Hashing and boolean decision trees in Rocq

Élise Souche (supervised by David Monniaux)
Verimag, Université Grenobles-Alpes
elise.souche@ens-lyon.fr
2025

Abstract

Proof assistants allow developing programs that are correct by construction and extracting them into a classical programming language. Such programs suffer from the limitations of the proof assistant: in particular, they need to be purely functional. However, there are many reasons to desire using impure features from the target language. We demonstrate a method to reason about these features and develop a certified SAT solver using them.

Contents

1 Background 2
2 The impure monad 3
  2.1 A monad 3
  2.2 Exiting the monad 4
  2.3 Models 5
  2.4 Physical equality 5
3 A boolean decision diagram library 6
  3.1 Satisfiability testing and ordering 6
  3.2 Hash consing 7
  3.3 Boolean operations 8
4 Results 9
  4.1 Correctness 9
  4.2 Ease of use 9
  4.3 Performance 10
5 Future work 12
Conclusion 13
Bibliography 13
Appendix 1 – Internship context 14

1 Background

Proof assistants such as Rocq or Agda are based on dependent type theories and implement a dependently typed lambda-calculus. In this language, it is possible to express both programs and proofs thanks to the Curry-Howard interpretation. The program can be extracted to a target language, usually a functional language like OCaml or Haskell. Thus are obtained certified programs that are proven correct, and can be compiled into decently performing machine code. Using this methodology, major works have been developped, such as CompCert [3], a certified C compiler.

There is however a problem with this approach: the proof assistant's language is a purely functional language. But there are many very valid reasons to want to use impure features of our target language! One of them is performance: maybe there is a nice performant algorithm for the problem we want to solve, but it is impure and cannot be expressed in the proof assistant. We want our solution to integrate nicely into the rest of the proof assistant (in our case, Rocq), not requiring a completely different methodology. And we want the program to extract to nice, readable, and performant code in our target language (in our case, OCaml).

A way of doing this is to have the extracted program depend on an impure oracle implemented directly in the target language. This oracle can perform arbitrary computations. On the proof assistant side, we see the oracle as a black box. We can choose whether or not to trust the oracle. If we do not trust the oracle, we can include checks guarding every call to the oracle in the certified code, falling back to a slow but certified implementation if the checks were to fail. This way, we do not compromise the formally-verified correctness of our program. This approach is notably used by the Chamois CompCert version [4].

We could simply add the OCaml functions we want to use as axioms, tell Rocq how to extract them, and use them as is. For instance, if we want an OCaml function of type `unit -> bool` that returns `true` on the first call, then on the next call `false`, then `true`, etc., we can do:

```
Rocq
Axiom flipper: unit -> bool.
Extract Constant flipper =>
    "(let cell = ref false in fun () -> cell := not !cell; !cell)".
```

Well, this is indeed nice, readable, and performant. It is also completely unsound:

```
Definition this_is_true := xorb (flipper tt) (flipper tt).
Lemma this_is_true_false: this_is_true = false.
    (* this works because Rocq thinks that flipper tt = flipper tt *)
    apply Bool.xorb_nilpotent.
Qed.
```

So we can prove that `this_is_true` is equal to `false`. But actually, once extracted to OCaml, it will be equal to `true`! We need something more sophisticated.

2 The impure monad

2.1 A monad

A simple solution is to wrap our impure computations in an impure monad [1] in the following fashion:

```
(** The type of impure computations *)
Axiom imp: Type -> Type.
Notation "??" := imp (at level 70, right associativity).
(** Standard monad operator: lift pure computation *)
Axiom ret: ∀ {A}, A -> ??A.
(** Standard monad operator: bind two impure computations *)
Axiom bind: ∀ {A B}, ??A -> (A -> ??B) -> ??B.
```

This way, `??A` denotes a computation that may non-deterministically return an object of type `A`. These operations can be extracted to no-ops in OCaml:

```
Extract Constant imp "" => "".
Extract Constant ret "" => "".
Extraction Inline imp ret.
Extract Inlined Constant bind => "(|>)".
```

Then, our `flipper` OCaml oracle can be expressed in the following way:

```
Axiom flipper: unit -> ??bool.
Extract Constant flipper =>
    "(let cell = ref true in fun () -> cell := !cell; !cell)".
```

and we are no longer able to do the proof above.

To be able to manipulate values in the impure monad, we introduce a few primitives and lemmas, which should be self-explanatory.

