Strong Automated Testing of OCaml Libraries

François Pottier
Inria Paris

Abstract

We present Monolith, a programmable tool that helps apply random testing or fuzz testing to an OCaml library. Monolith provides a rich specification language, which allows the user to describe her library’s API, and an engine, which generates clients of this API and executes them. This reduces the problem of testing a library to the problem of testing a complete program, one that is effectively addressed by off-the-shelf fuzzers such as AFL.

Prologue: How Ed Got Fired

Ed, an underpaid programmer at a company that sells software components, is assigned the task of implementing a persistent array library in OCaml. An “array”, in a broad sense, is a data structure that represents a map of some interval \([0, n]\) to values. A “persistent array” is immutable, or appears to be so. Ed is asked to implement the following signature:

\[
\begin{align*}
\text{type } 'a \ t \\
\text{val make : int } -> 'a \rightarrow 'a \ t \\
\text{val get : 'a \ t } -> \text{int } -> 'a \\
\text{val set : 'a \ t } -> \text{int } -> 'a \rightarrow 'a \ t
\end{align*}
\]

\(\text{make } n \ x\) creates an array of size \(n\) where every cell contains the value \(x\), while \(\text{get } a \ i\) returns the value stored in array \(a\) at index \(i\), and \(\text{set } a \ i \ x\) creates a new array that is identical to \(a\) except that the value stored at index \(i\) is \(x\). Note that \(\text{set } a \ i \ x\) does not modify the array \(a\).

Because Ed is unhappy with his work, he decides to botch the job. After spending an afternoon at the swimming pool, he returns with the following two-line implementation:

\[
\begin{align*}
\text{include Stdlib.Array} \\
\text{let set a i x = set a i x; a}
\end{align*}
\]

This implementation stores the data in a mutable array: it is incorrect. In the hope of disguising his fraud, Ed compiles it to machine code. Ed knows that the flaw cannot be detected by a “single-threaded” sequence of operations where each operation acts on the most recently-created array. To detect it, one must apply a \text{get} operation to an array that has already been the subject of a \text{set} operation. Ed happens to know that the employees of the company’s quality assurance department have been laid off a week earlier, and believes his boss incapable of imagining such a tricky scenario, so he thinks he can get away with it.

However, to Ed’s surprise, a few minutes after he has handed in his implementation, Bryn, his manager, barges into his office, yelling: “What is this? You’re fired!” as he slaps onto the desk a printout that reads as follows:

\[
\begin{align*}
\text{let a0 = make 1 0;;} \\
\text{let a1 = set a0 0 1;;} \\
\text{let observed = get a0 0;;} \\
\text{assert (observed = 0); (\text{candidate finds 1})}
\end{align*}
\]
Ed is shocked: how could Bryn come up so quickly with this scenario? “How...”, Ed begins. “Oh, that was easy,” Bryn snarls. “All I had to do was write a reference implementation of persistent arrays, in two lines, like this...”

```
include Stdlib.Array
let set a i x = let a = Array.copy a in set a i x; a
```

“A copy! Why didn’t I do a copy,” thinks Ed, who realizes that he could have been just as lazy without getting caught so quickly, while Bryn goes on: “... and write down the specifications of the three operations, like this:"

```
open Monolith
module R = Reference (* the above reference implementation *)
module C = Candidate (* Ed’s implementation *)
let () =
  (* 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
```

Bryn is gone. While packing, Ed pieces together how Monolith [11] enabled his boss to expose him so easily. In a few lines of code, Bryn declared the existence of an abstract type `array` and of three operations `make`, `get`, `set`. For each operation, he gave one specification and two implementations. For instance, the specification `length ^> element ^> array` indicates that `make` expects an integer length, which must be drawn from the interval `[0, 16]`, an integer element, where elements must be generated in a sequential manner, and returns an array. It is implemented by `C.make` on the candidate side and by `R.make` on the reference side. The specification `array ^>> fun a -> index a ^> element` states that `get` expects an array `a`, a nonnegative integer index that is less than the length of `a`, and returns an integer element. The abstract type `array` is represented by distinct types on the reference side and on the candidate side: indeed, the result type of `R.make` is `int R.t`, whereas the result type of `C.make` is `int C.t`. Bryn did not even need to point this out explicitly; this information was inferred.

Ed imagines that `main` randomly generates scenarios that exploit `make`, `get` and `set`, executes these scenarios using both the reference implementation and the candidate implementation, and compares the results. Either by purely random testing or by giving control of the source of random bits to an efficient grey-box mutation-based fuzzer, such as AFL [14], one can very quickly discover a scenario where the two implementations disagree.

“Unit testing has never been so easy,” thinks Ed, sadly, as he exits his former office.

1 Introduction

Fuzz testing, or fuzzing [9, 15], is an extremely effective approach to finding bugs in software. Fuzzing is usually applied to a standalone executable program. This program is expected to read a stream of input bits and to respond by either crashing or not crashing. The fuzzer’s job, then, is to craft a sequence of bits that causes a crash.
Fuzz testing is very much akin to random testing. The only difference between the two lies in the fact that the input stream is not random; it is controlled by an adversary. From the point of view of the program under test, this makes no difference. In terms of effectiveness, this can make a big difference: fuzzing can be much more effective at finding bugs than random testing.

Submitting a complete program to random testing or fuzz testing is definitely very interesting and useful. Yet, this begs the question: can these methods by applied to a library, as opposed to a self-contained executable program?

