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 parameter `k`. The continuation `k` itself has a data parameter `y` and a precondition ψ. Since Coma programs are in CPS, functions can only return to the caller by passing control to a continuation provided by the caller. In the function body `e`, every time we want to return a value `v`, we call `k v`. From a verification perspective, the precondition ψ of `k` must be proved when the continuation is called—before getting out of `f`. Therefore, in Coma, we do not need a dedicated construct for postconditions: they are the preconditions of continuations. Consequently, in the rest of this text, we use the term postcondition to refer to the preconditions of continuations.

Continuation-passing style allows Coma to have only a few language constructions. Indeed, this form is expressive enough to encode standard control structures. For example, we can easily provide if-then-else statements with the following primitive

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

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 composed of local function definitions, anonymous functions, and function applications. On top of that, an expression can be prefixed by an assertion or hidden below an explicit abstraction barrier `hide`.

In this article, we show how making the abstraction barrier explicit can be useful in program specification and verification. After a series of short examples in Section 2, we develop a complete Sudoku solver in Section 3. The complete Coma implementations can be found in the archive hosted at https://doi.org/10.5281/zenodo.17279467.

2 Explicit abstraction barrier

In this section, we present several ways to use the explicit abstraction barrier. From the VC generation point of view, the barrier syntactically separates what is verified at the definition site and what is verified at the call site. It means

that the VC of the part of code above the barrier is inlined in the VC of the caller. Conversely, the code below the barrier is hidden from the caller and proved on the callee side.

In the listings below, we use the traditional syntax for `if` statements and omit the types of data parameters when they can be inferred.

Recursive function. Let us take the traditional recursive computation of the Fibonacci sequence as our first example. As the function is recursive, we must put a barrier inside its definition, hiding all recursive calls. Otherwise, VC inlining on the caller side would be non-terminating.

```
let rec fib (n: int)
    (out: (r: int) { r = F(n) } → ⊥): ⊥
= 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) ))
```

We define a function `fib` that takes two parameters: `n` is index of the Fibonacci number to compute, and `out` is the continuation. The continuation itself expects an integer parameter `r`, which represents the result of the computation. The precondition of `out` mentions F, the mathematical definition of the Fibonacci function, assumed to be given.

The body of this function is a sequence of tests. If the parameter `n` is negative, then we call the primitive `fail` whose precondition is `false`. Otherwise, we can compute Fibonacci as usual with one base case and two recursive calls.

To illustrate the caller-side VC, consider the expression

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

where the VC of the `halt` primitive is `true`. The verification condition of this call is

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

The first part of the conjunction corresponds to the first test. Then, the first call to `out` (above the barrier in `fib`) is visible by the caller of `fib`. Therefore, the precondition of `out` (namely, `r = F(n)`) must be instantiated and proved by the caller. Finally, for the last part of the conjunction, what is under the barrier in `fib` is invisible to the caller and the latter recovers the postcondition of `fib` (the precondition of its continuation) as a hypothesis to prove what comes next.

The VC of `fib`'s definition is as follows:

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

The correctness of code above the barrier is assumed and what comes after must be verified. It means that the calls to `fail` and the first call to `out` with the initial `n` are not taken into account here, because they are verified by the caller. We merely recover the preconditions of the two `else` branches on the path leading to the barrier. The first block of the subsequent proof obligations corresponds to the VC of the first recursive call on `(n-2)`. Then, similar proof obligations are generated by the second recursive call on `(n-1)`. Finally, the last line of the VC corresponds to the call of `out`.

No barrier. Another possibility is not placing the barrier at all. When the function is not recursive, its code can always be inlined at the call sites, and there is nothing to be verified at the definition site.

This can be useful for simple functions whose contract, if written explicitly, would be a reformulation of their code. For example, consider Euclidian division:

```
let div (x y: int) (dbz: () → ⊥)
                   (k: (r: int) → ⊥): ⊥
= if y = 0 then dbz () else k (x / y)
```

We can hide the body of `div` behind a barrier and provide preconditions for the continuation parameters:

```
(dbz: () { y = 0 } → ⊥)
(k: (r: int) { y ≠ 0 ∧ r = x / y } → ⊥)
```

but this would add nothing to what is already in the implementation. It is simpler—and shorter—to leave this function as is, completely transparent.

Consider the following caller of `div`:

```
div x y fail
  (fun (r: int) → assert { r * y ⩽ x } halt ())
```

Its full verification condition comes from `div`'s body, inlined:

```
if y = 0 then false else (x / y) * y ⩽ x
```

Here, `false` is the precondition of `fail` and the `else` branch corresponds to the non-zero case.

