From Sets to Bits in Coq

Arthur Blot, Pierre-Évariste Dagand, and Julia Lawall

Sorbonne Universités, UPMC Univ Paris 06, CNRS, Inria, LIP6 UMR 7606

Abstract. Computer Science abounds in folktales about how — in the early days of computer programming — bit vectors were ingeniously used to encode and manipulate finite sets. Algorithms have thus been developed to minimize memory footprint and maximize efficiency by taking advantage of microarchitectural features. With the development of automated and interactive theorem provers, finite sets have also made their way into the libraries of formalized mathematics. Tailored to ease proving, these representations are designed for symbolic manipulation rather than computational efficiency. This paper aims to bridge this gap. In the Coq proof assistant, we implement a bitset library and prove its correctness with respect to a formalization of finite sets. Our library enables a seamless interaction of sets for computing — bitsets — and sets for proving — finite sets.

1 Introduction

Sets form the building block of mathematics, while finite sets are a fundamental data structure of computer science. In the world of mathematics, finite sets enjoy appealing mathematical properties, such as a proof-irrelevant equality [17] and the extensionality principle for functions defined over finite sets. Computer scientists, on the other hand, have devised efficient algorithms for set operations based on the representation of finite sets as bit vectors and on bit twiddling [3, 27], exploiting the hardware's ability to efficiently process machine words.

With interactive theorem provers, sets are reinstituted as mathematical objects. While there are several finite set libraries in Coq, these implementations are far removed from those used in efficient code. Recent work on modeling low-level architectures, such as the ARM [14] or x86 [18] processors, however, have brought the magical world of bit twiddling within reach of our proof assistants. We are now able to specify and reason about low-level programs. In this paper, we shall tackle the implementation of bitsets and their associated operations.

Beyond the goal of certifying low-level programs, our work can contribute to mechanized reasoning itself. Indeed, our work is deeply rooted in the Curry-Howard correspondence, which blurs the line between proofs and computations. As shown by SSReflect, proof-by-reflection [2] is a powerful technique to scale proofs up. At the heart of this technique lies the fact that computation happens *within* the type theory. Last but not least, it is revealing that the finite set library provided by the Coq standard library originates from the CompCert [19] project, whose certified compiler crucially relies on such efficient datastructures.

This paper recounts an investigation from the concrete world of bit vectors to the abstract world of finite sets. It grew from a puzzled look at the first page of Warren's *Hacker's Delight* [27], where lies the cryptic formula $x \mathbin{\&} (x-1)$. How do we translate the English specification given in the book into a formal definition? How do we prove that this formula meets its specification? Could Coq generate efficient and trustworthy code from it? And how efficiently could we simulate it within Coq itself? We aim to answer those questions in the following.

This paper makes the following contributions:

– in Section 3, we establish a bijection between bitsets and sets over finite types. Following a refinement approach, we show that a significant part of SSReflect finset library can be refined to operations manipulating bitsets;
– in Section 4, we develop a trustworthy extraction of bitsets down to OCaml's machine integers. While we are bound to axiomatize machine integers, we adopt a methodology based on exhaustive testing to gain greater confidence in our model;
– in Section 5, we demonstrate our library through two applications. We have implemented a Bloom filter datastructure, proving the absence of false negatives. We have also implemented and verified the n-queens algorithm [13].

The source code of our development is available at

`https://github.com/artart78/coq-bitset`

2 Finite Sets and Bit Vectors in Coq

Let us first recall a Coq formalization of finite sets and a formalization of bit vectors. The former provides basic algebraic operations, such as union, intersection, complement, *etc.*, and more advanced ones, such as cardinality and minimum. The latter offer extended support for describing bit-level computations, such as logical and arithmetic operation on memory words.

2.1 A Finite Set Library: finset

To manipulate finite sets in Coq, we rely on the finset library [20], provided by the Mathematical Components platform [15]. The finset library provides set-theoretic operators for dealing with sets of elements of a finite type, *i.e.* sets of finite cardinality. A finite set being a finite type itself, we also make extensive use of SSReflect's fintype library [25]. We recall their key definitions in Table 1.

Remark 1. It is crucial to constrain the underlying type to be finite: a bitset represents collections thanks to their finite enumeration. Indeed, the bitset encodes the fact that, for any given set, every element of this enumeration is either present or absent. □

The canonical example of a finite set is the type `'I_n : Type` (where `n : nat` is an index) of the finite ordinals below `n`. Intuitively, `'I_n` represents the set $\{0, \cdots, n-1\}$. Every finite set of cardinality n is isomorphic to `'I_n`.

