Verifying Two Lines of C with Why3: an Exercise in Program Verification[⋆]

Jean-Christophe Filliâtre

CNRS
LRI, Univ Paris-Sud, CNRS, Orsay F-91405
INRIA Saclay-Île-de-France, ProVal, Orsay F-91893

Abstract. This article details the formal verification of a 2-line C program that computes the number of solutions to the n-queens problem. The formal proof of (an abstraction of) the C code is performed using the Why3 tool to generate the verification conditions and several provers (Alt-Ergo, CVC3, Coq) to discharge them. The main purpose of this article is to illustrate the use of Why3 in verifying an algorithmically complex program.

1 Introduction

Even the shortest program can be a challenge for formal verification. This paper exemplifies this claim with the following 2-line C program:

```
t(a,b,c){int d=0,e=a&~b&~c,f=1;if(a)for(f=0;d=(e-=d)&-e;f+=t(a-d,(b+d)*2,(
c+d)/2));return f;}main(q){scanf("%d",&q);printf("%d\n",t(~(~0<<q),0,0));}
```

This rather obfuscated code was found on a web page gathering C signature programs[1] and was apparently authored by Marcel van Kervinc. This is a standalone C program that reads an integer n from standard input and prints another integer $f(n)$ on standard output. If n is smaller than the machine word size in bits (typically 32), then $f(n)$ appears to be the number of solutions to the well-known n-queens problem, that is the number of ways that n queens can be put on a $n \times n$ chessboard so that they do not attack each other. More surprisingly, this is a very efficient program to compute this number.

As a case study for Why3, a tool the author of this paper is co-developing [4], we consider verifying this program formally. Since Why3 is not addressing C programs, we make an abstraction of the algorithm above. Our goal is then a mechanically-assisted proof that this algorithm terminates and indeed computes the expected number. This is highly challenging, due to the algorithmic complexity of this program. The main contribution of this paper is to demonstrate the ability of our tool to tackle the wide range of verification issues involved in such a proof. In particular, it shows the relevance of using both automated and

[⋆] This work is partly supported by the ANR project *U3CAT* (ANR-08-SEGI-021) and the Open-DO project Hi-Lite.

[1] `http://www.iwriteiam.nl/SigProgC.html`

interactive theorem provers within the same framework. Additionally, this paper provides a nice benchmark for people developing tools for the verification of C programs; they may consider refining our proof into a proof of the C code above.

This paper is organized as follows. Section 2 "unobfuscates" the program, explaining the algorithm and its data. Section 3 briefly introduces Why3, a tool which takes annotated code as input and produces verification conditions in the native syntax of several existing provers. Section 4 details the verification process, namely the logical annotations inserted in the program and the methods used to discharge the resulting verification conditions. We conclude with a discussion in Section 5. Annotated source code and proofs are available online at `http://why3.lri.fr/queens/`. Proofs can be replayed in a batch mode.

2 Unobfuscation

Before we enter the formal verification process, we first explain this obfuscated C program. The code is divided into a recursive function `t`, which takes three integers as arguments and returns an integer, and a main function which reads an integer from standard input, calls function `t` and prints the result on standard output. With added type declarations and a bit of indentation, function `t` reads as follows:

```
int t(int a, int b, int c) {
  int d=0,e=a&~b&~c,f=1;
  if(a) for(f=0; d=(e-=d)&-e; f+=t(a-d,(b+d)*2,(c+d)/2));
  return f;
}
```

The assignment `d=(e-=d)&-e` does not strictly conform with ANSI C standard, because it assumes that the inner assignment `e-=d` is performed before evaluating `-e`. This is not guaranteed and the compiler may freely choose between both possible evaluation strategies. It is easy to turn the code in legal C: since `d` is initialized to 0, we can safely move assignment `e-=d` to the end of the loop body. Then we do not need the initialization `d=0` anymore[2]. The second modification we make is to replace the `main` function with a `queens` function from `int` to `int`, since we are only interested in the integer function and not in input-outputs. We end up with the code given in Fig. 1. Our goal is to show that `queens`(n) is indeed the number of solutions to the n-queens problem.