Rocq

```
(** The may-return relation. *)
(** k~~>a means that computation k may evaluate to a. *)
Axiom mayRet: ∀ {A:Type}, ??A -> A -> Prop.
Notation "k ~~> a" := (mayRet k a) (at level 40, no associativity).
(** Exiting the monad on termination *)
Axiom has_returned: ∀ {A}, ??A -> bool.
(** we can attach a proof that the computation gives the result *)
Axiom mk_annot: ∀ {A} (k: ?? A), ??{ a: A | k ~~> a }.

(** Axioms of monad operators *)
Axiom mayRet_ret: ∀ A (a b:A), (ret a ~~> b) -> a = b.
Axiom mayRet_bind: ∀ A B k1 k2 (b:B),
    (bind k1 k2) ~~> b ->
    ∃ a:A, k1 ~~> a /\ k2 a ~~> b.
Axiom has_returned_correct: ∀ A (k: t A),
    has_returned k = true -> ∃ r, k ~~> r.
```

2.2 Exiting the monad

This allows expressing impure computations in a safe way, and it does give nice, readable and performant code. However, we have not achieved our goal of having something that integrates nicely with the proof assistant. The problem is the same that always arises when using monads: once we enter the monad, we have no way of leaving. This is too restrictive. Often, algorithms using impure, non-deterministic features are actually overall observationally deterministic. An example of this is a SAT solver (which was the motivating case for this work): no matter how we do all the computations, the final result of whether or not a formula is satisfiable is deterministic.

Therefore, we want to introduce a new primitive `det_coerce` to exit the impure monad. We want to be able to coerce computations that are actually deterministic out of the monad. But how to formalize "actually deterministic"? A natural first approach would be to define it like so:

Rocq

```
Axiom det_coerce: ∀ {A} (k: ??A)
  (DET: ∀ a1 a2, k ~~> a1 -> k ~~> a2 -> a1 = a2),
  A.
Axiom det_coerce_correct: ∀ {A} (k: ??A) DET,
  k ~~> det_coerce k DET.
```

We allow exiting the non-deterministic monad if the computation always yields the same result. Unfortunately, this is unsound because it allows coercing

non-terminating computations: a `k: ??False` trivially verifies the condition `DET`. Thus we rather choose this alternative version:

```
⚙ Rocq
(** Exit the monad for observationally deterministic computations *)
Axiom det_coerce:
  ∀ {A} (k: ??A) (v: unit -> A) (DET: ∀ r, k ~~> r -> r = v tt), A.
Axiom det_coerce_correct:
  ∀ A (k: ??A) v (DET: ∀ r, k ~~> r -> r = v tt),
  (det_coerce k v DET) = (v tt).
```

The idea is that we can escape the monad if we have a pure, deterministic program such that our computation always gives the same result as that pure program. This forces us to show that the final type is inhabited and avoids the absurdity above. The deterministic program can be as inefficient as we want, because it will actually never be evaluated.

2.3 Models

There are many different possible models, that is, actual definitions for the axioms, for the monad. They show that these axioms do not add anything new to Rocq's logic. Examples of such models are:

- `??A := A`: we actually only allow pure computations
- `??A := A -> Prop`: we identify an impure computations with the set of values it can return
- `??A := State -> A * State`: the classical state monad

The implementation of the axioms as expression of OCaml oracles can also be seen as such a model, albeit one that cannot be expressed in Rocq.

In some models, we can express some additional operations, such as unrestricted loops. This is why we cannot generally use the first version of the `det_coerce` primitive, as it would be inconsistent in these models. However, in other models, it is acceptable because we cannot express non-termination (without involving OCaml oracles). In the oracle model, we could potentially use the first version if we choose to trust the oracles to terminate. We did not make that choice.

2.4 Physical equality

It may seem that physical equality is deterministic. This is not the case. For instance, consider the following OCaml fragment:

```
let () =
    let x = 0 in
    let a = [x] in
    let b = [x] in
    Format.printf "%b@." (a == b)
```

When compiling it using the bytecode compiler `ocamlc`, this prints false. However, using the native compiler `ocamlopt`, it is optimized away and prints true.

