Cameleer – Case Studies

Mário Pereira

NOVA LINCS & Nova School of Science and Technology, Portugal

Abstract. This is a working document where I maintain a list of all OCaml examples that are fully verified by the Cameleer tool[1]. For each case study, I provide relevant numerical and proof effort metrics. All of the listed examples are available at the project's GitHub repository, under the `examples` directory. Working version of February 19, 2021.

Case study	# VCs	LOC / Spec. / Ghost	Proof time	Immediate
Applicative Queue	23	25 / 17 / 4	1.90	✓
Arithmetic Compiler	258	235 / 48 / 155	18.96	✗
Binary Multiplication	12	10 / 6 / 0	0.79	✓
Binary Search	37	62 / 40 / 0	1.54	✓
Binary Search Trees	33	20 / 26 / 0	1.34	✗
Checking a Large Routine	16	25 / 15 / 0	0.80	✓
CNF Convertion	124	113 / 47 / 0	5.34	✓
Duplicates in an Array	11	10 / 9 / 0	0.59	✓
Ephemeral Queue	44	40 / 29 / 7	1.87	✓
Even-odd Test	6	6 / 8 / 0	0.60	✓
Factorial	8	10 / 9 / 0	0.58	✓
Fast Exponentiation	5	4 / 4 / 0	0.57	✓
Fibonacci	15	16 / 15 / 2	0.79	✓
FIND Algorithm	6	13 / 7 / 0	0.61	✓
Insertion Sort	30	13 / 34 / 0	2.56	✓
Integer Square Root	11	8 / 14 / 0	0.80	✓
Leftist Heap	133	99 / 179 / 0	5.70	✓
Mjrty	25	33 / 12 / 0	1.31	✓
OCaml List.fold_left	26	5 / 20 / 0	1.75	✗
OCaml Stack	22	25 / 27 / 1	1.01	✓
Pairing Heap	68	65 / 101 / 25	3.60	✗
Program Proofs	63	93 / 54 / 24	1.94	✗
Same Fringe	23	22 / 16 / 0	0.82	✓
Small-step Iterators	51	43 / 50 / 2	6.53	✗
Tree Height CPS	4	8 / 8 / 0	0.67	✓
Union Find	63	36 / 29 / 7	3.93	✓

Fig. 1. Summary of the case studies verified with the Cameleer tool.

[1] `https://github.com/mariojppereira/cameleer`

1 Case Studies

The complete set of case studies verified with the Cameleer tool is depicted in Fig. 1. All of these are fully verified, using a combination of the Alt-Ergo, CVC4, and Z3 SMT solvers. For each example I provide, in order, the following metrics:

1. The number of verification conditions generated by Why3, for that proof.
2. The number of OCaml lines of code, GOSPEL lines of code, and ghost lines of code. Ghost code includes ghost record fields, ghost functions, and lemma functions. It is worth noting that ghost code is regular OCaml code, hence ghost LOC are already included in the OCaml LOC measure.
3. The proof time it takes (measured in seconds), in order to successfully replay the proof.
4. A visual indication that tells whether or not manual interaction is required in order to complete the proof. The (✓) sign marks a proof that requires no extra user interaction, other than writing down the GOSPEL specification.

In the following, I briefly describe each case study.

1.1 Applicative Queue

A FIFO data structure implemented *à la Baker*, *i.e.*, it stores the elements in the Queue using two different lists. The first such list stores the elements in order, while the second one stores them in reverse order. This is captured by the following type definition:

```
type 'a t = {
  self : 'a list * 'a list;
  view : 'a list [@ghost];
}
```

The `view` ghost field is used as a logical model of the data structure. This is connected to the real field `self` via the following type invariant:

```
  (*@ invariant let prefix, xiffus = self in
               (prefix = [] -> xiffus = []) &&
               view = prefix @ List.rev xiffus *)
```