Let us now explain the algorithm and its data. This is a backtracking algorithm which fills the rows of the chessboard one at a time. More precisely, each call to `t` enumerates all possible positions for a queen on the current row inside the `for` loop and, for each of them, recursively calls `t` to fill the remaining rows. The number of solutions is accumulated in `f` and returned. The key idea is to use integers as *sets* or, equivalently, as *bit vectors*: i belongs to the "set" x if and only if the i-th bit of x is set. According to this trick, program variables `a`, `b`, `c`,

[2] This even reduces the size of the original code.

```
int t(int a, int b, int c) {
  int d, e=a&~b&~c, f=1;
  if (a)
    for (f=0; d=e&-e; e-=d)
      f += t(a-d,(b+d)*2,(c+d)/2));
  return f;
}
int queens(int n) {
  return t(~(~0<<n),0,0);
}
```

Fig. 1. Unobfuscated C code.

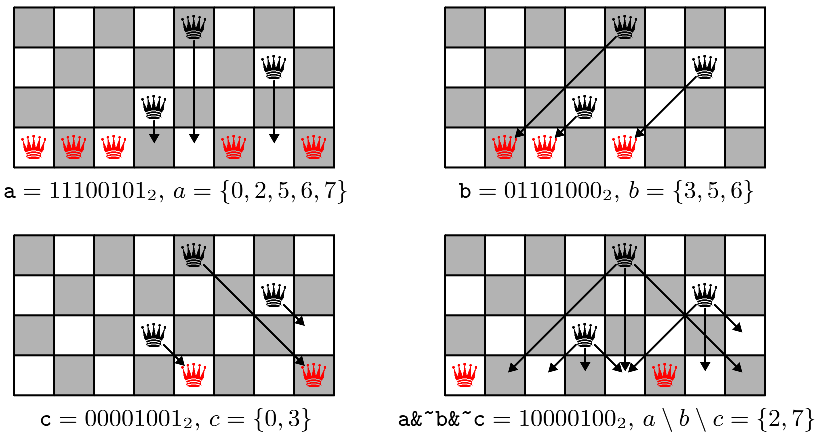
```
int $t$(set $a$, set $b$, set $c$)
    $f \leftarrow 1$
    if $a \neq \emptyset$
        $e \leftarrow (a \setminus b) \setminus c$
        $f \leftarrow 0$
        while $e \neq \emptyset$
            $d \leftarrow min\_elt(e)$
            $f \leftarrow f + t(a \setminus \{d\}, succ(b \cup \{d\}), pred(c \cup \{d\}))$
            $e \leftarrow e \setminus \{d\}$
    return $f$

int $queens$(int $n$)
    return $t(\{0, 1, \dots, n-1\}, \emptyset, \emptyset)$
```

Fig. 2. Abstract version of the code using sets.

`d` and `e` are seen as subsets of $\{0, 1, \dots, n-1\}$. Then almost all computations in this program are to be understood as set operations. Some of them are clear: `a&~b&~c` computes the set `a` $\setminus$ `b` $\setminus$ `c`, the test `if(a)` checks whether `a` is empty, etc. Others are more subtle. For instance, `e&-e` computes the smallest element of `e` (and returns the corresponding singleton set). This is a nice property of the two's complement arithmetic; see for instance [14, 10] for an explanation[3]. Then the result `d` can be removed from set `a` using subtraction `a-d` since the bit of `d` that is set is also set in `a`; similarly, `d` is added to sets `b` and `c` using a mere addition since the corresponding bit is not set in `b` and `c`. Another trick is the computation of the set $\{0, 1, \dots, \texttt{n}-1\}$ as `~(~0<<n)`. Finally, multiplication by 2 (resp. division by 2) is used to add 1 (resp. subtract 1) to each element of a set; from now on, we use *succ* and *pred* to denote those two set operations. We can now write a more abstract version of the code that only deals with finite sets. It is given in Fig. 2. Note that n, f, and returned values of t and *queens* are still integers.

It is now easier to explain the algorithm. Set a contains the columns not yet assigned to a queen, *i.e.* candidate positions for the queen to be set on the current

[3] This trick is used in Patricia trees [11] implementations.