Higher-order iteration. We show a function `iteri` that iterates over an interval and calls code, given as a parameter, on each integer between the two endpoints. The function has the signature

```
let iteri (lo hi: int) (seen: int → bool)
    (body: (i: int)
        { lo ⩽ i < hi }
        { ∀j. lo ⩽ j < i → seen j }
        (continue: () { seen i } → ⊥) → ⊥)
    (out: () → ⊥): ⊥
= ...
```

To be more precise, the loop goes through `[lo..hi[` calling the continuation body on each integer in between. We give two preconditions to body: (1) its argument `i` must be a valid index of the interval and (2) each integer processed before has to validate the predicate `seen`. The continuation `continue` represents what to do after each loop iteration. Therefore, its precondition must be an assertion that holds after one loop unrolling: the processed integer now belongs to the integers seen. The definition of `iteri` uses an inner recursive function `loop` to iterate from `lo` to `hi`.

```
let rec loop (i: int): ⊥
= assert { lo ⩽ i }
  assert { ∀j. lo ⩽ j < i → seen j }
  if i < hi
  then hide body i (fun () → loop (i+1))
  else out () in
loop lo
```

Afterwards, we give control back to the main continuation `out`. The continuation `continue` of body is used to store what is done after one body execution: call back the loop with `i+1`.

Notice that the principal continuation `out` does not have a precondition. We could have written the contract of `iteri` with

```
(out: () { ∀j. lo ⩽ j < hi → seen j } → ⊥)
```

which is a valid and provable specification, but slightly redundant.

There are two paths that lead to the call of `out`. First, if the interval is empty: we have nothing that we can learn. Otherwise, the interval is non-empty: `out` will be called after several executions of body, which will give the caller of `iteri` sufficient information.

Remark that `iteri` does not have any barrier in its definition. The entire VC of the definition of `loop` together with its initial call will be inlined in the caller's VC. This means that we have nothing to prove at the definition site.

3 Application: Sudoku

Given a partial and non-contradictory Sudoku grid, we want to find whether it can be completed up to a solution. In our formalization, we represent a Sudoku grid as a sequence of 81 integers between 0 and 9, where 0 denotes an empty cell:

```
type sudoku = seq int
```

Two predicates are used to check the validity of grid indices and filled cell values:

```
predicate bounds (c: int) = 0 ⩽ c < 81
predicate filled (v: int) = 1 ⩽ v ⩽ 9
```

Now, we can define the predicates that state the validity of a cell in a grid.

```
predicate same_zone (c1 c2: int) =
  div c1 9 = div c2 9 ∨ mod c1 9 = mod c2 9 ∨
  (div (div c1 9) 3 = div (div c2 9) 3 ∧
   div (mod c1 9) 3 = div (mod c2 9) 3)

predicate compatible (g: sudoku) (c1 c2: int) =
  c1 ≠ c2 ∧ same_zone c1 c2 → g[c1] ≠ g[c2]

predicate correct (g: sudoku) (c: int) =
  bounds c ∧ filled g[c] ∧
  ∀c'. bounds c' → compatible g c c'
```

Now, we can define a function check that verifies if the value in a given cell is consistent with the rest of the grid.

```
let check (g: sudoku) (c: int)
      (ok: () { correct g c } → ⊥)
      (ko: () { not correct g c } → ⊥): ⊥
= iteri 0 81
    (fun (i:int) → bounds i ∧ compatible g c i)
    (fun (i:int) (continue: () → ⊥) →
       if compatible g c i then continue ()
                           else ko ())
    ok
```

This function iterates through the cells of the grid, verifying that they do not clash with the cell `c`. It calls its continuation `ko` if it finds a conflict in the grid. Just like the `iteri` function, `check` does not contain a barrier and so is verified by its callers rather than at the definition site.

To verify a Sudoku solver with a backtracting procedure, we need an invariant and correctness+completeness criteria. The invariant of a recursive function is its precondition, and correctness+completeness, its postcondition.

The following two predicates provide the invariant. A well-formed grid is of length 81, filled with integers between 0 and 9, and none of its non-empty cells conflict with another.

```
predicate wf (g: sudoku) =
  length g = 81 ∧
  (∀i. bounds i → 0 ⩽ g[i] ⩽ 9) ∧
  (∀i. bounds i → filled g[i] → correct g i)
```

We also need a predicate to state the grid is filled up to a given index:

```
predicate filled_upto (g: sudoku) (c: int) =
  ∀i. 0 ⩽ i < c → filled g[i]
```

Now, we can define what constitutes a successful search. The `extends` predicates specifies if one grid is an extension of another, and the `solution` predicate says that all the cells in the grid are filled and compatible with each other.

```
predicate extends (g1 g2: sudoku) =
  ∀i. bounds i → filled g1[i] → g1[i] = g2[i]
```

```
predicate solution (g: sudoku) =
  ∀i. bounds i → correct g i
```

The completeness of the solver can be stated with the predicate `impossible` meaning that no extension of the initial grid is a solution.

```
predicate impossible (g: sudoku) =
  ∀g'. extends g g' → not solution g'
```