The first part of the invariant provides a very simple criteria to test the queue for emptiness, as follows:

```
let [@logic] is_empty {self; _} = match self with
  | [], _ -> true
  | _ -> false
(*@ b = is_empty q
      ensures b <-> q.view = [] *)
```

New elements are always pushed to the head of the second list of the `self` field *except* when the queue is empty, so to respect the type invariant. This is as follows:

```
let add elt {self; view} = match self with
  | [], [] ->
      { self = [elt], []; view = [elt] }
  | prefix, xiffus ->
      { self = prefix, elt :: xiffus; view = view @ [elt] }
(*@ r = add elt queue
      ensures r.view = queue.view @ (elt :: []) *)
```

Note the update of the `view` field, which boils down here to append the new element `elt` to the end of the list. This is inefficient but it does not jeopardize the expected complexity of this FIFO structure: this is ghost code hence it should be erased before compilation.

Elements are popped from the queue as follows:

```
let tail {self; view} = match self with
  | [_], xiffus ->
      { self = List.rev xiffus, []; view = tail_list view }
  | _ :: prefix, xiffus ->
      { self = prefix, xiffus; view = tail_list view }
  | [], _ -> raise Not_found
(*@ r = tail t
      raises  Not_found -> is_empty t
      ensures r.view = tail_list t.view *)
```

The `tail` function runs in $O(1)$ when there is more than one stored element; it runs in $O(n)$ when the queue has more than one element, where n is the size of the second list in `self`. It is this behavior of `tail` that provides an amortized execution complexity to a sequence of `head-tail` operations.

Function `tail` raises the `Not_found` exception (from the OCaml standard library) whenever the queue is empty. In order to update the `view` model, we use the `tail_list` function. This removes the last element of the `view` list, as follows:

```
let [@ghost] [@logic] tail_list = function
  | [] -> assert false
  | _ :: l -> l
(*@ r = tail_list q
      requires q <> []
      ensures  match q with [] -> false | _ :: l -> r = l *)
```

The above definition works only for non-empty queues, hence the given precondition. As such, the `assert false` expression marks a provable unreachable point in the code. Since function `tail_list` is only used to manipulate field `view`, we declare it as a ghost function. Also, and in order to provide a more concise and readable postcondition to function `tail`, we attach the `[@logic]` attribute to `tail_list`. This make it a functional symbol available both in specification and program.

This OCaml module features two more functions with the following type and contract:

```
val empty : 'a t
(*@ t = empty
      ensures t.view = [] *)

val head : 'a t -> 'a
(*@ x = head param
      raises  Not_found -> is_empty param
      ensures match param.view with [] -> false | y :: _ -> x = y *)
```

It is worth noting that this case study is issued from the OCamlGraph library source code [2]. Interestingly, the type invariant is provide as an informal comment in the library code.

1.2 Arithmetic Compiler

TODO:

1.3 Binary Multiplication

TODO:

1.4 Binary Search

TODO:

1.5 Binary Search Tree

1.6 Checking a Large Routine

TODO:

1.7 CNF Conversion

TODO:

1.8 Duplicates in an Array

TODO:

1.9 Ephemeral Queue

TODO:

[2] https://github.com/backtracking/ocamlgraph/blob/master/src/lib/persistentQueue.ml

1.10 Even-odd Test

TODO:

1.11 Factorial

TODO:

1.12 Fast Exponentiation

TODO:

1.13 Fibonacci

TODO:

1.14 FIND Routine

TODO:

1.15 Insertion Sort

TODO:

1.16 Integer Square Root

TODO:

1.17 Leftist Heap

TODO:

1.18 Mjrty Algorithm

TODO:

1.19 OCaml List.fold_left

TODO:

1.20 OCaml Stack

TODO:

1.21 Pairing Heap

TODO:

1.22 Program Proofs

TODO:

1.23 Same Fringe

TODO:

1.24 Small-step Iterators

TODO:

1.25 Tree Height in CPS

TODO:

1.26 Union Find

TODO: