Fail Faster

Staging and Fast Randomness for High-Performance PBT

ANONYMOUS AUTHOR(S)

Property-based testing (PBT) relies on generators for random test cases, often constructed using embedded domain specific languages, which provide expressive combinators for building and composing generators. The effectiveness of PBT depends critically on the speed of these generators. However, careful measurements show that the generator performance of widely used PBT libraries falls well short of what is possible, due principally to (1) the abstraction overhead of their combinator-heavy style and (2) suboptimal sources of randomness. We characterize, quantify, and address these bottlenecks.

To eliminate abstraction overheads, we propose a technique based on multi-stage programming, dubbed Allegro. We apply this technique to leading generator libraries in OCaml and Scala 3, significantly improving performance. To quantify the performance impact of the randomness source, we carry out a controlled experiment, replacing the randomness in the OCaml PBT library with an optimized version. Both interventions exactly preserve the semantics of generators, enabling precise, pointwise comparisons. Together, these improvements find bugs up to 13× faster.

1 Introduction

Property-based testing is a software testing technique that uses random test inputs to validate logical specifications [13]. A recent study on PBT usage in industry [21] shows that many practitioners run their property-based tests very frequently, and with a very short time budget. The faster the tests fail, the better.

One important opportunity for performance improvement is the *generators* that produce random inputs to the properties under test. These generators are written with the help of *generator libraries*, usually expressed as embedded domain-specific languages (eDSLs) that provide combinators for building and composing generators.

However, careful measurements show that the performance of existing generator libraries falls far short of what is possible, for two reasons. First, the high-level design approach followed by generator combinator libraries introduces layered abstractions that are difficult for compilers to optimize. Second, the calls a generator makes to its randomness library can constitute a large proportion of its run-time, magnifying the cost of an inefficient implementation. In this paper, we characterize, quantify, and address these issues.

We address the overhead of many common generator abstractions using *multi-stage programming* (or *staging*), a well-studied technique for building fast DSLs "without regret" [53]. We refer to this approach as Allegro. To demonstrate its generality, we implement staged generator DSLs in two strict functional languages: OCaml and Scala 3. In OCaml, we build AllegrOCaml, based on `Base_quickcheck`—a high-quality PBT library in OCaml, authored by software engineers at the trading firm Jane Street [61]. In Scala 3, we build ScAllegro, based on ScalaCheck—the language's standard PBT library [2].

We isolate and quantify the performance benefits of faster randomness, showing that a DSL's choice of randomness library has a dramatic effect on its performance. Based on the observation that the OCaml implementation of `Base_quickcheck`'s randomness library is slow in a way that is simple to fix, we perform a controlled experiment, comparing fast and slow versions of the library to analyze the impact of faster randomness libraries on generator performance.

2025. ACM XXXX-XXXX/2025/3-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Both interventions preserve semantics on the nose: given the same random seed, generators written using AllegrOCaml or ScAllegro produce exactly the same sequence of values as equivalent generators using `Base_quickcheck` or ScalaCheck. This semantic equivalence enables pointwise—as opposed to distributional—comparisons of bug-finding effectiveness, ensuring that, if an Allegro generator finds bugs faster, this is attributable *solely* to the staging and randomness library interventions, not lucky choice of seeds.

We evaluate the optimizations with a series of case studies in each generator library, and show that the Allegro generators run faster than their unstaged equivalents. In AllegrOCaml, they run up to 7× faster, and in ScAllegro, up to 13× faster. Using our improved version of `Base_quickcheck`'s randomness library in AllegrOCaml, we see even greater gains—more than 12×—showing that both staging and fast randomness play distinct and complementary roles in improving generator performance. Further, we show that AllegrOCaml generators find bugs faster by running our case studies in the Etna platform, which uses generated values as inputs to a buggy system and measures how quickly different generators can detect the bugs. In Etna, AllegrOCaml generators find bugs up to 2.5× faster on average than their `Base_quickcheck` counterparts. With fast randomness, the average speedup rises to 3.8×, with some cases exceeding 13×.

In summary, we show that PBT generator DSLs incur significant performance costs across languages and present two interventions that significantly improve their efficiency while preserving their idiomatic style. Concretely, we offer the following contributions:

(1) We identify two key sources of inefficiency in PBT generator libraries—abstraction overhead and choice of randomness library—both of which significantly impact performance (Section 2).
(2) We present Allegro, a staging technique that eliminates the abstraction overhead of generators. We apply Allegro to standard generator libraries in both OCaml and Scala 3, showcasing its generality (Section 3).
(3) We demonstrate that writing generators using Allegro and fast randomness libraries yields substantial performance improvements, both separately and in combination, and we show that these performance improvements extend to significantly improved bug-finding speed (Section 4).

Section 5 discusses how the Allegro technique could be applied to PBT libraries in other languages—Racket, F#, Haskell, and Rust. Sections 6 and 7 present related and future work.