Now, we can write the main solver function:

```
let rec solve (g: sudoku) (c: int)
    { wf g } { 0 ⩽ c ⩽ 81 } { filled_upto g c }
    (ok: (g': sudoku)
       { extends g g' } { solution g' } → ⊥)
    (ko: () { impossible g } → ⊥): ⊥
= if c = 81    then ok g else
  if g[c] ≠ 0  then solve g (c+1) ok ko else
  iteri 1 10 (fun v → impossible g[c←v])
    (fun (v: int) (next: () → ⊥) →
       let g = g[c←v] in
       check g c
         (fun () → solve g (c+1) ok next)
         next)
    ko
```

We want to build, if exists, one solution that extends the initial grid. The first argument g is the Sudoku grid to solve. The second argument c represents the cell index that must be filled during this call. This index must be between 0 and 81, and g must be filled up to c. This function has two continuation parameters: ok for the success, and ko for the error. The last call to this function, when c is 81, can call the continuation ok, since the grid is full and well-formed, and thus is a solution.

As with postconditions, the presence of preconditions implicitly adds a barrier at the entry point of the function. Consequently, `solve`'s definition contains a barrier and must therefore be proved. Its VC is rather large, but it is easily discharged by SMT solvers.

Last, the main function calls `solve` on the first cell.

```
let solve_main (g: sudoku) { wf g }
    (ok: (g': sudoku)
       { extends g g' } { solution g' } → ⊥)
    (ko: () { impossible g } → ⊥): ⊥
= solve g 0 ok ko
```

It has the trivial following verification condition:

```
∀g. wf g →
  (wf g ∧ (0 ⩽ 0 ⩽ 81) ∧ filled_upto g 0) ∧
  (∀g'. (extends g g' ∧ solution g') →
        (extends g g' ∧ solution g')) ∧
  (impossible g → impossible g)
```

where the parts of the VC written in bold (resp. non-bold) indicate the assumptions and goals coming from `solve_main` (resp. `solve`).

This concludes our Sudoku solver verification that is totally verified in less than a second.

4 Comparison

The same program can easily be written and verified in many comparable IVLs. Let us take WhyML [1], a well-known IVL with a fairly comprehensive surface language including loops, algebraic data types, functions, exceptions, and specification material such as assertions, preconditions and postconditions.

The first essential difference is that the function `iteri` simply cannot be written in WhyML, due to its higher-order aspect. However, we can locally simulate it with a for loop. For example, we can define check within the same logical context:

```
let check (g: sudoku) (c: int) : bool
  requires { bounds c }
  requires { filled g[c] }
  ensures  { result ↔ correct g c }
= for i = 0 to 80 do
    invariant { ∀j. 0 ⩽ j < i →
                    not clash g c j }
    if clash g c i then return false
  done;
  return true
```

There are two differences. Firstly, the construct `for` is part of the language WhyML where the function `iteri` is defined in Coma. Secondly, the WhyML definition requires a specification and a proof, whereas in Coma its VC can be discharged to the caller. In WhyML, the function cannot obtain information from the caller's context, and we must provide it as preconditions. In total, for the proof of the Sudoku solver, WhyML requires 11 specification units instead of 9 in Coma.

We can conclude from these observations, brief but enlightening, that Coma seems just as expressive as WhyML while being more versatile.

5 Conclusion

We presented Coma (Section 1) and its explicit abstraction barrier. We saw how this construct can simplify programs and proofs by allowing, for example, some functions to be inlined totally or partially (Section 2). Then, we applied these principles to verify a Sudoku solver (Section 3). Finally, we compared our solution to WhyML, a state-of-the-art IVL, which can prove correct the same functions, but with a weaker degree of freedom (Section 4).

Acknowledgements

This research was supported, in part, by the ANR project ANR-22-CE48-0013 “GOSPEL” and, in part, by the Décysif project funded by the Île-de-France region and by the French government in the context of “Plan France 2030”.

References

[1] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. 2011. *The Why3 platform*. https://www.why3.org/.

[2] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: a Foundry for the Deductive Verication of Rust Programs. In *International Conference on Formal Engineering Methods - ICFEM*. Springer, Madrid, Spain.

[3] K Rustan M Leino. 2008. This is Boogie 2. *KRML Manuscript* 178 (2008).

[4] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*. Springer, Springer, 348–370.

[5] Andrei Paskevich, Paul Patault, and Jean-Christophe Filliâtre. 2025. Coma, an Intermediate Verification Language with Explicit Abstraction Barriers. In *Programming Languages and Systems*, Viktor Vafeiadis (Ed.). Springer Nature Switzerland, Cham, 175–201.

[6] Paul Patault, Arnaud Golfouse, and Xavier Denis. 2025. Remonter les barrières pour ouvrir une clôture. In *JFLA 2025 - 36es Journées Francophones des Langages Applicatifs*. Roiffé, France. https://inria.hal.science/hal-04859517