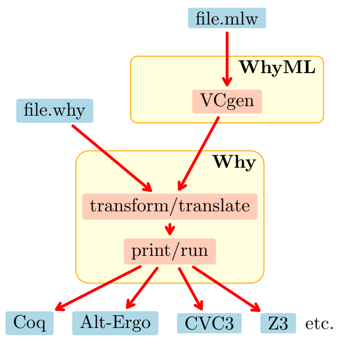
`a` $= 11100101_2$, $a = \{0,2,5,6,7\}$
`b` $= 01101000_2$, $b = \{3,5,6\}$
`c` $= 00001001_2$, $c = \{0,3\}$
`a&~b&~c` $= 10000100_2$, $a \setminus b \setminus c = \{2,7\}$

Fig. 3. Interpretation of variables `a`, `b`, and `c` as sets.

row. Initially, a contains all possible positions, that is $a = \{0, 1, \ldots, n-1\}$. If we have found one solution, a becomes empty, then we return 1. Otherwise, we have to consider all possible positions on the current row. Sets b and c respectively contain the positions to be avoided because they are on an ascending (resp. descending) diagonal of a queen on previous rows. Thus $e = a \setminus b \setminus c$ precisely contains the positions to be considered for the current row. They are all examined one at a time by repeatedly removing the smallest element from e, which is set to d. Then next rows are considered by a recursive call to t with a, b and c being updated according to the choice of column d for the current row: d is removed from the set of possible columns ($a \setminus \{d\}$), added to the set of ascending diagonals which is shifted ($succ(b \cup \{d\})$), and similarly added to the set of descending diagonals which is shifted the other way ($pred(c \cup \{d\})$). The values of a, b and c are illustrated in Fig. 3 for $n = 8$ on a configuration where 3 rows are already set (columns are numbered from right to left, starting from 0).

3 Overview of Why3

Why3 is a set of tools for program verification. Basically, it is composed of two parts, which are depicted to the right: a logical language called Why with an infrastructure to translate it to existing theorem provers; and a programming language called WhyML with a verification condition generator.



The logic of Why3 is a polymorphic first-order logic with algebraic data types and inductive predicates [5]. Logical declarations are organized in small units called *theories*. In the following, we use two such theories from

```
let rec t (a b c: set int) =
  if not (is_empty a) then begin
    let e = ref (diff (diff a b) c) in
    let f = ref 0 in
    while not (is_empty !e) do
      let d = min_elt !e in
      f := !f + t (remove d a) (succ (add d b)) (pred (add d c));
      e := remove d !e
    done;
    !f
  end else
    1

let queens (q: int) =
  t (below q) empty empty
```

Fig. 4. Why3 code for the program in Fig. 2.

Why3's standard library: integers and finite sets of integers. The latter provides a type `set int` and several operations: a constant `empty` for the empty set; functions `add`, `remove`, `diff`, `min_elt`, `cardinal`, and `below` (`below` n is the set $\{0, 1, \ldots, n-1\}$); a predicate `is_empty`. Operations `succ` and `pred` are missing from this library and we need to introduce them. First, we declare them as follows:

```
function succ (set int) : set int
function pred (set int) : set int
```

Then we axiomatize them as follows:

```
axiom succ_def:
  ∀ s: set int, i: int. mem i (succ s) ↔ i ≥ 1 ∧ mem (i-1) s
axiom pred_def:
  ∀ s: set int, i: int. mem i (pred s) ↔ i ≥ 0 ∧ mem (i+1) s
```

Why3 provides a way to show the consistency of these axioms (by providing a definition in Coq); however, we haven't done it.

On top of this logic, Why3 provides a programming language, WhyML, with a verification condition generator. This is a first-order language with an ML-flavored syntax. It provides the usual constructs of imperative programming (while loop, sequence, exceptions) as well as several constructs of ML (pattern matching, local functions, polymorphism). All symbols from the logic (types, functions, predicates) can be used in programs. Mutable data types can also be introduced, by means of record types with mutable fields. This includes polymorphic references, which are part of Why3's standard library. A reference `r` to a value of type τ has type `ref` τ, is created with function `ref`, is accessed with `!r`, and assigned with `r :=` e. Why3 code for the program in Fig. 2 is given in Fig. 4.

Programs are annotated using pre- and postconditions, loop invariants, and variants to ensure termination. Verification conditions are computed using a weakest precondition calculus and then passed to the back-end of Why3 to be sent to theorem provers.

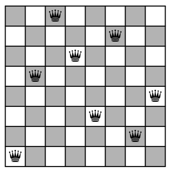
4 Verification

We focus here on the verification of the code in Fig. 4. (The verification of the original C code in Fig. 1 is discussed at the end of this paper.) We need to prove three properties regarding this code: it does not fail, it terminates, and it indeed computes the number of solutions to the n-queens problem. The first property is immediate since there is no division, no array access, or any similar operation that could fail. We will consider termination later, as part of the verification process (Sec. 4.2). Let us first focus on the specification.