At first sight, this may seem difficult, because a library presents a much more complex interface to the outside world than a standalone program. A library usually publishes many types and operations, and may impose complex constraints on their usage. Manually or automatically constructing a good test suite (that is, a collection of usage scenarios that exercises the library in a wide variety of ways) is difficult [1].

In the setting of OCaml, this paper attempts to answer this question in the affirmative. We present Monolith [11], a programmable tool that helps transform a library into a self-contained program, therefore reducing the problem of testing this library to the problem of testing a complete program. Monolith offers a rich set of combinators that allow a library author to construct a description of her library’s API. The author must also provide a reference implementation of her library. Once these two pieces are provided, Monolith takes care of the rest. Its engine generates a valid usage scenario, executes this scenario, and (if an incorrect behavior is detected) prints this scenario and crashes. Thus, the library under test, combined with Monolith’s engine, forms a self-contained executable program that can be submitted to random testing or fuzz testing via standard means.

The paper is structured as follows. First (§2), we describe the combinators out of which specifications are built, which form a rich domain-specific language. Then, we briefly describe Monolith’s engine (§3), give a few more examples of the use of Monolith (§4), and discuss some of its limitations (§5). Finally, we give a brief review of the related work (§6) and conclude (§7).

2 A Specification Language

To be able to invoke the operations of the library under test, Monolith must have access to their specifications. There are two reasons why this is so: first, because OCaml is a typed language, Monolith must at least have access to the type of each operation; second, to be able to generate suitable arguments and to determine what to do with the results, Monolith must have access to richer specifications that may carry generators, preconditions, postconditions, and so on.

For this purpose, Monolith gives the user a means of constructing specifications. This takes the form of a type \( ('r, 'c) \text{spec} \) of specifications, together with a rich collection of combinators that build complex specifications out of simpler ones.

The type \( \text{spec} \) is parameterized with the types ‘\( r \) and ‘\( c \) of the values that a specification describes. The type ‘\( r \) is for the reference side, while ‘\( c \) is for the candidate side. Indeed, because an abstract type can be (and is usually) implemented in different ways by the reference implementation and by the candidate implementation, it is possible for an operation to have different types on either side.

To summarize, a specification of type \( ('r, 'c) \text{spec} \) is a runtime description of a pair of OCaml values whose respective types are ‘\( r \) and ‘\( c \). We sometimes refer to such a pair as a dual value. The specification tells Monolith how to exploit this dual value.

Within the universe of specifications, we distinguish subsets of constructible specifications, which describe values that Monolith is able to construct, and deconstructible specifications, which describe values that Monolith can deconstruct or test for equality. We impose a number
Figure 1: The specification language: core combinators

of rules, such as the following: (1) a function argument must be constructible; (2) a function result must be deconstructible; (3) a function is neither constructible nor deconstructible (§5). By lack of space, we omit the details. This discipline is currently enforced at runtime. In the future, we plan to enforce it statically via extra parameters of the type spec.

2.1 Core Specification Combinators

A slightly simplified presentation of the core specification combinators appears in Fig. 1. When displaying a scenario, Monolith must be able to print values, operations, and so on. In the figure, for the sake of simplicity, we have removed everything that has to do with printing.

**Basic concrete types** The function `constructible` (Fig. 1) lets Monolith regard an OCaml type `t` as constructible: that is, it equips Monolith with a way of constructing values of type `t`. No runtime description of this type needs be given. A generator, a function of type `unit -> t`, must be provided. Through an API not shown in this paper, a generator has access to a source of “random” bits, which may be either truly random or controlled by a fuzzer. The values produced by the generator are used both by the reference implementation and by the candidate implementation. For this reason, the return type of `constructible` is `('t, 't) spec`. 
The function `deconstructible` lets Monolith regard an OCaml type `t` as deconstructible: that is, it equips Monolith with a way of deconstructing values of type `t`. More accurately, it lets Monolith observe these values, that is, test them for equality. An equality test of type `t -> t -> bool` must be provided. It is used by Monolith to compare the values produced by the reference implementation and by the candidate implementation. A discrepancy is regarded as a fatal error, causing a report and a crash.

A constructible specification is used to describe an argument of an operation, whereas a deconstructible specification is used to describe the result of an operation. The combinator `ifpol` allows combining one specification of each kind into a single specification that is both constructible and deconstructible. It can be used to describe base types (§2.2), algebraic data types, first-class function types, and more (§5).

**Basic abstract types** The function `declare_abstract_type` makes Monolith aware of the existence of an abstract type, represented by the OCaml types `r` and `c` on the reference side and on the candidate side, respectively. The two implementations are allowed to use different runtime representations of this abstraction.

The specification returned by `declare_abstract_type` is constructible and deconstructible. Yet, Monolith never generates or observes a value of this type. When it must construct such a value, it selects a value of this type from those that have been obtained as the result of previous operations. When it must deconstruct such a value, it simply records this value in an environment, for use in a future operation.

An abstract type is by default opaque: when a candidate-side data structure has abstract type, Monolith has no way of checking at runtime that this data structure is well-formed or that it agrees with the data structure maintained by the reference implementation. Yet, it can be desirable to allow Monolith to see under the hood, so to speak, and to perform a well-formedness check. This can be done by supplying a `check` function of type `r -> c -> unit` as an optional argument to `declare_abstract_type`. (This is not shown in Fig. 1.) This function is expected to either silently succeed or fail by raising an exception. It is up to the user to decide how thorough, and how costly, this check should be.