Testing for physical equality is therefore a good example of a useful impure computation. The impure monad offers a convenient way to define it as an oracle of type `forall {A}, A -> A -> ??bool`. To reason about it, we want to say that physical equality implies structural equality. We must however be careful to restrict ourselves to certain "safe" types. Therefore, we add a predicate `phys_eq_safe`. In particular, elements of `Set` are deemed safe for physical equality.

```
Axiom phys_eq_safe: Type -> Prop.
Axiom phys_eq_safe_set: ∀ {A: Set}, phys_eq_safe A.
Axiom phys_eq: ∀ {A}, A -> A -> ?? bool.
Axiom phys_eq_correct:
 ∀ A, phys_eq_safe A -> ∀ (x y: A), phys_eq x y ~~> true -> x = y.
Extract Inlined Constant phys_eq => "(==)".
```

3 A boolean decision diagram library

My work was to demonstrate that this idea was viable and useful. To that end, we have developed a certified SAT solver that uses boolean decision diagrams, themselves using many impure features of OCaml.

Boolean decision diagrams (BDD) [2] are a data structure that can be used to represent boolean functions. A BDD is a binary tree: the leaves are true or false, and each node contains a boolean variable. To evaluate the BDD on a certain valuation (map from variables to booleans), we descend down the tree, at each branch going left or right depending on the value of the variable. Using BDDs as our representation for formulas has the benefit of allowing a straightforward and fast algorithm to check for satisfiability. We support all boolean unary and binary operations on BDDs. Their semantics is to follow the usual notion of unary and binary operations on boolean functions.

3.1 Satisfiability testing and ordering

Let $\mathbb{V}$ be the set of boolean variables. The value of a BDD B over a valuation `val`: $\mathbb{V} \to \mathbb{B}$, denoted B val is defined inductively as

```
B val =
  match b with
    | true -> true | false -> false
    | node v l r -> if val v then r else l
```

A BDD is thus said to be satisfiable if there exists a valuation such that it evaluates to true.

To ensure that the satisfiability test can be done in a fast and straightforward way, we need a particular structural property: variables present in nodes must not occur again in subtrees. If we have this property, it is easy to check for satisfiability. We simply need to find a true leaf, and the path from it to the root tells us which values to give to variables. The structural property above ensures that variables occur at most once in the path, and thus that we don't need to change our mind midway or to resolve conflicts.

However, this particular property is not preserved by the various operations that can be done on BDDs, so we rather enforce a stronger order property. We assume that we know a total order on the set of variables. Then, we enforce the following ordering property: the sequence of variables encountered in a path from the root to a leaf must be strictly increasing. Ordering trivially implies the weaker structural property.

Thereafter, all BDDs are assumed to be ordered. Proving the algorithms correct mostly involved proving that they had the correct evaluation semantics and that they preserved order.

3.2 Hash consing

We want our BDDs to all be unique: two BDDs must be equal if and only if they are physically equal in memory. This is very important as it has many performance implications and allows many optimizations. Not only are equal subtrees inside a BDD shared, but they can be shared between BDDs across computations.

When we want to create a new node `(v, l, r)`, we check in a global hash table whether such a node already exists. If it does we return it, otherwise we create the new node and store it in the table. To speed up hashing, we use shallow hashing: we hash the pointers to `l` and `r` rather than recursing in them. Lastly, to ensure correctness (we do not assume that the hash table structure is correctly implemented!), we perform physical equality checks.

Unfortunately, the nature of OCaml makes it so that we cannot actually use pointers for hashing, because the garbage collector moves things around in memory. Pointers are therefore not stored as-is in the hash table. Instead we rely on globally unique identifiers that are stored in each node. This is yet another impure feature, since they are generated over the course of the program's execution.

Here, the impure oracles are the functions to manipulate the global hash table, as well as the hashing functions. The equality checks allow us to state and prove the correctness.

3.3 Boolean operations

We want to represent propositional formulas using our BDDs. Since these are built with the usual logical connectives, a crucial feature is the ability to apply these operations to BDDs.

Let $\star : \mathbb{B} \to \mathbb{B} \to \mathbb{B}$ be a binary operation on booleans. It can be applied to two BDDs like so:

```
a ⋆ b =
  match a, b with
    (* trivial cases *)
    | false, false | ... | true, true -> _
    (* trivial too *)
    | false, node v l r | ... | node v l r, true -> _
    | (node va la ra), (node vb lb rb) when a = b ->
        node a (la ⋆ lb) (ra ⋆ rb)
    | (node va la ra), (node vb lb rb) when a > b ->
        node b ( a ⋆ lb) ( a ⋆ rb)
    | (node va la ra), (node vb lb rb) when a < b ->
        node a (la ⋆  b) (ra ⋆  b)
```

Implementing this naively would be quite inefficient. We wouldn't be benefitting from the fact that many subtrees are shared: many BDD structures can lead to duplication of the work. A crucial optimization is to memoize the intermediate results. Therefore, we maintain a hash table with the goal of storing triplets `(a, b, c)` such that `c = a ⋆ b`.

