|x| x + 1);

let b = o.map(|x| (2 * x, x));
```

Take away

Coma

- new IVL
- explicit abstraction barrier
- used by Creusot

ESOP 2025

Dafny 2026