The `check` function is applied by Monolith after every operation and to every dual value that has been recorded in the environment, even if this value is not ostensibly affected by this operation (that is, even if it is not an argument of this operation). This can be very useful when the candidate implementation involves shared mutable state: if an operation on a candidate data structure corrupts another candidate data structure that happens to share part of its representation with the former, this can be detected immediately, whereas in the absence of a `check` function, one or more operations on the latter would be required for the problem to become apparent.

**Structural types** The combinator `***` describes a value of a product type, that is, a pair. The combinators `option` and `result` describe a value of a sum type. These product and sum combinators produce constructible (resp. deconstructible) specifications when they are applied to constructible (resp. deconstructible) specifications.

Monolith is able to construct a value of a product or sum type out of smaller values. When constructing a sum type, this involves a choice: for this purpose, one “random” bit is read.

Monolith is able to deconstruct a value of a product or sum type into smaller values. A pair is decomposed into its components. When deconstructing a sum type, Monolith checks whether the reference implementation and the candidate implementation agree on the tag. For instance, if one implementation produces `None` while the other implementation produces a value of the form `Some _`, then a discrepancy has been detected, causing a report and a crash. If both implementations produce values of the form `Some _`, then the data constructor `Some` can be stripped off on both sides, and the deconstruction process can continue.
By combining basic concrete types, basic abstract types, products, and sums, one teaches Monolith how to handle values of composite structural types. For instance, suppose that some operation named \texttt{choose} has type \texttt{set \rightarrow \text{element} * \text{set} \text{option}}. Assuming that \texttt{element} has been declared as a (constructible and deconstructible) concrete type and assuming that \texttt{set} has been declared as an abstract type, the result of this operation can be described by the specification \texttt{option (element *** set)}, which is constructible and deconstructible. In particular, to deconstruct the result of \texttt{choose}, Monolith checks that either the reference implementation and the candidate implementation both return \texttt{None}, or they respectively return \texttt{Some (re, rs)} and \texttt{Some (ce, cs)}. In the latter case, Monolith further checks that the elements \texttt{re} and \texttt{ce} are equal and records the existence of a new dual value of abstract type \texttt{set}, whose reference-side and candidate-side projections are respectively \texttt{rs} and \texttt{cs}. This dual value can be passed as an argument to a future operation that requires a set.

**Recursive types** The combinator \texttt{fix} allows the user to construct a recursive specification, that is, a cyclic specification. This feature can be used to describe algebraic data types (§5). It can in theory cause Monolith to diverge while attempting to construct or deconstruct a value. This is not a problem in practice, because the probability of divergence is usually zero, and because a limit on execution time is usually imposed. AFL imposes such a limit.

**Function types** The combinator \texttt{^>} constructs a simple (nondependent) function specification. Its two arguments describe the function's domain and codomain. For instance, the operation \texttt{choose} that was mentioned above has specification \texttt{set \rightarrow option (element *** set)}. Like the function type constructor \texttt{->}, the combinator \texttt{^>} is right-associative, so curried functions can be easily described: for instance, an operation \texttt{add} that inserts an element into a set could have specification \texttt{element \rightarrow set \rightarrow set}.

The combinator \texttt{^>>} constructs a dependent function specification. In contrast with \texttt{^>}, it allows the specification of the codomain to depend on the function's argument. That is, the codomain is not just a specification, of type \texttt{('r2, 'c2) spec}, but a function of an argument to a specification, of type \texttt{'r1 -> ('r2, 'c2) spec}. This function has access to the reference-side projection of the actual argument generated by Monolith, and can refer to it in the specification of the codomain. This feature is typically exploited in concert with either \texttt{constructible} or the subset combinator \texttt{\%}. Either way, the point is to let the value of an earlier argument influence the choice of a later argument, either by dictating how it must be generated, or, after it has been generated, by rejecting it if it is unsuitable. This is explained next.

**Preconditions** Broadly speaking, there are two ways of expressing a precondition: either a priori, by influencing the manner in which an argument is generated, or a posteriori, by filtering out unsuitable values for this argument.

Let us illustrate the first approach via an example. Suppose that the library under test involves an abstract type of sequences. Suppose that \texttt{get} expects two arguments, namely a sequence \texttt{s} and an index \texttt{i} into the sequence \texttt{s}, and returns an element. The specification of this operation could be \texttt{seq ^>> fun s -> interval 0 (R.length s) ^> element}. The dependent function combinator \texttt{^>>} is used to bind the variable \texttt{s} to the reference-side projection of the first argument. This enables the function call \texttt{R.length}, which queries the reference implementation for the length of the sequence \texttt{s}. This in turn allows generating an integer index directly in the desired range. The constructible specification \texttt{interval i j} describes an integer in the semi-open interval \texttt{[i, j)}. It is defined in terms of \texttt{constructible}, \texttt{deconstructible}, and \texttt{ifpol}.

The second approach exploits the subset combinator \texttt{\%}. This combinator must be applied to a constructible specification and produces a constructible specification. It carries a predicate of type \texttt{'r -> bool}, which has access to the reference-side projection of the value that has been
constructed and must indicate whether this value is acceptable.