<table>
<tr><th>Coq judgment</th><th>Informal semantics</th></tr>
<tr><td>T : finType</td><td>card(T) is finite</td></tr>
<tr><td>T : finType ⊢ {set T} : Type</td><td>type of finite sets of T-elements</td></tr>
<tr><td>A : {set T} ⊢ #| A | : nat</td><td>cardinality of the set A</td></tr>
<tr><td>x : A ⊢ x \in A : bool</td><td>membership test</td></tr>
<tr><td>k : T ⊢ k |: A : {set T}</td><td>insertion of the element k in A</td></tr>
<tr><td>A :\ k : {set T}</td><td>removal of the element k from A</td></tr>
<tr><td>P : T → bool ⊢ [set x : T | P] : {set T}</td><td>subset of T satisfying P</td></tr>
<tr><td>A, B : {set T} ⊢ A :|: B : {set T}</td><td>union of A and B</td></tr>
<tr><td>A, B : {set T} ⊢ A :&: B : {set T}</td><td>intersection of A and B</td></tr>
<tr><td>A, B : {set T} ⊢ A :\: B : {set T}</td><td>difference of A and B</td></tr>
<tr><td>i0 : T ⊢ [arg min_(i < i0 in A) M] : T</td><td>an i \in A minimizing M</td></tr>
</table>

Table 1. Key operations on finite sets [20, 25]

<table>
<tr><th>Coqjudgment</th><th>Informal semantics</th></tr>
<tr><td>n : nat ⊢ BITS n : Type</td><td>vector of n bits</td></tr>
<tr><td>bs : BITS n, k : nat ⊢ getBit bs k : bool</td><td>test the k^{th} bit</td></tr>
<tr><td>xs, ys : BITS n ⊢ andB xs ys : BITS n</td><td>bitwise and</td></tr>
<tr><td>xs, ys : BITS n ⊢ orB xs ys : BITS n</td><td>bitwise or</td></tr>
<tr><td>xs, ys : BITS n ⊢ xorB xs ys : BITS n</td><td>bitwise xor</td></tr>
<tr><td>xs : BITS n ⊢ invB xs : BITS n</td><td>bitwise negation</td></tr>
<tr><td>xs : BITS n, k: nat ⊢ shrBn xs k : BITS n</td><td>right shift by k bits</td></tr>
<tr><td>xs : BITS n, k: nat ⊢ shlBn xs k : BITS n</td><td>left shift by k bits</td></tr>
</table>

Table 2. coq-bits API (fragment)

Remark 2. We are confident that our development could carry over to different formalizations of finite sets and finite ordinals such as, *e.g.*, the MSets library [10] and the Finite_sets library [9] provided by the Coq standard library.

2.2 A Bit Vector Library: coq-bits

To model operations on bitsets, we rely on coq-bits [18], a formalization of logical and arithmetic operations on bits. A bit vector is defined as an SSReflect tuple [26] of bits, *i.e.* a list of booleans of fixed (word) size. The key abstractions offered by this library are listed in Table 2. The library characterizes the interactions between these elementary operations and provides embeddings to and from $\mathbb{Z}/2^n\mathbb{Z}$.

3 Sets as Bit Vectors, Bit Vectors as Sets

There is an obvious bijection between a finite set of cardinality n and a bit vector of size n. Since we can sequentially enumerate each inhabitant of a finite type, we can uniquely characterize an inhabitant by its *rank* in this enumeration. Thus, a finite set can be concisely represented by setting the k^{th} bit to true if and only if the element of rank k belongs to the set.

In Coq, this bijection is captured by the (extensional) definition

```
Definition repr {n}(bs: BITS n) E :=
  E = [ set x : 'I_n | getBit bs x ].
```

where the right-hand side reads a standard set comprehension. We shall therefore say that a bit vector `bs` *represents* a set `E` if `repr bs E` holds.

The crux of this definition is to establish a relation between the abstract notion of finite sets — convenient for mathematical proofs — and the concrete artefact of bit vectors — enabling efficient computations. This relational presentation establishes a *data refinement* of finite sets by bitsets [8, 12].

In the following sections, we show that logical operations on finite sets are reflected by concrete manipulations on bitsets. In each case, we also prove that the refinement relation is preserved. As a result, an algorithm defined parametrically over the representation of a finite set will be instantiable to finite sets — for proofs — *and* bit sets — for computations. We shall illustrate this technique in Section 5.1.

3.1 Set Membership

Over finite sets, set membership merely requires checking whether an element belongs to an enumeration of the set's elements. It is therefore a decidable property, provided by the finset operator `x : T, A : set T` $\vdash$ `x \in A : bool`

In terms of bitsets, this can be implemented by shifting the k^{th} bit to the least significant position and masking the resulting bit vector with 1:

```
Definition get {n}(bs: BITS n)(k: 'I_n): bool
  := (andB (shrBn bs k) #1) == #1.
```

We then prove that our refinement of finite sets is respected by `get`. To do so, we show that, given a finite set `E` represented by a bitset `bs`, testing the membership of an element `k` in `E` is equivalent to getting the k^{th} bit in `bs`:

Lemma 1 (Representation of membership). *For a non-empty finite set `E` of cardinality `n.+1` represented by a bitset `bs`, `get` agrees with the set membership operation for every element of `E`, i.e.*

```
Lemma get_repr: forall n (k: 'I_n.+1)(bs: BITS n.+1) E, repr bs E →
      get bs k = (k \in E).
```

3.2 Inserting and Removing Elements

Inserting an element `k` into a bitset `bs` amounts to setting the k^{th} bit to 1. For instance, to set a specific bit, we apply an or-bitmask

```
Definition insert {n}(bs: BITS n) k: BITS n := orB bs (shlBn #1 k).
```