4.1 Specification

We need to express that the value returned by a call to `queens` n is indeed the number of solutions to the n-queens problem. As we have seen, the program is building solutions one by one. Thus we have to prove that it finds *all* solutions, *only* solutions and that it does *not find* the same solution *twice*. There is a major difficulty here: the program is not storing anything, not even the current solution being built. How can we state properties about the solutions being found?

One solution is to use *ghost code*, that is additional code not participating in the computation of the final result but potentially accessing the program data. This ghost code will fill an array with all solutions. One solution is represented by an array of n integers. Each cell gives the column assigned to the queen on the corresponding row. For instance, the array $\boxed{5}\boxed{2}\boxed{4}\boxed{6}\boxed{0}\boxed{3}\boxed{1}\boxed{7}$ corresponds to the solution of the 8-queens problem displayed to the right. Rows are numbered from top to bottom and columns from right to left — the latter follows the usual convention of displaying least significant bits to the right, as in Fig. 3. Arrays used in ghost code do not really have to be "true" arrays: there is need neither for efficiency, nor for array bound checking. Thus we can model such arrays using purely applicative maps from Why3's standard library. Thus we simply define



```
type solution = map int int
```

We introduce a global variable `col` to record the current solution under construction, as well as a global variable `k` to record the next row to be filled:

```
val col: ref solution   (* solution under construction *)
val k  : ref int        (* next row in the current solution *)
```

The set of all solutions found so far is recorded into another array. It has type

```
type solutions = map int solution
```

and is declared as a global variable `sol`, together with another global variable `s` holding the next empty slot in `sol`:

```
val sol: ref solutions (* all solutions *)
val s  : ref int       (* next slot for a solution *)
```

If solutions are stored in `sol` starting from index 0, then `s` is also the total number of solutions found so far.

Using these four ghost variables, we can instrument the program with ghost code to record the solutions. First, we surround the recursive call to `f` (line 7) with code to record the value `d` for row `k` and to update `k`:

```
  (* ghost *) col := !col[!k ← d];
  (* ghost *) incr k;
  f := !f + t (remove d a) (succ (add d b)) (pred (add d c));
  (* ghost *) decr k;
```

Function `incr` (resp. `decr`) is a shortcut to increase (resp. decrease) an integer reference. Second, when a solution is found, we record it into `sol` and increase `s` by one, just before returning 1 (line 12):

```
    else begin
      (* ghost *) sol := !sol[!s ← !col];
      (* ghost *) incr s;
      1
    end
```

So far we have instrumented the code to record the solutions it finds. We still have to define what a solution is and to use this definition to specify the code. From now on, it is convenient to introduce the number n of queens as a parameter:

```
function n : int
```

This is not a limitation: a suitable precondition to function `queens` will say that its argument `q` is equal to `n` (and we don't even have callers). An alternative would be to pass `n` as a parameter everywhere, but we prefer avoiding it for greater clarity. To define what a solution is, we first define the notion of *partial solution*, up to row `k` (excluded):

```
predicate partial_solution (k: int) (s: solution) =
  ∀ i: int. 0 ≤ i < k →
  0 ≤ s[i] < n ∧
  (∀ j: int. 0 ≤ j < i →
     s[i] ≠ s[j] ∧ s[i]-s[j] ≠ i-j ∧ s[i]-s[j] ≠ j-i)
```

Note that we avoid the use of the absolute value function: we do so to relieve the automated theorem provers from resorting to the definition (typically an axiom). The notion of solution is derived immediately, by instantiating `k` with `n`:

```
predicate solution (s: solution) = partial_solution n s
```

To prove the absence of duplicate solutions, it is convenient to equip the set of solutions with a *total order*. It is naturally given by the code: since elements of `e` are processed in increasing order, by repeated use of function `min_elt`, solutions are found in lexicographic order.