As an example, suppose that one wishes to test a library that involves an abstract type of file descriptors. Suppose that, via `declare_abstract_type`, one has obtained a specification `fd` of type `(R.fd, C.fd) spec`, where the types `R.fd` and `C.fd` are the types of file descriptors on the reference side and on the candidate side, respectively. Now, suppose that a file descriptor can be either valid or invalid, and suppose that the reference implementation keeps track of this information, so that there is a function `R.valid` of type `R.fd -> bool`, which determines whether a reference-side file descriptor is valid. Finally, let us assume that `close` requires a valid file descriptor, and invalidates it. The specification of this operation could be `R.valid % fd ^> unit` (The combinator `%` binds tighter than `^>`) The precondition `R.valid` prevents Monolith from applying `close` to an invalid file descriptor, which would not make sense. The fact that `close` invalidates its argument is not visible in the specification; it is visible in the implementation of the functions `R.close` and `R.valid`, which must update and consult the validity information associated with a descriptor. On the candidate side, no function `C.valid` is required: the candidate implementation need not keep track of validity at runtime.

In this example, the subset combinator `%` is used at an abstract type, namely the type of file descriptors. Although it can also be used at a concrete type, this is not recommended, as it is inefficient to generate unacceptable inputs, only to filter them out. It is preferable then to directly generate inputs that are acceptable by construction.

**Postconditions** Whereas a precondition expresses a property of an argument, which the library under test expects to hold, a postcondition expresses a property of a result, which the library under test must guarantee. It is often the case that the postcondition of an operation is deterministic, that is, a single value is considered a valid result. As this is the common case, Monolith adopts it as the default. Monolith requires the user to provide a reference implementation of each operation (§3.1) and, by default, expects the reference implementation and the candidate implementation to produce the same result. In that case, no explicit postcondition needs be given.

Sometimes, however, an operation has a nondeterministic specification: that is, several distinct results are permitted. In such a situation, the `nondet` combinator must be used. It produces a deconstructible specification. It is typically placed in the codomain of a function specification, that is, after the rightmost arrow combinator `^>`. Using `nondet` changes the type and the role of the reference implementation. Whereas the reference implementation normally returns a result of type `'r`, it must now return a function of type `'c -> `'r diagnostic` (Fig. 1). This function is in fact a postcondition: it has access to the value of type `'c produced by the candidate implementation and must return a value of type `'r diagnostic`. Such a value can be viewed as a truth value, enriched with additional information. Indeed, the type `'r diagnostic` is defined as a sum type: the data constructor `Valid` indicates that the candidate result is acceptable, whereas the data constructor `Invalid` indicates that it is not. In the former case, the diagnostic includes the result of type `'r` that the reference implementation wishes to return. An example of the use of `nondet` appears in §4.1.

**Input and output transformations** `map_into` and `map_outof` describe a transformation that must be applied by Monolith to a piece of data. They require the user to supply two transformation functions: a transformation of type `'r1 -> `'r2 for use on the reference side, and a transformation of type `'c1 -> `'c2 for use on the candidate side.

In the case of `map_outof`, the user must provide a specification of type `(R1, C1) spec`, that is, a description of the input of the transformation. This means that the transformation can be used to map data out of a type whose structure is known to Monolith. Thus, `map_outof` must be applied to a constructible specification, and returns one. It is typically used to preprocess
an argument of an operation.

In the case of `map_into`, the user must provide a specification of type \( ('r_2, 'c_2) \text{ spec} \), that is, a description of the output of the transformation. Thus, the transformation maps data into a type whose structure is known to Monolith. Therefore, if `map_outof` is applied to a deconstructible specification, then it returns one. It is typically used to postprocess the result of an operation. It can also be exploited to wrap an operation in an OCaml context that alters its calling convention: this is illustrated by the manner in which we handle exceptions (§2.2).

### 2.2 Derived Specification Combinators

**Predefined concrete types** A number of common concrete types are predefined via calls to `declare_concrete_type`. These include `unit`, `bool`, `int`, `exn` (Fig. 2). All are deconstructible. The types `unit` and `bool` are constructible, whereas `int` and `exn` are not, as it seems desirable to let the user ponder the choice of a suitable generator. `interval i j` is constructible, and causes an integer in the semi-open interval \([i, j)\) to be generated.

**Effectful functions** By default, Monolith allows neither the reference implementation nor the candidate implementation to raise an exception: both events are considered abnormal. If an operation is expected to raise an exception under normal circumstances, then this operation must be wrapped in an exception handler, so that Monolith can view it as a function that does not raise exceptions. This is done in a couple lines of code, as follows:

```ocaml
let handle f x = try Ok (f x) with e -> Error e
let (^!>) a b = map_into handle handle (a ^> result b exn)
```

The function `handle` turns a function that can raise an exception, whose type is \('a -> 'b\), into one that cannot raise an exception, whose type is \('a -> ('b, exn) result\). The application of `handle` to an operation is hidden underneath a thin layer of syntactic sugar: the exceptional function combinator `a ^!> b` is defined as a short-hand for an application of `map_into` that instructs Monolith to use `handle`, both on the reference side and on the candidate side, and to
deconstruct the value of type ("b, exn) result that is thus obtained. When this combinator is used, Monolith checks that either the reference implementation and candidate implementation both return a value, or they both raise an exception. In the latter case, Monolith uses the equality function at type exn to check that they raise related exceptions. By default, equality at type exn is OCaml's generic equality; however, this can be customized via a mechanism not described in this paper.

To sum up, the combinators ^> (Fig. 1) and ^!?> (Fig. 2) have the same type. The former specifies that no exception may be raised, while the latter specifies that the same exception (or related exceptions) must be raised by the reference implementation and by the candidate implementation.