Once again, the formal specification and the computational realizer are related through a representation lemma, *e.g.*:

Lemma 2 (Representation of insertion).
For a finite set `E` *represented by a bitset* `bs`*, set insertion is refined by* `insert`*:*

```
Lemma insert_repr: forall n (bs: BITS n) (k: 'I_n) E, repr bs E →
    repr (insert bs k) (k |: E).
```

3.3 Algebra of Sets

The refinement relation holds for the standard algebra of sets. For two finite sets `A, B : set T`, we have that

- the complement `~: A` is realized by `invB` (bitwise negation),
- the intersection `A :&: B` is realized by `andB` (bitwise and),
- the union `A :|: B` is realized by `orB` (bitwise or), and
- the symmetrical difference `(A :\: B) :|: (B :\: A)` is realized by `xorB` (bitwise xor).

For each of these definitions, we prove the corresponding representation lemmas.

3.4 Cardinality

Computing the cardinality of a bitset requires counting the number of bits set to 1. To the delighted hacker, this merely amounts to implementing a *population count* algorithm [27, Sec. 5-1]. Several efficient implementations of this algorithm exist: we refer our reader to the above reference for a tour of each of them.

We chose to implement the population count *via* a lookup table. The gist of the algorithm is as follows. Let us consider a bitvector `bs` of size n (*e.g.*, $n = 64$) and let k be a divisor of n (*e.g.*, $k = 8$). We tabulate the number of 1s in all the bit vectors of size k. The idea is that for a sufficiently small value of k, this table fits within a single cache line. Therefore, to compute the number of 1s in `bs`, we can add the number obtained by looking up the key corresponding to the segment $[k \times i, k \times (i + 1) - 1]$ in the table, for $i \in [0, n/k - 1]$.

For example, on a 64-bit architecture, one would typically split the bit vector into segments of 8 bits, pre-computing a lookup table of 256 elements. Because the table fits in a single cache line, the individual lookups are fast. We have thus traded space (an impossibly large lookup table covering all 64-bit numbers) for time (by iterating the lookup 8 times instead of performing it once).

The first step, which happens off-line, thus involves computing a lookup table mapping any number between 0 and 2^k to its number of bits set:

```
Definition pop_table {n}(k: nat): seq (BITS n).
```

Looking up the number of bits set in the segment $[i \times k, i \times (k+1) - 1]$ is a matter of right shifts followed by a suitable and-mask to extract the segment. We obtain the segment's population count by a lookup in the pre-computed map:

```
Definition pop_elem {n}(k: nat)(bs: BITS n)(i: nat): BITS n
  := let x := andB (shrBn bs (i * k)) (decB (shlBn #1 k)) in
     nth (zero n) (pop_table k) (toNat x).
```

Finally, we obtain the total population count by iterating over the i segments of bit vectors of size k, adding their individual population counts:

```
Fixpoint popAux {n}(k: nat)(bs: BITS n)(i: nat): BITS n :=
  match i with
  | 0 => zero n
  | i'.+1 => addB (pop_elem k bs i') (popAux k bs i')
  end.

Definition cardinal {n}(k: nat)(bs: BITS n): BITS n
  := popAux k bs (n %/ k).
```

As before, the implementation has been shown to refine its specification.

3.5 Minimal Element

Finding the minimal element of a bitset amounts to identifying the least significant bit that is set to one. To put it another way, the rank of the minimal element is the *number of trailing zeros* [27, Sec. 5-4]. The classical formula for computing the number of trailing zeros for a bit vector of size n is given by

```
Definition ntz {n}(k: nat)(bs: BITS n): BITS n
  := subB #n (cardinal k (orB bs (negB bs))).
```

The intuition is that `orB bs (negB bs)` has the same number of trailing zeros as `bs` while all the bits beyond the minimal element are set. Therefore, the cardinality of this bit vector is its length minus the number of trailing zeros. We prove the usual representation lemma.

4 Trustworthy Extraction to OCaml

While bit vectors provide a faithful *model* of machine words, their actual representation in Coq — as lists of booleans — is far removed from reality. To extract our programs to efficient OCaml code, we must bridge this last gap and develop an axiomatic presentation of OCaml's machine integers.

We shall specify the semantics of this axiomatization by means of the coq-bits primitives. Once again, we rely on a refinement relation, stating that OCaml's integers refine coq-bits's integers (in fact, they are in bijection) and asserting that each axiomatized operation on OCaml's integers is a valid refinement of the corresponding operation in the coq-bits library. In effect, each abstract operation can be seen as a *specification*.

However, introducing new logical axioms cannot be taken lightly: one invalid assumption and the actual behavior of an OCaml operation could significantly diverge from its Coq specification. Built on such a quicksand, a formal proof is close to useless. For example, when extracting a Coq model of 8-bits integers onto OCaml 63-bit integers, it is all too easy to forget to clear the 55 most significant bits[1]. An operation overflowing a byte followed by a left shift — such as `shrB (shlB #0 9) 1` — would incorrectly expose the overflow, thus betraying the encoding. We can however take advantage of the fact that there is only a finite number of OCaml integers and that our specifications are decidable properties: we gain a higher level of trust in our model by exhaustively testing each specification against its OCaml counterpart.