For instance, the first five solutions for $n = 8$ are displayed to the right. To define the lexicographic order, we first define the property for two arrays to have a common prefix of length `i`:

0	4	7	5	2	6	1	3
0	5	7	2	6	3	1	4
0	6	3	5	7	1	4	2
0	6	4	7	1	3	5	2
1	3	5	7	2	0	6	4
⋮							

```
predicate eq_prefix (t u: map int α) (i: int) =
  ∀ k: int. 0 ≤ k < i → t[k] = u[k]
```

We make this a polymorphic predicate, to reuse it on both solutions and arrays of solutions. Then it is easy to define the lexicographic order over solutions:

```
predicate lt_sol (s1 s2: solution) =
  ∃ i: int. 0 ≤ i < n ∧ eq_prefix s1 s2 i ∧ s1[i] < s2[i]
```

Finally, we introduce two convenient shortcuts for the forthcoming specifications. Equality of two solutions is defined using `eq_prefix`:

```
predicate eq_sol (t u: solution) = eq_prefix t u n
```

The property for an array of solutions `s` to be sorted in increasing order between index `a` included and index `b` excluded is defined in an obvious way:

```
predicate sorted (s: solutions) (a b: int) =
  ∀ i j: int. a ≤ i < j < b → lt_sol s[i] s[j]
```

This completes the set of definitions needed to specify the code's behavior. The full specification for function `queens` (lines 14–15) is the following[4]:

```
let queens (q: int) =
  { 0 ≤ q = n ∧ !s = 0 ∧ !k = 0 }
  t (below q) empty empty
  { result = !s ∧ sorted !sol 0 !s ∧                                  (S)
    ∀ u: solution.
    solution u ↔ (∃ i:int. 0 ≤ i < result ∧ eq_sol u !sol[i]) }
```

The precondition requires both `s` and `k` to be initially equal to zero. The postcondition states that the returned value is equal to the number of solutions stored in array `sol`, that is `!s`. Additionally, it states that array `sol` is sorted and that an array `u` is a solution if and only if it appears in `sol`.

At this point, the reader should be convinced that specification (S) is indeed expressing that this program is computing the number of solutions to the n-queens problem. This is slightly subtle, since the absence of duplicated solutions is not immediate: it is only a provable consequence of `sol` being sorted. Our proof includes this property as a lemma.

4.2 Correctness Proof

We now have to prove that function `queens` terminates and obeys specification (S) above. As a warm-up, let us prove termination first.

[4] The code with all annotations is given in the appendix.

Termination. Termination reduces to that of function `t`. This involves proving its termination as a recursive function, as well as proving the termination of the `while` loop it contains. The termination of the `while` loop (lines 5–9) is immediate, since the cardinality of `e` is decreased by one at each step of the loop. We give the loop a variant accordingly:

```
while not (is_empty !e) do variant { cardinal !e } ...
```
(V_1)

The proof is immediate. Regarding the termination of recursive calls, there is also an obvious variant, namely the cardinality of `a`. It is indeed decreased by one at each recursive call. We give this variant for function `t` as follows:

```
let rec t (a b c: set int) variant { cardinal a } = ...
```
(V_2)

The proof is not immediate, however. Indeed, for the cardinality to decrease, we have to prove that `d` is an element of `a`. Within the loop, we only know for sure that `d` is an element of `e`. Thus we need a loop invariant to maintain that `e` is included in `a`. This could be the following:

```
while not (is_empty !e) do invariant { subset !e a } ...
```

However, we will later need a more accurate invariant, which states that `e` remains included in its initial value, that is `diff (diff a b) c`. Thus we favor the following invariant:

```
while not (is_empty !e) do
  invariant { subset !e (diff (diff a b) c) } ...
```
(I_1)

Remaining Annotations. To prove that function `queens` satisfies specification (S) above, we have to give function `t` a suitable specification as well. Obviously, this is a generalization of specification (S). Let us start with the precondition for `t`. First, variable `k` must contain a valid row number and `s` should be non-negative:

```
{ 0 ≤ !k ∧ !k + cardinal a = n ∧ !s ≥ 0 ∧ ... }
```
(P_1)

Second, sets `a`, `b`, and `c` must contain elements that are consistent with the contents of array `col`:

```
{ ...
  (∀ i: int. mem i a ↔
    (0 ≤ i < n ∧ ∀ j: int. 0 ≤ j < !k →  !col[j] ≠ i)) ∧
  (∀ i: int. i ≥ 0 → not (mem i b) ↔
    (∀ j: int. 0 ≤ j < !k → !col[j] ≠ i + j - !k)) ∧
  (∀ i: int. i ≥ 0 → not (mem i c) ↔
    (∀ j: int. 0 ≤ j < !k → !col[j] ≠ i + !k - j)) ∧ ... }
```
(P_2)

Finally, array `col` must contain a partial solution up to row `k` excluded:

```
{ ... partial_solution !k !col }
```
(P_3)