It seems convenient to also define a ^?> b as a short-hand for a ^> nondet b, suggesting that nondeterminism is an effect:

```ocaml
let (^?>) a b = a ^> nondet b
```

Finally, it seems useful to define an arrow combinator that combines the exception effect and the nondeterminism effect. The combinator ^!?> is used to describe an operation that may choose whether it wishes to raise or not raise an exception. It is defined as follows:

```ocaml
let (^!?>) a b = map_into (fun x -> x) handle (a ^> nondet (result b exn))
```

The candidate implementation is wrapped with handle, so as to catch any exception that it raises. The nondet combinator indicates that the reference implementation has access to a value of type ("c2, exn) result, which describes the behavior of the candidate, and must produce a value of type ("r2, exn) result diagnostic, which indicates whether this behavior is acceptable and how the reference implementation responds. The reference implementation is not allowed to raise an exception: instead, it describes its own behavior by returning either Valid (Ok _) or Valid (Error _). For this reason, the reference implementation need not be wrapped with handle: it is wrapped with the identity function instead.

**Rearranging arguments** Sometimes, the arguments of an operation come in an inconvenient order: for instance, the first argument may be subject to a precondition that depends on the second argument. This is the case, for example, of a remove operation of type element -> set -> set, where the element that is passed as the first argument is required to be a member of the set that is passed as the second argument. When testing this operation, one would like to first pick a set s that is at hand, then pick a member x of this set. It would not make much sense to first blindly generate x and then hope that some set that contains x is at hand. Fortunately, it is easy to define a combinator rot2 that exchanges the order of the two arguments. This is yet another application of map_into:

```ocaml
let rot2 f y x = f x y
let rot2 spec = map_into rot2 rot2 spec
```

Using rot2, it is easy to express the specification of remove. It can be written under the form rot2 (R.nonempty % set ^>> fun s -> choose s ^> set), where choose s is an abbreviation for constructible (fun () -> R.choose s). It can be read informally as follows: once its arguments are exchanged, remove expects a nonempty set s and an element that must be chosen in the set s and returns a set. We assume that the function R.nonempty tests whether a (reference-side) set is nonempty and that the function R.choose, which has access to the source of “random” bits, chooses an element within a nonempty (reference-side) set.
3 Monolith’s Engine

3.1 Usage

Once the specification language described in the previous section is understood, using Monolith is very easy. The engine’s API consists of just two functions, declare and main (Fig. 3). declare declares the existence of an operation, and must be called once per operation; main runs the engine, and must be called once, after all operations have been declared.

An operation is described by its name, by a specification of type (‘r, ‘c) spec, and by its implementations on the reference side and on the candidate side, whose respective types are ‘r and ‘c. OCaml’s type discipline guarantees that the specification provided by the user is consistent with the types of the reference implementation and candidate implementation.

When the engine is started, by invoking main, an integer number n, which represents a certain amount of “fuel”, must be given. This number bounds the length of the scenarios that Monolith investigates: that is, Monolith generates and executes sequential scenarios of at most n instructions. An instruction is of the form let p = e, where e is an OCaml expression, typically an application of an operation to a suitable series of arguments, and p is an OCaml pattern, which deconstructs the result of executing e. Each scenario is executed both under the reference implementation and under the candidate implementation, step by step, synchronously. If a discrepancy is detected, then Monolith prints the entire scenario and crashes. Otherwise, the scenario is abandoned after n instructions have been generated and executed, and Monolith moves on to another scenario.

During this process, every time a choice must be made, the source of “random” bits is accessed. Choices are made when Monolith chooses an operation or generates an argument for an operation. Thus, viewed from the outside, the complete executable program behaves like a black box that reads a stream of bits and responds (in a deterministic manner) by either terminating normally or crashing.

The goal of the testing process, then, is to feed this executable program with random data, or with adversely crafted data, in order to cause a crash.

Random testing can be performed by letting the executable program read data from the pseudo-device /dev/urandom. Grey-box mutation-based fuzzing can be performed by requesting the OCaml compiler to instrument the executable program with AFL-specific instructions and by letting AFL drive its execution. A blog post by Rebours [12] explains the workflow, and the scripts in Makefile.monolith, which is bundled with Monolith, automate most of it.

When the executable program is about to crash, it first prints a scenario on its standard output channel. This is a sequence of OCaml instructions that leads to a discrepancy between the reference implementation and the candidate implementation. In principle, the user can copy and paste this scenario directly into the OCaml REPL in order to reproduce the problem.

As a rough indication of performance, when driven by AFL and running under Linux on a desktop machine, we observed about 10K executions per second in single core mode and about 5K executions per second and per hardware thread in multicore mode, all the way up to 32
hardware threads, for a total of 160K executions per second. These figures are of course very much dependent on the library under test, on the reference implementation, on the amount of fuel that is given to Monolith, and on the operating system and hardware.

3.2 Implementation

Monolith’s implementation is relatively straightforward: all in all, it is a few thousand lines of OCaml code. The type \((\_, \_)\) \texttt{spec} is a generalized algebraic data type (GADT). The engine simultaneously generates a well-typed straight-line scenario and executes this scenario both under the reference implementation and under the candidate implementation. Each instruction in this scenario is of the form \(\text{let } p = e\), where the expression \(e\) is built by choosing an operation and constructing a series of arguments for it, while the pattern \(p\) is obtained by deconstructing the result of executing this operation. Both of these processes, namely construction of arguments and deconstruction of results, are driven by the structure of the specification provided by the user for this operation.

All of the variables introduced by a pattern \(p\) have abstract type. Indeed, a result whose type is concrete does not need to be bound to a variable; it can be deconstructed, checked for validity and forgotten. A result whose type is abstract, on the other hand, cannot be validated. It must be bound to a variable, so it can later be used as an argument to another operation. Monolith internally maintains a runtime environment where every variable has abstract type and is bound to a dual value.