4.1 Axiomatization and Extraction of `Int8`

Our axiomatization of machine integers merely involves importing the functions relative to integers defined in the OCaml standard library [23]. The list of axiomatized operations is summarized in Table 3. Concretely, the axioms and their realizers are defined as follows:

```
Axiom Int8: Type.
Extract Inlined Constant Int8 => "int".

Axiom lt: Int8 → Int8 → bool.
Extract Inlined Constant lt => "(<)".
```

To mediate between machine integers and bit vectors, we define two conversion functions

```
  Definition bitsToInt8  : BITS 8 → Int8 := (..).
  Definition bitsFromInt8 : Int8 → BITS 8 := (..).
```

which ought to establish a bijection between `Int8` and `BITS 8`. This fact cannot be established within Coq: `bitsFromInt8` and `bitsToInt8` perform various shifts and tests on machine integers, operations of which Coq has no knowledge of since they were axiomatized. To Coq, an axiomatized operation is nothing but a constant, *i.e.* a computationally inert token.

4.2 Gaining Trust in Extraction

Although our axiomatisation of machine integers is computationally inert, it can be extracted to OCaml, where it computes. In OCaml, we can therefore easily *run* the tests `bitsFromInt8 (bitsToInt8 bs) = bs` for all 8-bit vector `bs`. If this equality is experimentally verified, this provides a strong (meta-level) indication that `bitsToInt8` is cancelled by `bitsFromInt8`. We thus propose to gain trust in our model by (exhaustively) testing it [14]. We adopt a systematic infrastructure, inspired by translation validation [22]. Let us illustrate with the cancelativity property.

First of all, bit vectors of size 8 being finitely enumerable, we can write a test — in Coq — checking the cancelativity property for all possible bit vectors:

```
Definition bitsToInt8K_test: bool :=
 [forall bs , bitsFromInt8 (bitsToInt8 bs) == bs ].
```

[1] Needless to say, this example is drawn from the authors' harsh experience.

After extraction to OCaml, we can inspect the value `bitsToInt8K_test`: if it is `false`, then our specification is definitely incorrect. If it is `true`, then we may confidently accept the validation axiom

```
Axiom bitsToInt8K_valid: bitsToInt8K_test.
```

that reflects in SSReflect/Coq[2] the fact that we observed `true` in OCaml. Using this axiom and by the very definition of our test, we can *prove* the cancelativity property:

```
Lemma bitsToInt8K: cancel bitsToInt8 bitsFromInt8.
Proof.
  move=> bs; apply/eqP; move: bs.
  by apply/forallP: bitsToInt8K_valid.
Qed.
```

We follow the same methodology for the remaining specifications. For a desired specification `Spec`, we

1. implement an exhaustive test `spec_test` checking this property;
2. check that the extracted code returns `true`;
3. reflect its validity through an axiom `spec_valid`;
4. prove the desired property `Spec` from the test and its axiomatized validity.

To establish a bijection between `BITS 8` and `Int8`, we chose to test for injectivity of `bitsFromInt8`. From injectivity, we easily deduce cancelativity and bijectivity follows naturally. The injectivity lemma is stated as follows:

```
Lemma bitsFromInt8_inj: injective bitsFromInt8.
```

We can reflect the concluding equality in terms of the decidable equality `==` of bit vectors. However, the premise refers to the propositional equality of two `Int8` values. As such, we have no way to turn this statement into a checkable assertion. Morally, however, we know that the propositional equality over `Int8` should be consistent with OCaml's equality, which we have axiomatized as `eq`. This leads us to introduce the following — uncheckable — axiom:

```
Axiom eqInt8P : Equality.axiom eq.
```

where `eq` is an axiom that extracts to OCaml's structural equality test.

Similarly, we need a device for verifying universal quantifications over bit vectors. This decision procedure is realized by a simple enumeration routine (Figure 1) postulated as an axiom in Coq:

```
Axiom forallInt8 : (Int8 → bool) → bool.
Extract Inlined Constant forallInt8 => "Forall.forall_int".
```

The reflection property is once again uncheckable and therefore postulated

```
Axiom forallInt8P : forall P PP,
  viewP P PP → reflect (forall x, PP x) (forallInt8 (fun x => P x)).
```

[2] Boolean values are transparently lifted to types through the `is_true: bool → Prop` predicate that assigns the empty set to `false` and the unit set otherwise.

```
exception TestFailure of int ;;
let forall_int k =
  try
    for i = 0 to (1 lsl 8) - 1 do
      if (not (k i)) then
        raise (TestFailure i)
    done;
    true
  with (TestFailure i) →
    Printf.printf "failed %d\n" i; false
```

Fig. 1. Realizer for the `forallInt8` quantifier

Remark 3 (Trusted proving base).