This completes the precondition for function `t`. Let us consider now its postcondition. First, it says that `s` must not decrease and that `k` must not be modified:

```
{ result = !s - old !s ≥ 0 ∧ !k = old !k ∧ ... }
```
(Q_1)

Then it says that all solutions found in this run of `t`, that is between the initial and final values of `s`, must be sorted in increasing order:

```
{ ... sorted !sol (old !s) !s ∧ ... }
```
(Q_2)

Additionally, these new solutions must be exactly the solutions extending the first `k` rows of array `col`:

```
{ ...
  (∀ u: solution.
     solution u ∧ eq_prefix !col u !k ↔
     ∃ i: int. old !s ≤ i < !s ∧ eq_sol u !sol[i]) ∧ ... }
```
(Q_3)

Finally, the first `k` rows of `col` must not be modified, and so are the solutions that were contained in `sol` prior to the call to `t`:

```
{ ... eq_prefix (old !col) !col !k ∧
      eq_prefix (old !sol) !sol (old !s) }
```
(Q_4)

With such pre- and postcondition for function `t`, function `queens` can be proved correct easily (verification conditions are discharged automatically).

The last step in the specification process is to come up with a loop invariant for function `t` (lines 5–9). It should be strong enough to establish postconditions (Q_1)–(Q_4). We already came up with invariant (I_1) to ensure termination. To ensure postcondition (Q_1), there is an obvious invariant regarding `s` and `k`:

```
{ ... !f = !s - at !s 'L ≥ 0 ∧ !k = at !k 'L ∧ ... }
```
(I_2)

Notation `at !s 'L` is used to refer to the value of `s` at the program point designated by label `'L`. This label is introduced before the `while` keyword at line 5 (this label appears in the code given in the appendix).

One key property to ensure that solutions are found in increasing order for `lt_sol` is that we traverse elements of `e` in increasing order, by repeated extraction of its minimum element. This must be turned into a loop invariant. It states that elements of `e` already considered are all smaller than elements of `e` yet to be considered:

```
{ ...
  (∀ i j: int.
    mem i (diff (at !e 'L) !e) → mem j !e → i < j) ∧ ... }
```
(I_3)

Additionally, we must maintain that solutions found in this run of `t`, that is between the initial value of `s` and its current value, are sorted in increasing order:

```
{ ... sorted !sol (at !s 'L) !s ∧ ... }
```
(I_4)

We also have to maintain property (P_3), since array `col` is modified by recursive calls to `t`:

```
{ ... partial_solution !k !col ∧ ... }
```
(I_5)

The most complex part of the loop invariant is surely the following, which is needed to ensure postcondition (Q_3). It states that the solutions found so far in this run of function `t` are exactly those extending the first `k` rows of `col` with an element of `e` already processed:

```
{ ...
  (∀ u: solution.
    solution u ∧ eq_prefix !col u !k ∧
    mem u[!k] (diff (at !e 'L) !e)                                (I6)
    ↔
    ∃ i: int. (at !s 'L) ≤ i < !s ∧ eq_sol u !sol[i]) ∧ ... }
```

Finally, we complete the loop invariant with an invariance property for `col` and `sol` similar to (Q_4):

```
{ ... eq_prefix (at !col 'L) !col !k ∧                             (I7)
      eq_prefix (at !sol 'L) !sol (at !s 'L) }
```

This completes the specification for function `t`. Fully annotated code is given in the appendix. We end up with 46 lines of annotations (not including the preliminary definitions and axiomatizations!) for 2 lines of code. This huge ratio should be considered as extreme: we are proving a very complex property of a smart algorithm.

Mechanical Proof. The proof is performed using the SMT solvers Alt-Ergo [3] and CVC3 [1], and the Coq proof assistant [13, 2]. Running Why3 on the resulting annotated source code produces 41 verification conditions for function `t` and 2 for function `queens`. The latter are automatically discharged by CVC3. As expected, verification conditions for `t` are more difficult to prove. Only 35 of them are discharged automatically, either by Alt-Ergo or CVC3. The remaining 6 verification conditions are discharged manually, using the Coq proof assistant. They are the following:

– precondition (P_2) for the recursive call to `t` (3 goals, corresponding to the 3 right to left implications);
– preservation of invariant (I_3);
– preservation of invariant (I_4);
– postcondition (Q_3) (left to right implication).

The Coq proof scripts amount to 142 lines of tactics and represent a few hours of work. It is important to point out that these Coq proofs only involve steps that could, in principle, be performed by SMT solvers as well (case analysis, Presburger arithmetic, definition expansion, rewriting, quantifier instantiation).