The fact that the scenario is generated and executed at the same time plays an important role in a few places. When a value of a sum type must be deconstructed, this value is available already, so its tag can be inspected, and a suitable pattern can be constructed. In other words, there is no need for generating case analysis constructs with multiple branches: one branch, the correct branch, is always enough. When a variable of a certain abstract type must be chosen among those that exist in the environment while respecting a certain precondition, the fact that every variable is already bound to a value is essential: this means that one can effectively test which choices satisfy the precondition and which do not.

4 Examples

We present a few examples of the use of Monolith. In each case, we give a specification and a reference implementation, but do not show the candidate implementation, which is irrelevant.

4.1 Nondeterministic Increasing-Sequence Generators

We wish to specify nondeterministic generators of integer sequences, whose API is as follows:

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

There is an abstract type \(t\) of generators. The function \(\text{create}\) returns a fresh generator. The function \(\text{next}\) queries a generator for the next integer number in the sequence that it represents. The function call \(\text{next } g\) must produce a number that is nonnegative and strictly greater than the number produced by any previous call \(\text{next } g\).

To test an implementation of this API, one can write the following main program, which declares the type \(t\) and its operations and runs Monolith:

```ocaml
let t = declare.abstract.type() in
declare "create" (unit ^> t) R.create C.create;
```
The specification \( t ^{?} \text{int} \), which is sugar for \( t ^{> \text{nondet int}} \), indicates that the operation \textit{next} is nondeterministic. According to this specification, whereas the candidate function \textit{C.next} has type \( C.t \rightarrow \text{int} \), the reference function \textit{R.next} has type \( R.t \rightarrow \text{int} \rightarrow \text{int} \). A reference implementation can be written as follows:

```ocaml
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
```

The reference implementation uses a piece of mutable state to record the last value produced by the candidate implementation. The function \textit{R.next} looks up this state in order to determine whether \textit{candidate} is an acceptable result. If it is, then it updates this state and returns \textbf{Valid} \_; otherwise, it returns \textbf{Invalid}.

### 4.2 Semi-Persistent Arrays

Semi-persistent arrays [6] resemble persistent arrays, which we have encountered in the prologue. However, they offer a more restrictive API, which disallows certain access patterns. For this reason, they can be implemented more efficiently. Their API is as follows:

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

There is an abstract type \( 'a \ t \) of arrays of elements of type \( 'a \). The operation \textit{make} \( n \ x \) creates a fresh array of length \( n \) that holds \( n \) times the element \( x \). The operation \textit{length} \( a \) returns the length of the array \( a \). The operation \textit{get} \( a \ i \) reads the array \( a \) at offset \( i \), while \textit{set} \( a \ i \ x \) updates the array \( a \) at offset \( i \) with the value \( x \), producing a new array, which is considered a \textit{child} of \( a \). At any point in time, the arrays that have been created so far, equipped with the child relationship, form a forest. Within each tree, an array is considered \textit{valid} if and only if it is a (reflexive, transitive) parent of the array that was most recently accessed via \textit{get} or \textit{set}. Whereas \textit{length} can be applied to any array, \textit{get} and \textit{set} must be applied to a valid array.

To test an implementation of this API, one can write the following main program:

```ocaml
let elt = sequential()
and t = declare_abstract_type() in
declare "make" (lt 16 ^> elt ^> t) R.make C.make;
decrare "length" (t ^> int) R.length C.length;
decrare "get" (R.valid % t ^>> fun a -> lt (R.length a) ^> elt) R.get C.get;
decrare "set" (R.valid % t ^>> fun a -> lt (R.length a) ^> elt ^> t) R.set C.set;
main 5
```

We use \( lt \ j \) as a short-hand for \textit{interval 0 j}. The operation \textit{length}, which can be applied to any array, does not have a precondition, while \textit{get} and \textit{set}, which must be applied to a valid array, have the precondition \( R.\text{valid} \). This prevents Monolith from generating an access pattern that violates the semi-persistent array API.
There remains to provide a reference implementation of semi-persistent arrays, which must offer not only the four operations \texttt{R.make}, \texttt{R.length}, \texttt{R.get}, and \texttt{R.set}, but also the runtime validity test \texttt{R.valid} that has been used above as part of the specification. One of the simplest possible implementations, inspired by Conchon and Filliâtre’s paper [6], is to associate with each tree a stack of currently valid arrays. This stack represents a path from the most-recently accessed array in this tree up to the root of this tree. To test whether an array is valid, one searches for it in the stack. To invalidate the strict descendants of a valid array, one truncates the stack after this array. Both \texttt{get} and \texttt{set} perform such an invalidation step.