The axioms `eqInt8P` and `forallInt8P` are the only axioms whose validity is not safeguarded by experimental validation. `eqInt8P` seems rather innocuous since it merely asserts that equality over OCaml's integers is defined precisely by OCaml's implementation of equality. An error in `forallInt8P` would be more consequential: if, for instance, the bounds `min_int` and `max_int` are both mistakenly set to `0`, then many false properties of machine integers would be presented as "experimentally true." As usual with a mathematical definition, it is only by confronting this definition against expected properties (such as, for example, the cyclic properties of `Int8`) that confidence can be gained in its validity. □

Using these two devices, we can test injectivity of `bitsFromInt8` with

```
Definition bitsFromInt8_inj_test: bool :=
  forallInt8 (fun x =>
    forallInt8 (fun y =>
      (bitsFromInt8 x == bitsFromInt8 y) ==> (eq x y))).
```

Running this test confirms its validity, which we can then postulate in our model. Injectivity follows, by a small proof involving the reflection of integer equality and of quantification over integers. From which we conclude by establishing the existence of a bijection between `Int8` and `BITS 8`:

```
Lemma bitsFromInt8_bij: bijective bitsFromInt8.
```

Remark 4. The execution of the OCaml-extracted test takes 4 seconds for to cover all 8-bits integers. The equivalent test for 16-bit integers did not complete after several hours. Using manually optimized (and, therefore, less trustworthy) OCaml code, we were able to run the tests in 0.23 seconds for 8-bit integers and in 7 hours 27 for 16-bit integers. Our hand-tuned test routine includes the following optimizations:

- factorizing the conversion from integers to bitsets across multiple tests;
- avoiding Peano integers by directly manipulating native OCaml integers.

The last point is essential for keeping a quadratic algorithm. Running the test for 32-bit integers is feasible, but is likely to take years of CPU time. Obviously, 64-bit integers cannot be exhaustively tested.

Despite our best efforts, our extraction remains *unverified* in a formal sense: it is trustworthy in as much as it gives consistent results with a particular version of the OCaml compiler (or interpreter), running on a particular operating system and a specific machine. To all intents and purposes, we have not provided a proof of correctness of our extraction: we have merely developed an experimental process by which to test its validity.

4.3 Refining Bit Vectors to Integers

The bijection naturally leads us to a refinement relation from Coq's bit vectors down to OCaml's machine integers. We thus define

```
Definition native_repr (i: Int8)(bs: BITS wordsize): bool
  := eq i (bitsToInt8 bs).
```

that is to say: an integer refines a bit vector if they are in bijection.

Following the refinement methodology, we then show that each operation on bit vectors is refined by a corresponding operation on machine integers. Let us consider the case of bitwise negation. We would like to prove that `lnot`, which extracts to OCaml's `lnot`, is a valid refinement of `invB`:

```
Lemma lnot_repr: forall i bs,
    native_repr i bs → native_repr (lnot i) (invB bs).
```

This statement reads as follows: if `i` is a native integer corresponding to the bitset `bs`, then the `lnot` operator acts exactly the same way as `invB` on it. The operator `invB` — bitwise negation — thus provides a specification for the operation `lnot` axiomatized in Coq. To prove this property, we craft a exhaustive test

```
Definition lnot_test: bool
  := forallInt8 (fun i =>
       native_repr (lnot i) (invB (bitsFromInt8 i))).
```

that we extract and run in OCaml. The result being `true`, we feel confident in asserting its validity to Coq:

```
Axiom lnot_valid: lnot_test.
```

The lemma `lnot_repr` follows from the definition of `lnot_test` and `lnot_valid`. We similarly specified, tested and proved the validity of all remaining operations (Table 3).

4.4 Refining Sets to Machine Integers

In Section 3, we have established a refinement relation between `BITS n` and finite sets. In Section 4.1, we have established another refinement relation between `Int8` and `BITS 8`. By transitivity, we obtain a refinement of finite sets to `Int8`:

Informal semantics	Coq axiom	OCaml extraction
Zero	`zero`	`0`
Successor	`suc`	`fun x → (x + 1) land 0xff`
Arithmetic negation	`neg`	`fun x → (-x) land 0xff`
Addition	`add`	`fun x y → (x + y) land 0xff`
Bitwise negation	`lnot`	`fun x → (lnot x) land 0xff`
Bitwise and	`land`	`(land)`
Bitwise or	`lor`	`(lor)`
Bitwise xor	`lxor`	`(lxor)`
Shift left	`lsl`	`fun x y → (x lsl y) land 0xff`
Shift right	`lsr`	`(lsr)`
Equality	`eq`	`(=)`
Comparison	`lt`	`(<)`

Table 3. Specifications, axioms and the realizers of `Int8`

```
Definition machine_repr (n: Int32)(E: {set 'I_wordsize}): Prop :=
  exists bv, native_repr n bv ∧ repr bv E.
```

The desired representation lemmas then carry over from finite sets to integers, trickling through bit vectors. For example, one defines the complement and easily proves its associated representation lemma

5 Applications

To illustrate our approach, we now tackle two examples of algorithms that rely on finite sets for their proof and bitsets for their efficient execution. In Section 5.1, we present a certified Bloom filter [4] implementation. In Section 5.2, we implement an algorithm solving the n-queens problem.