Beside verification conditions, our proof also contains two lemmas: one for the absence of duplicate solutions (see end of Section 4.1) and one technical lemma regarding `partial_solution`. They are respectively discharged by CVC3 and Alt-Ergo.

5 Discussion

We have presented the formal verification of an extremely short but also extremely complex program using Why3. Beyond being a nice specification exercise, it was the opportunity to introduce program verification using Why3 and to illustrate several key features such as user axiomatizations or combined use of interactive and automated theorem provers. We conclude this paper with several discussions.

Originality. The verification competition organized during VSTTE 2010 [9] already included a problem related to the n-queens problem. It was simpler, though, since the code to be verified only had to check the existence of at least one solution (and to return one, if any).

Ghost code. This case study is yet another example of where *ghost code* is useful in verification [12]. In this particular case, the program is enumerating the solutions to a problem, but does not store any of them, not even the current one. Thus we enriched the code with new statements so that a rich specification is possible. There is currently no support for ghost code in Why3; we plan to add this feature in the future. In particular, this will include a check that (1) ghost code is not modifying the program data, and (2) the program is not accessing the ghost data. In this proof, we have only performed this verification manually.

Verification of the original C code. We have not verified the original C code, only its abstraction into WhyML. Regarding the code structure, this is not really an issue, since all C features involved (recursive function, while loop, mutable variables) are available in WhyML as well. Regarding the code data, on the contrary, our proof did not consider the use of integers as bit vectors; we used sets instead. Our purpose was to focus on the specification of the algorithm.

Now that we have come up with a suitable specification, we could refine our proof into a proof of the original C code. A possible route is to introduce a function symbol, say `bits`, that turns an integer into the set of 1-bits in its two's complement representation. Then we can mechanically translate all the annotations, replacing `a` with `bits a`, `b` with `bits b`, and so on. The only change in the annotations is likely to be an extra precondition stating that the upper bits of `c` are zeros (otherwise, ones could be erroneously introduced by the divisions by two). The proof then requires extra lemmas to justify the tricks used in the code. For instance, a lemma will show that, under suitable conditions on `x`, we have `bits (x & -x) = singleton (min_elt (bits x))`. A bit vector library with two's complement interpretations is currently under development in Why3; we consider refining our proof along the lines we just sketched in a future work. Last, translating the resulting proof into a verification tool for C programs, such as VCC [6] or Frama-C [8], should be straightforward. It would be interesting to see which level of proof automation can be achieved.

Overflows. There are two kinds of integer overflows in this program, depending on the use of integers as bit vectors or as counters. Regarding integers used as bit vectors, we can easily cope with the boundedness of integers by imposing the precondition $n \leq$ `size` where `size` stands for the machine word size[5]. The program is performing overflows as soon as $n >$ `size`$/2$ since `b` may contain bits which will overflow due to the repetitive multiplications by 2. These are harmless overflows, but any suitable model should allow them.

Yet there is another source of integer overflows, in variable `f` and the returned value[6]. And it is more difficult to cope with. Even unsigned, 32-bit integers are not large enough to hold the number of solutions to the n-queens problem as soon as $n \geq 19$, the number of solutions for $n = 19$ being 4,968,057,848. Even if we use 64-bit integers for the result, we would need to limit n accordingly (most likely with $n \leq 28$) and then to prove the absence of overflow. But this would in turn require to know the number of solutions, which is precisely what we are trying to compute. An upper bound for the number of solutions would be enough, but there is no good one (and even if it would exist, this would require to be proved). One workaround would be to make the code detect overflows, and fail in such a case. Then our proof can be seen as a proof of the following statement: "if there is no overflow, then the returned value is indeed the number of solutions". Another workaround would be to perform the computation using arbitrary precision integers, which would be faithful to the proof we have made. But this would slow the computation; considering that a record attempt already requires dozens of years of total CPU time, we can hardly afford slowing it.

Acknowledgement. Natarajan Shankar kindly encouraged me to submit this paper; I'm glad I followed his advice. I'm grateful to Andrei Paskevich and Evgeny Makarov for their detailed proofreadings of an early version of this paper. Finally, I wish to thank the VSTTE reviewers for their helpful comments.

References

1. Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302, Berlin, Germany, July 2007. Springer.
2. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.
3. Francois Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The Alt-Ergo automated theorem prover, 2008. `http://alt-ergo.lri.fr/`.
4. Francois Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. *The Why3 platform*. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.64 edition, February 2011. `http://why3.lri.fr/`.

