Strong Automated Testing of OCaml Libraries

François Pottier

June 9, 2021
Sek, an implementation of ephemeral and persistent sequences.

- 4 abstract types (ephemeral/persistent sequences/iterators)
- 150+ operations
- shared mutable internal state
  - operations can have results or effects that are not directly observable
  - operations can affect objects that are not arguments

Want to test this library as a unit.
Reducing unit testing to whole-program testing

Want to exploit *random testing* and *fuzz testing*.

A fuzzer expects an *executable* and feeds it adverse data so as to try and make it crash.

Therefore, need to *wrap* the library with a driver.
Cannot test one operation at a time in isolation.

- (cannot generate data; cannot observe results)

Must generate and execute *scenarios*:

- *sequences* of instructions,
- involving *more than one* data structure.
OCaml is typed.

- generating well-typed scenarios requires *type information*

How can one tell if a scenario exhibits correct or incorrect behavior?

- a crash (e.g. an uncaught exception) is definitely bad; still,
- testing requires a *specification* of the expected behavior
I ask the user to provide

- a *reference implementation*, plus
- a description of the *relation* between candidate and reference

The reference and candidate

- must provide the same types and operations,
- but may implement each *abstract type* in *different* ways.

The goal is to *test a logical relation!*
  - recall Reynolds’ fable about Professors Descartes and Bessel.
Example: Persistent Arrays
Persistent arrays

The persistent array API.

```ocaml
type 'a t
val make : int -> 'a -> 'a t
val get : 'a t -> int -> 'a
val set : 'a t -> int -> 'a -> 'a t
```
An efficient but *incorrect* candidate implementation, in two lines.

```ocaml
include Stdlib.Array
let set a i x = set a i x; a
```
An inefficient but correct reference implementation, also in two lines.

```ocaml
include Stdlib.Array
let set a i x = let a = Array.copy a in set a i x; a
```
A typical output that we hope to obtain.

(* ./output/crashes/id:000000,sig:06,src:000000,op:flip16,pos:6 *)
(* @03: Failure in an observation: candidate and reference disagree. *)
(* @01 *) let a0 = make 1 0;;
(* @02 *) let a1 = set a0 0 1;;
(* @03 *) let observed = get a0 0;;
   assert (observed = 0);; (* candidate finds 1 * )
The specification and harness. R is reference, C is candidate.

```
(* Specs. *)
let array  = declare_abstract_type()
and element = sequential()
and length  = interval 0 16
and index a = interval 0 (R.length a) in
(* Operations. *)
declare "make" (length ^> element ^> array) R.make C.make;
declare "get" (array ^>> fun a -> index a ^> element) R.get C.get;
declare "set" (array ^>> fun a -> index a ^> element ^> array) R.set C.set;
(* Run, with 5 units of fuel. *)
main 5
```

A domain-specific language of specifications is used.
Combinators for Specifications
A value of type ('r, 'c) spec is a *runtime representation of a relation* between a reference value of type 'r and a candidate value of type 'c.

```plaintext
type ('r, 'c) spec
```
Concrete base types

Base types in *argument* position must come with a generator.

```ocaml
val constructible: (unit -> 't) -> ('t, 't) spec
```

Base types in *result* position must come with a comparator.

```ocaml
val deconstructible: ('t -> 't -> 'bool) -> ('t, 't) spec
```

The two can be combined, allowing a type to appear in either position.

```ocaml
val ifpol: ('r, 'c) spec -> ('r, 'c) spec -> ('r, 'c) spec
```

`sequential()` and `interval 0 16` have type `(int, int) spec` and are defined using these combinators.
Abstract base types

Abstract types need neither generator nor comparator.

```ocaml
val declare_abstract_type : unit -> ('r, 'c) spec
```
The ordinary arrow:

```
val (^>) : ('r1, 'c1) spec -> ('r2, 'c2) spec -> ('r1 -> 'r2, 'c1 -> 'c2) spec
```

An arrow cannot be nested in the left of an arrow.

A value of a function type cannot be generated (⋆).

(⋆) It can, in some cases, via a different mechanism.
A dependent arrow allows access the argument that has been picked:

\[ \text{val} \ (\ ^{\gg\gg} \ ) : \ (\ 'r_1, \ 'c_1 \ ) \ \text{spec} \to \ (\ 'r_1 \to (\ 'r_2, \ 'c_2 \ ) \ \text{spec}) \to \ (\ 'r_1 \to 'r_2, 'c_1 \to 'c_2 \ ) \ \text{spec} \]

This can be used e.g. to generate the next argument in a suitable range:

```plaintext
let index a = interval 0 (R.length a) in
declare "get" (array ^>> \textbf{fun} a -> index a ^> element) R.get C.get;
```
When this combinator is used, *the candidate runs first*, so the reference can *verify the candidate’s result* and use it to decide its own result.

```ocaml
type 'r diagnostic = Valid of 'r | Invalid
val nondet: ('r, 'c) spec -> ('c -> 'r diagnostic, 'c) spec
```

This allows non-deterministic specifications.
There are more combinators for

- structural types (products and sums),
- recursive types,
- transforming data after it has been generated,
- transforming data before it is compared,
- rejecting unsuitable data,
- dealing with exceptions,

and more.
Example: Nondeterminism
The OCaml API.

```ocaml
type t
val create : unit -> t
val next : t -> int
```

create() returns a fresh generator.

next g must produce a number that is

- nonnegative,
- strictly greater than the numbers produced by previous calls next g.
The specification.

```ml
let t = declare_abstract_type() in
declare "create" (unit ^> t) R.create C.create;
declare "next" (t ^> nondet int) R.next C.next;
```

Whereas `C.next` has type `C.t -> int`,
`R.next` has type `R.t -> int -> int diagnostic.`
The reference implementation.

type t = int ref
let create () = ref 0
let next (g : t) (candidate : int) : int diagnostic =
  if !g < candidate then (g := candidate; Valid candidate)
  else Invalid
Example: Semi-Persistent Arrays
Semi-Persistent Arrays

```ocaml
type 'a t
val make : int -> 'a -> 'a t
val get : 'a t -> int -> 'a
val set : 'a t -> int -> 'a -> 'a t
```

set \(a\) \(i\) \(x\) produces a new array, a *child* of \(a\).

At any time, the arrays created so far form a *tree*.

An array is *valid* if it is an ancestor of the most recently accessed array.

- Accessing an array invalidates all other arrays except its ancestors.
The reference serves as an oracle to reject invalid scenarios.

```plaintext
let elt = sequential()
and t = declare_abstract_type() in
declare "make" (lt 16 ^> elt ^> t)
    R.make  C.make;
declare "get"  (R.valid % t ^>> fun a -> lt (R.length a) ^> elt)
    R.get  C.get;
declare "set"  (R.valid % t ^>> fun a -> lt (R.length a) ^> elt ^> t)
    R.set  C.set;
```
The reference implementation recognizes valid accesses at runtime:

```ocaml
type 'a t = { data: 'a array; stack: 'a t list ref }

(* A validity test and an invalidation operation. *)
let valid a = List.memq a !(a.stack)

let invalidate_descendants a = ...

(* Operations on semi-persistent arrays. *)
let make n x = ...

let get a i = invalidate_descendants a; ...

let set a i x = invalidate_descendants a; ...
```
Conclusion

Used to test Sek, has helped find many bugs.

Supports multiple modes of use, such as:

- provide a *deterministic* reference, use *equality* to compare results
- provide a *trivial* reference, do not test results, watch out for crashes
- run candidate first, let reference *verify* the candidate’s result

Reference can serve as an *oracle* that helps pick suitable arguments.