5.1 Bloom Filters

A Bloom filter is an efficient — but approximate — abstraction for monotone sets. It offers an operation for inserting an element into the set and another for testing membership. It is approximate in the sense that it is subject to *false positives*: an element might be signaled as belonging to a set into which it has never been inserted. However, it is free of *false negatives*: if the membership test fails, then it is indeed the case that the element has never been inserted. Combined with its small memory footprint, this last property makes this data structure very useful in practice.

Under the hood, a Bloom filter relies on a collection (H_i) of hashing functions onto `'I_n`, for some integer n (usually, the architecture's word size). Upon inserting an element p, we compute the i hashes of p and collect them in a single *signature set* of cardinality n:

```
Fixpoint bloomSig_aux (curFilter: T)(H: seq (P → 'I_wordsize))(e: P): T
 := match H with
   | [::] => curFilter
   | h :: H => bloomSig_aux ((singleton (h e)) \cup curFilter) H e
```

```
  end.

Definition bloomSig (H: seq (P → 'I_wordsize))(e: P): T
 := bloomSig_aux \emptyset H e.
```

The k^{th} element of the signature set is thus set if and only if there is hashing function reducing to this value. To update the Bloom filter, we simply take the union of this signature set and the previously-computed ones:

```
Definition bloomAdd (S: T)(H: seq (P → 'I_wordsize))(add_elt: P): T
 := S \cup (bloomSig H add_elt).
```

To check whether an element belongs to the filter, we once again compute its signature. If all the signature is a subset of the Bloom filter, then the corresponding element *may* have been inserted into the set. Otherwise, it definitely was not:

```
Definition bloomCheck (S: T)(H: seq (P → 'I_wordsize))(e: P) : bool
 := let sig := bloomSig H e in (sig \cap S) = sig.
```

The correctness of our implementation is established by

Theorem 1 (Absence of false negatives). *Let (H_i) be a collection of hashing functions. If an element belongs to the Bloom filter, then this element belongs to any subsequent extension of the Bloom filter. Or, contrapositively:*

```
Lemma bloom_correct: forall T T' H add check, machine_repr T T' →
  (~ bloomCheck (bloomAdd T H add) H check) →
    (~ bloomCheck T H check) ∧ (add <> check).
```

This ensures that the element is still detected in all subsequent Bloom filters generated by adding more elements, i.e. it will never be a false negative.

Remark 5. Although insertion (`bloomAdd`) and membership test (`bloomCheck`) are implemented over native integers for efficiency, the correctness argument is more easily established by reasoning over abstract sets. To bridge this gap, we merely instantiate our parametric definition to use finite sets (Section 4.4), thus obtaining an abstract specification `bloomAdd_finset`. Parametricity tells us that the specification and its implementation verify the refinement relation.

5.2 The n-queens Problem

Our second application is a freshman's classic. The n-queens problem involves finding the number of ways to place n queens on a $n \times n$ board so that no queen threatens another, *i.e.* belongs to the same row, column or diagonal. To do so, the algorithm recursively fill the board row-by-row, making sure at each step to put the queen on a safe column. To enforce this invariant, Richards [24] has shown that it is sufficient to maintain a (finite) set of occupied columns and of the left and right diagonals at the given position. Upon moving to the next row, we update the occupied column and the diagonal sets: the new queen occupies a new column, while the diagonals are merely shifted by one element.

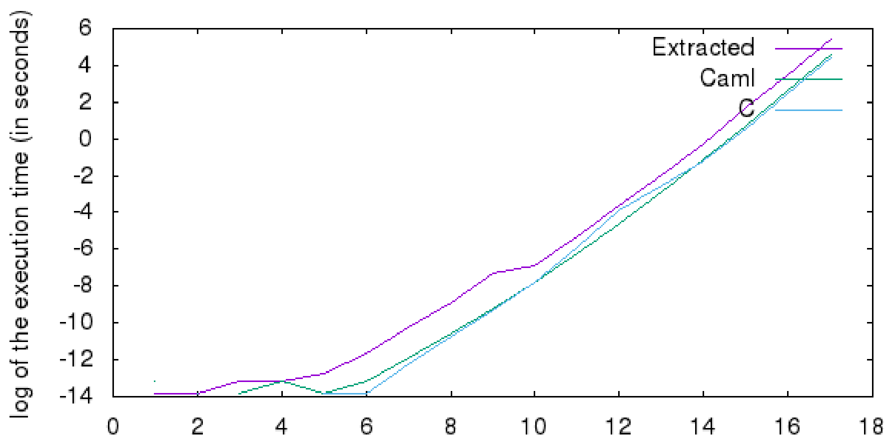

Fig. 2. Execution time for the n-queens algorithm

A particularly eager freshman (or one of Filliâtre's students [13]) would use a bitset `ld` to store the occupied left diagonals (relative to the current line), a bitset `rd` to store the occupied right diagonals (relative to the current line), and a bitset `col` to store the occupied columns. The set of possible positions is then concisely described by the set `~: (ld :|: rd :|: col)`. To decide on the next position to explore, we may take the minimal element of this set, using `ntz` (Section 3.5). The algorithm terminates when the set of columns `col` is full.

The correctness proof covers about 1300 lines of code, including about 50 lines of intermediairy definitions. Once again, we crucially rely on the equivalence between machine integers and finite sets (Section 4.4) to streamline the proof. We provide the performance of the extracted code in Figure 2, comparing it against a hand-written OCaml implementation and a C implementation. The hand-coded OCaml executes within 30% of the execution time of a reference C implementation, as is common for OCaml code. The extracted Coq code is twice as slow as the OCaml one. We attributes this slow-down to the naivety of our Coq implementation that encodes mutual recursion through an indexed type. The resulting extracted code thus repeatedly performs a (needless) boolean test in a tight loop.

6 Related Work

Our treatment of bitsets is rooted in the *data refinement* approach [1]. This approach involves relating a formal specification to a concrete implementation, refining the model at each step. Refinements have made their way into interactive theorem provers, such as Isabelle [7, 16] and Coq [11]. Our presentation builds upon the work of Denes *et al.* [12] in the Coq proof assistant. In particular, we follow the authors in using parametricity to abstract over representations and obtain the representation lemmas for (almost) free [8].

We demonstrated our library by implementing and verifying two algorithms in Coq. By doing so, we use Coq as a software verification platform. This approach is reminiscent — although at a much smaller scale — of the CFML [6] tool. Indeed, CFML provides a verification platform for OCaml programs by embedding an axiomatic model — the characteristic formula — in Coq and providing a program logic suitable to higher-order, effectful programs in Coq. We took the more lightweight (but also more restrictive) approach of writing programs directly in Coq, relying on extraction to obtain executable OCaml programs.

Why3 [5] is another platform for deductive program verification. It uses a collection of SMT solvers and interactive theorem provers to prove that programs meet their specifications. It supports manipulation of and reasoning about bitsets. To this end, the SMT solvers are extended with an axiomatic theory of bitsets. This theory has been shown consistent through a Coq model. Whenever an SMT solver fails to discharge a proof obligation, the Coq formalization can be used to, manually and interactively, prove the corresponding statement. Why3 is thus able to reason about algorithms manipulating bitsets automatically. For example, the n-queens algorithm was proved correct by Filliâtre [13]. Amazingly, most proof obligations (35 out of 41) are discharged automatically by the SMT solvers, freeing the programmer from the burden of writing formal proofs. The remaining proof obligations were proved in Coq, in as little as 142 lines of tactics in total. Our proof was meant to exercise our library and was thus developed without automated assistance. As a consequence, it is much longer (1200 lines of tactics) and admittedly more pedestrian.

7 Conclusion

In this paper, we have developed an effective formalization of bitsets, covering a significant fragment of SSReflect's finset library. We summarize the equivalences we have established in Table 4. Through this work, we hope to rejoice both Hackers and Mathematicians with delights [27]. To account for both — often divergent — point of views, we have adopted the data refinement approach advocated by Denes *et al.* [8]. We leveraged parametricity — a deep metamathematical property — to relate the proof-oriented and the computation-oriented specializations of our generic programs.

We would like to extend our work beyond a single machine word so as to support arbitrarily large bitsets. To this end, we would need native support for persistent (or non-persistent) arrays in Coq. Finally, our bitset library is but a first step toward building certified domain-specific compilers for programming low-level systems. In particular, device drivers are typically configured through intricate combinations of bitsets, *e.g.* for setting flags or checking the configuration status. We wish for our library to provide a verified connecting rod between the low-level interaction with the device and its high-level specification [21].

Acknowledgements We are grateful to Arthur Charguéraud and Maxime Dénès for several discussions on these questions. We also thank Jean-Christophe Filliâtre for suggesting the n-queens example, and Clément Fumex and Claude Marché for pointers to the literature. We finally thank the anonymous FLOPS reviewers, whose remarks helped us to streamline our presentation.

<table>
<tr><th>Informal semantics</th><th>finset definition</th><th>bitset definition</th></tr>
<tr><td>Membership:</td><td><code>\in : T → {set T} → bool</code></td><td><code>get: Int8 →</code>
<code>'I_wordsize → bool</code></td></tr>
<tr><td>Insertion:</td><td><code>|: : T →</code>
<code>{set T} → {set T}</code></td><td><code>insert: Int8 → Int8 → Int8</code></td></tr>
<tr><td>Removal:</td><td><code>:\ : T →</code>
<code>{set T} → {set T}</code></td><td><code>remove: Int8 → Int8 → Int8</code></td></tr>
<tr><td>Empty set:</td><td><code>set0 : {set T}</code></td><td><code>zero: Int8</code></td></tr>
<tr><td>Full set:</td><td><code>setT : {set T}</code></td><td><code>one: Int8</code></td></tr>
<tr><td>Complement:</td><td><code>~: : {set T} → {set T}</code></td><td><code>compl: Int8 → Int8</code></td></tr>
<tr><td>Intersection:</td><td><code>:&: : {set T} →</code>
<code>{set T} → {set T}</code></td><td><code>inter: Int8 → Int8 → Int8</code></td></tr>
<tr><td>Union:</td><td><code>:|: : {set T} →</code>
<code>{set T} → {set T}</code></td><td><code>union: Int8 → Int8 → Int8</code></td></tr>
<tr><td>Sym. difference:</td><td><code>:\: : {set T} →</code>
<code>{set T} → {set T}</code></td><td><code>symdiff: Int8 → Int8 → Int8</code></td></tr>
<tr><td>Cardinality:</td><td><code>#|_| : {set T} → nat</code></td><td><code>cardinal: Int8 → Int8</code></td></tr>
<tr><td>Minimal element:</td><td><code>[arg min_(i < _ in _) i]</code>
<code>: T → {set T} → T</code></td><td><code>ntz: Int8 → Int8</code></td></tr>
</table>

Table 4. Refined operations over finite sets

Bibliography

[1] Abrial, J.R.: The B-book: Assigning Programs to Meanings. Cambridge University Press (1996)

[2] Allen, S.F., Constable, R.L., Howe, D.J., Aitken, W.E.: The semantics of reflected proof. In: Symposium on Logic in Computer Science (LICS 1990). pp. 95–105 (1990)

[3] Beeler, M., Gosper, R.W., Schroeppel, R.: Hakmem. Tech. rep., Massachusetts Institute of Technology (1972), `http://hdl.handle.net/1721.1/6086`

[4] Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7), 422–426 (1970)

[5] Bobot, F., Filliâtre, J.C., Marché, C., Melquiond, G., Paskevich, A.: The Why3 platform. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay (2015)

[6] Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: International Conference on Functional programming (ICFP'11). pp. 418–430 (2011)

[7] Cock, D., Klein, G., Sewell, T.: Secure microkernels, state monads and scalable refinement. In: Theorem Proving in Higher Order Logics (TPHOL'08), pp. 167–182 (2008)

[8] Cohen, C., Dénès, M., Mörtberg, A.: Refinements for free! In: Certified Programs and Proofs (CPP'13), pp. 147–162 (2013)

[9] Coq Standard Library: Finite sets, `https://coq.inria.fr/library/Coq.Sets.Finite_sets.html`

[10] Coq Standard Library: Modular implementation of finite sets, `https://coq.inria.fr/library/Coq.MSets.MSets.html`

[11] Delaware, B., Pit-Claudel, C., Gross, J., Chlipala, A.: Fiat: Deductive synthesis of abstract data types in a proof assistant. In: Principles of Programming Languages (POPL'15). pp. 689–700 (2015)

[12] Dénès, M., Mörtberg, A., Siles, V.: A refinement-based approach to computational algebra in Coq. In: Interactive Theorem Proving (ITP'12), pp. 83–98 (2012)

[13] Filliâtre, J.C.: Verifying two lines of C with Why3: An exercise in program verification. In: Verified Software: Theories, Tools, Experiments (VSTTE'12), pp. 83–97 (2012)

[14] Fox, A., Myreen, M.: A trustworthy monadic formalization of the ARMv7 instruction set architecture. In: Interactive Theorem Proving (ITP'10), pp. 243–258 (2010)

[15] Gonthier, G., Mahboubi, A.: A small scale reflection extension for the Coq system. Tech. Rep. RR-6455, INRIA (2008)

[16] Haftmann, F., Krauss, A., Kunčar, O., Nipkow, T.: Data refinement in Isabelle/HOL. In: Interactive Theorem Proving (ITP 2013). pp. 100–115 (2013)

[17] Hedberg, M.: A coherence theorem for Martin-Löf's type theory. Journal of Functionnal Programming 8(4), 413–436 (1998)
[18] Kennedy, A., Benton, N., Jensen, J.B., Dagand, P.E.: Coq: The world's best macro assembler? In: Symposium on Principles and Practice of Declarative Programming (PPDP'13). pp. 13–24 (2013)
[19] Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009)
[20] Mathematical Components: Finite sets, `http://ssr2.msr-inria.inria.fr/doc/mathcomp-1.5/MathComp.finset.html`
[21] Mérillon, F., Réveillère, L., Consel, C., Marlet, R., Muller, G.: Devil: an IDL for hardware programming. In: Symposium on Operating Systems Design and Implementation (OSDI'00) (2000)
[22] Necula, G.C.: Translation validation for an optimizing compiler. In: Conference on Programming Language Design and Implementation (PLDI'00). pp. 83–94 (2000)
[23] OCaml Standard Library: Pervasives, `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html`
[24] Richards, M.: Backtracking algorithms in mcpl using bit patterns and recursion (1997)
[25] Ssreflect library: Finite type, `http://ssr.msr-inria.inria.fr/doc/ssreflect-1.5/Ssreflect.fintype.html`
[26] Ssreflect library: Tuple, `http://ssr.msr-inria.inria.fr/doc/mathcomp-1.5/MathComp.tuple.html`
[27] Warren, H.S.: Hacker's Delight. Addison-Wesley Professional (2012)