[5] Typically 32 or 64. On a practical point of view, imposing $n \leq 32$ is not an issue since the "world record", *i.e.* the largest value of n for which we have computed the solution, is $n = 26$ only [7]; we can expect all machines to be 64-bits before the limit $n = 32$ is reached, if ever.

[6] Historically, the first number of solutions announced for $n = 24$ was erroneous, due to 182 overflows on 32-bit integers — see [7] for details.

5. Francois Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, August 2011.
6. Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *Lecture Notes in Computer Science*. Springer, 2009.
7. Technische Universität Dresden. The world record to the n-queens puzzle ($n = 26$). `http://queens.inf.tu-dresden.de/`, 2009.
8. The Frama-C platform for static analysis of C programs, 2008. `http://www.frama-c.cea.fr/`.
9. Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. The 1st Verified Software Competition: Experience report. In Michael Butler and Wolfram Schulte, editors, *Proceedings, 17th International Symposium on Formal Methods (FM 2011)*, volume 6664 of *LNCS*. Springer, 2011. Materials available at `www.vscomp.org`.
10. Donald E. Knuth. *The Art of Computer Programming, volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley Professional, 1st edition, 2011.
11. Donald R. Morrison. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4):514–534, 1968.
12. Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, may 1976.
13. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.3*, 2010. `http://coq.inria.fr`.
14. Henry S. Warren. *Hackers's Delight*. Addison-Wesley, 2003.

A Annotated Source Code

This is the annotated source code for the program in Fig. 4.

```
let rec t (a b c : set int) variant { cardinal a } =
  { 0 ≤ !k ∧ !k + cardinal a = n ∧ !s ≥ 0 ∧
    (∀ i: int. mem i a ↔
      (0≤i<n ∧ ∀ j: int. 0 ≤ j < !k →  !col[j] ≠ i)) ∧
    (∀ i: int. i≥0 → not (mem i b) ↔
      (∀ j: int. 0 ≤ j < !k → !col[j] ≠ i + j - !k)) ∧
    (∀ i: int. i≥0 → not (mem i c) ↔
      (∀ j: int. 0 ≤ j < !k → !col[j] ≠ i + !k - j)) ∧
   partial_solution !k !col }
  if not (is_empty a) then begin
    let e = ref (diff (diff a b) c) in
    let f = ref 0 in
 'L:while not (is_empty !e) do
      invariant {
        !f = !s - at !s 'L ≥ 0 ∧ !k = at !k 'L ∧
        subset !e (diff (diff a b) c) ∧
        partial_solution !k !col ∧
        sorted !sol (at !s 'L) !s ∧
        (∀ i j: int. mem i (diff (at !e 'L) !e) → mem j !e → i < j) ∧
        (∀ u: solution.
         (solution u ∧ eq_prefix !col u !k ∧ mem u[!k] (diff (at !e 'L) !e))
         ↔
         (∃ i: int. (at !s 'L) ≤ i < !s ∧ eq_sol u !sol[i])) ∧
        eq_prefix (at !col 'L) !col (at !k 'L) ∧
        eq_prefix (at !sol 'L) !sol (at !s 'L) }
      variant { cardinal !e }
      let d = min_elt !e in
      (* ghost *) col := !col[!k ← d];
      (* ghost *) incr k;
      f := !f + t (remove d a) (succ (add d b)) (pred (add d c));
      (* ghost *) decr k;
      e := remove d !e
    done;
    !f
  end else begin
    (* ghost *) sol := !sol[!s ← !col];
    (* ghost *) incr s;
    1
  end
  { result = !s - old !s ≥ 0 ∧ !k = old !k ∧
    sorted !sol (old !s) !s ∧
    (∀ u: solution.
       ((solution u ∧ eq_prefix !col u !k) ↔
        (∃ i: int. old !s ≤ i < !s ∧ eq_sol u !sol[i]))) ∧
    eq_prefix (old !col) !col !k ∧
    eq_prefix (old !sol) !sol (old !s) }

let queens (q: int) =
  { 0 ≤ q = n ∧ !s = 0 ∧ !k = 0 }
  t (below q) empty empty
  { result = !s ∧ sorted !sol 0 !s ∧
    ∀ u: solution.
      solution u ↔ (∃ i: int. 0 ≤ i < result ∧ eq_sol u !sol[i]) }
```