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

(* Stack operations. *)
let peek stack =
  match !stack with
  | [] -> assert false
  | x :: _ -> x

let pop stack =
  match !stack with
  | [] -> assert false
  | _ :: xs -> stack := xs

let push x stack =
  stack := x :: !stack; x

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

let invalidate_descendants spa =
  while peek spa.stack != spa do
    pop spa.stack
  done

(* Operations on semi-persistent arrays. *)
let make n x =
  let data = Array.make n x in
  let stack = ref [] in
  push { data; stack } stack

let length spa =
  Array.length spa.data

let get spa i =
  invalidate_descendants spa;
  Array.get spa.data i

let set spa i x =
  invalidate_descendants spa;
  let data = Array.copy spa.data and stack = spa.stack in
  Array.set data i x; push { data; stack } stack
```

4.3 Sek

We have used Monolith in conjunction with AFL in order to test Sek [4], a library that offers efficient implementations of ephemeral sequences, persistent sequences, and ephemeral iterators on both kinds of sequences. Sek publishes 4 abstract types and over 150 operations. Its data structures involve complex balancing invariants, shared mutable state, and a subtle ownership policy that determines when an object can be updated in place and when it must be copied. Sek itself represents about 6000 lines of nonblank noncomment lines of code. The reference implementation and the specifications together occupy about 1500 lines of code, that is, about 10 lines per operation on average. As Sek evolved over the course of several months, we found fuzzing extremely effective in uncovering bugs: in our experience, many bugs can be found in minutes, and some more can be found in hours. The fact that Monolith always produces a short scenario is invaluable. It is of course impossible to tell how many bugs in Sek were not found. Still, we can confidently say that testing has saved us a lot of trouble and embarrassment.

5 Limitations and Work-Arounds

User-defined algebraic data types Only a fixed collection of product types and sum types are known to Monolith (§2.1). There is no built-in support for tuple types of arity greater than two, for sum types other than \texttt{option} and \texttt{result}, or for algebraic data types. Such support can however be programmed by the user. Indeed, thanks to the combinators \texttt{fix, ifpol, map_outof,}
and map_into, the isomorphism between an algebraic data type and a sum of products can be declared and exploited.

The algebraic data type list, for instance, could be described in this way. However, such an approach would perhaps be too simple-minded. It would cause Monolith to construct a list by first flipping a coin so as to choose between the data constructors [] and :: and then, if :: has been chosen, by constructing an element and a sublist, thereby repeating the process. This approach would make it unlikely for long lists to be constructed. In practice, it seems preferable to generate a list by first selecting its length within a certain range, then generating as many elements as necessary. Fortunately, the core combinators that we have presented allow using this strategy on the construction side while sticking with the simple-minded approach on the deconstruction side.

Because lists are so common, Monolith does provide a list combinator (not shown in Fig. 1 or Fig. 2). This combinator could in principle be defined by the user outside Monolith.

**Functions as arguments** Monolith currently does not allow an operation to take a function as an argument: a specification of the form (a ^> b) ^> c is rejected. This is a consequence of the rules set forth at the beginning of §2: a function argument must be constructible, and a function is not constructible. It is not clear at this time how one might relax this restriction. Indeed, several of Monolith’s design principles seem incompatible with the desire of generating a function that satisfies a certain specification. The fact that Monolith generates code and executes it on the fly, the fact that it generates straight-line code only, and the fact that it generates code without a predetermined goal, all seem incompatible with the idea of generating code for a function with the goal of obeying a predetermined specification.

Fortunately, it is possible to some extent to work around this limitation. Using constructible to equip the function type a -> b with a custom generator is one possible workaround. Using map_outof to transform some other constructible type into the function type a -> b is another approach. Finally, one could also declare a -> b as an abstract type and equip it with one or more operations that construct values of this type. Although these approaches are limited to generating values in a strict subset of the type a -> b, they can be good enough.

In some cases, testing a higher-order function can be reduced to testing a well-chosen first-order function. For instance, to test a fold function that iterates over a collection, one could argue that it suffices to define the function elements (which constructs a list of all elements of the collection) in terms of fold, and to test elements. Indeed, no information other than a sequence of elements can be extracted out of fold.

**Functions as results** One could say that Monolith does not allow an operation to return a function as a result. Indeed, although a specification of the form a ^> b ^> c is accepted, it is treated as the specification of a function of arity two. Thus, an operation op whose specification is a ^> (b ^> c) is always applied to two arguments: Monolith does not perform partial applications. Furthermore, the specification a ^> (b1 ^> c1) + (b2 ^> c2) is rejected altogether. This is a consequence of the rules set forth at the beginning of §2: a function result must be deconstructible, and a function is not deconstructible.

Fortunately, these restrictions are superficial and can be worked around. A function type such as b -> c can be declared as an abstract type t and equipped with an identity operation id of type t ^> b ^> c. The operation op above can then receive the specification a ^> t. This yields the desired behavior: Monolith applies op to just one argument, binds the resulting

---

1The reader might remark that this fails to exercise some exotic scenarios, such as a scenario where fold is applied to a function that raises an exception, or a scenario where fold is applied to a function that invokes fold in a reentrant manner. This is true. It is up to the user to think about these scenarios and to decide whether, by not exercising them, she is likely to miss some bugs.
function to a variable, and (in subsequent operations) can apply this function several times. For convenience, Monolith offers a derived combinator \texttt{abstract} (not shown in Fig. 2) that does this on the fly. Thus, the operation \texttt{op} can be given the specification \texttt{a ^> \text{abstract} (b ^> c)}.

**Sequences** OCaml’s standard library defines the type \texttt{`a Seq.t} to represent on-demand sequences of elements of type \texttt{`a}. An element is produced only when the sequence is queried by a consumer. The type of sequences is not an abstract type: instead, it is a function type. It is therefore an example of a procedural abstraction \cite{13}.

Sequences can be given Monolith specifications by using the techniques described in the previous two paragraphs. The combinator \texttt{ifpol} allows specifying separately how to construct and how to deconstruct sequences. On the construction side, one can generate a sequence essentially in the same way as one generates a list. On the deconstruction side, one can declare a type \texttt{seq} of sequences as an abstract type and equip it with a \texttt{query} operation whose specification is \texttt{seq ^> option (element *** seq)}. This allows Monolith to demand elements one at a time, and also allows it to make multiple queries (possibly interleaved with other instructions) on a single sequence. This is appropriate when dealing with persistent sequences, which can be queried as many times as one wishes.

The treatment of affine sequences, which can be queried at most once, is slightly different. On the construction side, one must construct a sequence that fails at runtime (by raising an exception) if it is queried twice. This allows detecting a situation where an operation queries an affine sequence several times. On the deconstruction side, Monolith must be instructed that an affine sequence must not be queried twice, so as to prevent it from generating illegal usage scenarios. To do so, one must first map a sequence into a custom representation that supports not only querying a sequence, but also testing whether a sequence is valid (that is, whether it has not yet been queried). This custom representation of sequences (which the end user remains unaware of) is then declared as an abstract type \texttt{seq}, whose \texttt{query} operation is assigned the specification \texttt{valid % seq ^> option (element *** seq)}, where the precondition \texttt{valid} prevents Monolith from attempting to query a sequence twice.

Because sequences are commonly used, Monolith provides two functions \texttt{declare_seq} and \texttt{declare_affine_seq} (not shown in Fig. 2) that implement these ideas.

**Polymorphism** Monolith has no explicit support for polymorphic functions or parameterized types. To test a polymorphic function, one must test one or more monomorphic instances of it. Bernardy et al. \cite{2} discuss systematic ways of choosing a single monomorphic instance. To work with a parameterized type, such as \texttt{`a set}, one can choose to work with a specific monomorphic instance of it, such as \texttt{int set}. One can also define a parameterized combinator that represents this parameterized type: as an example, Monolith’s \texttt{list} combinator has type \texttt{(‘r, ‘c) spec -> (‘r list, ‘c list) spec}.

**Shrinking** Monolith does not currently perform any shrinking. AFL provides a tool, \texttt{afl-tmin}, which performs a certain amount of shrinking, and appears to give relatively good results. However, because this tool is unaware of the structure of the instruction sequences built by Monolith, its effectiveness remains limited. It would be interesting to build a custom shrinking algorithm into Monolith.

6 Related Work

Perhaps the best-known and most influential approach to testing in the functional programming community is QuickCheck \cite{8}. QuickCheck offers a library of generators that can construct various kinds of typed data. One popular style of use of QuickCheck is property-based testing, where the user states a universally quantified property, such as \texttt{remove x (insert x s) = s}, and
the system verifies that a finite number of randomly-selected instances of this property hold. To do so, QuickCheck must generate candidate-side data: if a set $s$ is represented in the candidate implementation as a balanced binary search tree, then QuickCheck must be programmed so as to generate such trees. This style is well-suited to a purely functional programming setting, where data structures are immutable and sharing is unobservable, but not to an imperative programming setting, where data structures are mutable and involve sharing. In such a setting, another style of use of QuickCheck, investigated by Claessen and Hughes [5], is model-based testing. This style involves generating sequences of instructions and verifying that the candidate implementation executes them in a manner that is deemed correct with respect to a model.

By design, Monolith encourages model-based testing: it generates and executes sequences of instructions, and requires the user to provide a reference implementation. It can generate concrete data, but does not generate inhabitants of abstract types: it obtains them only as the result of executing instructions. If desired, Monolith can also be used to test properties, such as $\text{remove } x \ (\text{insert } x \ s) = s$. To test this property, one embeds it inside a pseudo-operation of type $\text{element } \rightarrow \text{set } \rightarrow \text{unit}$, whose reference implementation does nothing, and whose candidate implementation tests the property and fails if the property is violated.

Dolan introduces support for AFL in the OCaml compiler and is the author of Crowbar [7], a library of data generators that facilitates property-based testing in conjunction with AFL. Crowbar does not offer any support for model-based testing.

Articheck [3] is somewhat more closely related to Monolith. Like Monolith, it requires the user to provide runtime descriptions of each operation, and executes operations in order to construct values of abstract types. However, it differs from Monolith in several key ways. It does not require a reference implementation: it searches for faults in a candidate implementation. It runs for a long time, potentially accumulating many values of each type, whereas Monolith investigates very short scenarios. When it detects a fault, it cannot print a short scenario that exhibits the fault, whereas Monolith can. Finally, it is not designed to be externally controlled by a fuzzer.

Klein et al. [10] perform random testing of Scheme programs that may involve higher-order functions and mutable state. They use higher-order contracts both as generators and as oracles: we use Monolith specifications for the same dual purpose. Whereas we perform code generation and execution at the same time, they separate these tasks. For this reason, code generation has access to the type of every variable, but not to its value, and code generation is performed with the goal of constructing a value of a certain predetermined type. On the positive side, this allows generating an inhabitant of a function type, and allows the generation process to involve backtracking search. On the negative side, this requires exploring a huge search space, and requires generating conditional instructions: this is the case, in particular, when a pre- or postcondition must be met, and when a value of a sum type must be deconstructed.

7 Conclusion

We have presented Monolith, a powerful tool for testing an OCaml library in isolation. Monolith requires a reference implementation and a concise specification of each operation. It supports random testing and fuzzing. Monolith’s combinators allow constructing a runtime description of the logical relation that connects the reference implementation and the candidate implementation. Monolith’s engine is both a program generator and a dual tagless interpreter: it generates a scenario and runs it under two distinct interpretations of the library’s abstract types and operations. Although it is by now very common to use GADTs to express unary runtime type descriptions and implement tagless interpreters, this may be the first example of using a GADT to express a binary logical relation and implement a dual tagless interpreter.
References