This raises a major problem. We know that the elements inserted are going to verify this condition by construction. However, there is no guarantee that the elements retrieved still verify this condition, unless we assume that the hash table implementation is correct. Our hash table could very well return gibberish. This is very important because we only want to use OCaml code as untrusted oracles. We already had this problem for hash consing, but it was trivial to check that the retrieved data satisfied the necessary properties with simple physical equality tests. But here, this would be costly and would defeat the whole purpose of memoization.

In Rocq, we can replace these `(a, b, c)` triples with quadruples `(a, b, c, P)` where `P` is a proof that `c = a ⋆ b`. Our hash table thus maps couples `(a, b)` to quadruples `(a, b, c, P: a ⋆ b = c)`. Of course, our OCaml tables cannot store such proof objects, which are going to be erased at extraction time. A crucial insight by Sylvain Boulmé is that we can use a polymorphic table: the third and fourth parameters in the quadruples are replaced by a generic `'a`. Then, we can reason that, in a way similar to the theorems for free of P. Wadler [5], this polymorphic table is unable to forge new quadruples when we try to retrieve data, since it cannot generically create elements of `'a`. Hence, when we query the table for `(a, b)` and a quadruple `(a', b', c, P)` is returned by our oracle, it

suffices to check that `a == a'` and `b == b'`. Then we are sure that the quadruple is indeed the one we inserted, and we have `P: a ⋆ b = c`.

We use similar techniques to implement unary operations, that is, negation. It is however much simpler.

4 Results

Source code, both Rocq proof scripts and OCaml supporting libraries, can be found online[1].

4.1 Correctness

The BDD library was implemented in Rocq and designed to be extracted to OCaml. The impure oracles (hash tables, physical equality, etc.) all come from OCaml. As we explained in Section 2, the impure monad ensures that we don't abuse the non-deterministic and impure nature of these oracles to create logical absurdities on the Rocq side of things. Therefore, the algorithms are correct as far as Rocq is concerned.

However, without additionnal care, it could still be possible to extract the oracles to some nonsensical OCaml code, or just to unintentionally use an incorrect implementation of hash maps for instance. To avoid this, care was taken to ensure that the Rocq-side of things always checked the results returned by the oracles. The only case where that was not done was for the intermediary results hash maps used for binary operations, because the checks would be as costly as the computation itself. There, we instead relied on the "theorem for free"-like use of polymorphic tables to ensure that data could not be arbitrarily forged by the oracle.

4.2 Ease of use

Working inside of the impure monad is not very different from usual Rocq; as such, it succeeds at integrating with the rest of the proof assistant. The main difference with classical Rocq is that this approach makes it hard to reason about the results of computations inside the monad from the outside. For instance, it is customary in Rocq to first write an algorithm, then prove it:

```
Definition algo (x1: A1) ... (xn: An): A := ....
Theorem algo_correct:
 ∀ x1 ... xn, is_expected_result (algo x1 ... xn) x1 ... xn.
```

[1] https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/rocq-hash

But when working with the impure monad, it is much easier to instead treat the above as

```
Definition algo (x1: A1) ... (xn: An):
 ??{x: A | is_expected_result x x1 ... xn}.
```

and use tactics and refinement to build the body of the algorithm along with the proofs.

With this in mind, writing and proving the algorithms was not hindered by this technique. However, exiting the monad as described in Section 2.2 complicates matters and requires a lot of work. The final algorithm that was proven deterministic was the decision procedure of deciding whether a formula is satisfiable: `formula_satisfiable: formula -> bool`. Obviously, a decision procedure can only ever return the same result: any decision procedure must be deterministic. Had we had available the naive determinization primitive, it would have been trivial. However, the correct version of the `det_coerce` primitive required writing a completely separate SAT solver. Despite being a conceptually trivial algorithm – bruteforcing – it still required quite some effort to prove correct. This burden is unfortunately inherent to the `det_coerce` primitive as it stands.

The main programming error I made is unrelated to the use of the monad itself. To make the algorithms generic (mainly over the type of variables), I used module functors. Unfortunately, it is not possible in Rocq to specify `Extract Constant` directives to specify what axioms defined in functors should be extracted to. I had to wrap everything in functors to perform a kind of dependency injection. This makes the code quite unwieldly.

4.3 Performance

The whole goal of using impure oracles is to improve performance compared to pure Rocq. We made some preliminary benchmarks of our BDD library, and compared our results to Arlen Cox' library mlbdd[2].

We used both BDD libraries (ours and mlbdd) as SAT solvers. We measured how long they took to decide the satisfiability of formulas. The formulas were generated randomly using two algorithms:
- Algorithm 1 generates a formula by uniformly choosing a logical connective and recursively generating subformulas. This constructs deeply nested formulas.
- Algorithm 2 generates formulas in Conjunctive Normal Form. The lengths of the clauses and the number of clauses follow a geometric law.

[2]https://github.com/arlencox/mlbdd

Further investigations should be performed using repositories of SAT problems found online, as these formulas are probably not very representative of real SAT problems. Furthermore, measurements were only made on quite small formulas, in part due to time constraints.

With Algorithm 2, we can see on Figure 1 (plotting the mean execution time versus the expected formula size) that performance scales with formula size in a similar fashion between our library and mlbdd.

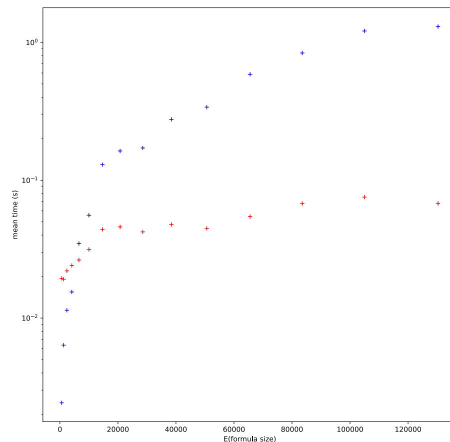
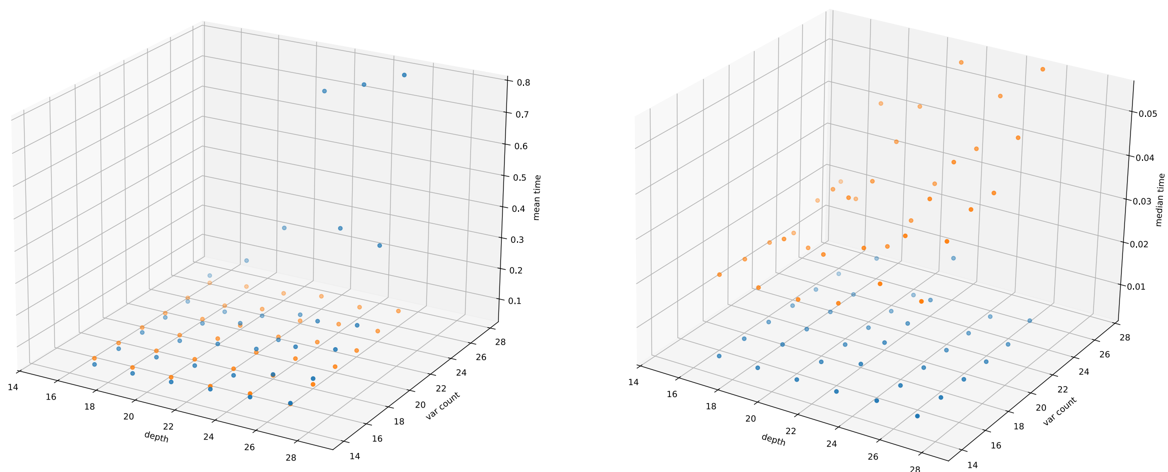


Figure 1: Performance on Algorithm 2 (red is mlbdd, blue is ours)

Our library consistently performs about 10 times slower than mlbdd. With Algorithm 1, we observe a surprising phenomenon. As shown in Figure 2b, median times are way better with our library. However, average times quickly become worse as shown by Figure 2a. This suggests that our library is generally more performant on most cases, but sometimes stumbles upon particular formulas on which it struggles enormously.



(a) Mean times (b) Median times

Figure 2: Performance on Algorithm 1 (red is mlbdd, blue is ours)

The implementation of mlbdd, according to its description, is relatively similar to ours. One major difference is that it implements *signed BDDs*. Signed BDDs are a variation on BDDs where each node carries on additional boolean indicating whether its right child should be interpreted negated. This makes negation trivial: it suffices to flip a boolean. We tried optimizing our implementation of negation, but it remains $O(n)$. Formulas with many negations may thus be the ones that cause these large slowdowns, but this hypothesis remains untested.

The most performant BDD libraries use additionnal heuristics. Notably, they can reorder variables on the fly [6]. Indeed, some orderings make the BDDs much smaller, and thus operations are made faster. mlbdd does not have this feature, but it is expected that our library would be even more outperformed against such implementations.

However, despite being slower than other BDD implementations, our SAT solver proved much, much faster (by factors of a thousand) than the naive implementation. We did not have time to compare it to other Rocq-only SAT solver, but this nonetheless seems to show the potential of this approach for developping more performant, but still formally verified, programs.

5 Future work

Originally, the plan was to develop the BDD library, then use it to make the SAT solver. If possible, it could have then been expanded to model checking. BDDs are indeed a classical tool for symbolic model checking. Unfortunately, the implementation of the SAT solver took much longer than expected, due to the need to write a reference implementation (see Section 4.2). I was thus unable to work on model checking beyond reviewing the literature. In the future, work could be done on developping such a certified, and performant, symbolic model checker.

As to the method itself, we explained in Section 4.1 that we had to manually check the oracles' results or rely on the polymorphism trick. In the future, investigations may be made into ways to automate these checks. Attempts to formalize the polymorphism trick are also underway by Sylvain Boulmé.

Conclusion

The impure monad and the associated primitives to exit it allow formally-verified programs to rely on impure but performant oracles in a safe way. This way, one can develop software that is faster than pure Rocq-only equivalent solutions, while retaining the correction guarantees of Rocq. This approach seems to yield promising results. For projects such as CompCert, it can be used to replace parts of the program with more performant algorithms. In this case, the difficulty needed to provide a reference deterministic implementation to exit the monad is not present, as one can simply use the old implementation.

Bibliography

[1] S. Boulmé, “Formally Verified Defensive Programming (efficient Coq-verified computations from untrusted ML oracles),” 2021. [Online]. Available: https://hal.science/tel-03356701

[2] Bryant, “Graph-Based Algorithms for Boolean Function Manipulation,” *IEEE Transactions on Computers*, no. 8, pp. 677–691, 1986, doi: 10.1109/TC.1986.1676819.

[3] X. Leroy, “Formal Verification of a Realistic Compiler,” *Communications of the ACM*, vol. 52, p. , 2009, doi: 10.1145/1538788.1538814.

[4] D. Monniaux and S. Boulmé, “Chamois: agile development of CompCert extensions for optimization and security,” in *35es Journées Francophones des Langages Applicatifs (JFLA 2024)*, Saint-Jacut-de-la-Mer, France, Jan. 2024. [Online]. Available: https://inria.hal.science/hal-04406465

[5] P. Wadler, “Theorems for free!,” in *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, in FPCA '89. Imperial College, London, United Kingdom: Association for Computing Machinery, 1989, pp. 347–359. doi: 10.1145/99370.99404.

[6] C. Jiang, J. Babar, G. Ciardo, A. S. Miner, and B. Smith, “Variable Reordering in Binary Decision Diagrams,” *26th International Workshop on Logic & Synthesis*, [Online]. Available: https://par.nsf.gov/biblio/10051062

Appendix 1 – Internship context

My internship was in the Verimag team at Université Grenoble-Alpes. It is an joint research team of CNRS and the university. It aims to develop “formal methods and tools for safe and secure computing” as well as “applications to cyber-physical systems”.

It is a relatively large team: it spans half a floor in the building, and we were almost 20 interns and PhD students. As such, it works on many projects that are somewhat loosely related. On the positive side, this had the benefit of allowing interactions with people working on very different subjects from mine, ranging from theoretical to very applied. On the negative side, there was perhaps less cohesion between people.

I felt that there wasn’t a huge lot of interactions between the researchers and the interns/PhD students. Thus, it was mostly with the latter that I had interactions. These were pleasant and fruitful, both inside and outside of the laboratory.

My main regret is that there weren’t many seminars and conferences, compared to what some classmates told me of their internships. Unfortunately, for practical reasons, I was unable to come to the “journée au vert”. But overall, this internship was a great experience.