2 What are Generator Libraries, and Why are They Slow?

Property-based testing [14] is an approach to software testing that centers around executable specifications of programs called *properties*. For example, if a programmer wants to test an invariant of a binary search tree (BST) implementation that they are working on, they may write a property like

```
prop_insert_invariant t x = isBST t ==> isBST (insert x t)
```

to check that for any tree `t` and value `x`, if `t` is already a valid BST then inserting `x` into it also yields a valid BST. Once a developer has a property, they test that property by executing it on hundreds or thousands of random test inputs. These test inputs are usually produced by a *generator*—a program written in some domain-specific language (DSL) that allows the developer to express precisely how values should be sampled.

The standard design for a generator DSL, introduced in Haskell's QuickCheck library [13] and copied in dozens of other frameworks, is via an embedded *monadic* language [44]. We use the syntax and types from OCaml's `Base_quickcheck` library, which is presented in Figure 1, but similar DSLs can also be found in languages like Haskell, Scala, Python, and many more. The library

```
module Bq : sig
  type 'a t
  val gen_int : int -> int -> int t
  val return : 'a -> 'a t
  val bind : 'a t -> ('a -> 'b t) -> 'b t

  val weighted_union : (int * 'a t) list -> 'a t

  val size : int t
  val with_size : int -> 'a t -> 'a t

  val fixed_point : ('a t -> 'a t) -> 'a t
  ...
end
```

Fig. 1. Some functions from the API of the Base_quickcheck generator DSL.

provides some basic generators, for example `gen_int` for generating random integers in a range, along with the functions `return` and `bind`. The generator `return x` is the constant generator, always generating the value `x`. Running a generator `bind g k` runs the generator `g`, producing a value `a`, and then runs the generator `k a`.

Together, these three functions are the bare minimum for constructing arbitrary random data generators: `gen_int` provides a base source of randomness, and `return` and `bind` allow generators to be composed to create larger, more complex generators. Figure 2a shows a generator built with these operations; it first samples an int between 0 and 100, names it x, samples another between 0 and x, and then returns the pair of them.

```
let int_pair : (int * int) Bq.t =
  Bq.bind (Bq.gen_int 0 100) (fun x ->
    Bq.bind (Bq.gen_int 0 x) (fun y ->
      Bq.return (x,y)))
```

(a) A simple generator using `bind` explicitly.

```
let int_pair : (int * int) Bq.t =
  let%bind x = Bq.gen_int 0 100 in
  let%bind y = Bq.gen_int 0 x in
  return (x,y)
```

(b) An equivalent generator using the macro for `bind`.

Fig. 2. Simple monadic generators for a pair of ordered integers.

Most languages in which monadic APIs are common expose some sort of syntactic sugar for them. In OCaml, [1], this looks like `let%bind x = e in e'` which desugars to `bind e (fun x => e')`. Figure 2b shows the same generator, written the monadic syntax.

Generator libraries also include other functions that make generator construction easier; some examples of these are included in Figure 1. The `weighted_union` function is a particularly well-used one: it makes a weighted choice between different generators, allowing the developer to combine different sub-generators into a single program and tune the data distribution. Also important are functions like `size` and `with_size` that are used to control the sizes of generated values and `fixed_point` that is used to define recursive generators.

[1] OCaml actually has a few ways to implement monadic syntax; this is the one provided by Jane Street's libraries.

```
module Bq = struct
  type 'a t = int -> SR.t -> 'a

  let return (x : 'a) : 'a t = fun _ _ -> x

  let bind (g : 'a t) (k : 'a -> 'b t) : 'b t =
    fun size random ->
      let a = g size random in
      (k a) size random

  let gen_int (lo : int) (hi : int) : int t =
    fun _ random -> SR.int random lo hi
end
```

Fig. 4. Internals of a Monadic Generator eDSL

The generator in Figure 3 uses all of these features. It uses `fixed_point` to define a recursive generator that reads the current value of `size` to determine how to generate a tree. It uses `weighted_union` to make a random choice between an empty tree and a node, choosing a node with weight proportional to the current size and choosing a leaf otherwise. When generating a node, it uses `with_size` to reduce the value of the size parameter for future iterations.

```
let tree_of g = fixed_point (fun rg ->
  let%bind n = size in
  weighted_union [
    (1, return E);
    (n,
      let%bind x = g in
      let%bind l = with_size (n / 2) rg in
      let%bind r = with_size (n / 2) rg in
      return (Node (l,x,r)))
  ])
```

Fig. 3. A generator using a variety of convenience functions.

2.1 Abstraction Overhead of Generator DSLs

Just how large is the abstraction overhead of monadic generator DSLs, and where does it come from? Figure 4 shows the internals of (a simplified version of) `Base_quickcheck`. A generator `'a Bq.t` is just a function of type `int -> SR.t -> 'a`, taking an `int` representing the current size parameter and a random seed `SR.t`, and returning a generated value `'a`. The random seed `SR.t` from `Base_quickcheck`'s randomness library called `Splittable_random`, henceforth aliased in code as SR. It is an invariant of the generator library that every function of this type is deterministic: for a fixed size and seed, it will always return the same value, so all of the randomness in testing comes from varying the initial seed. The monad functions `return` and `bind` are defined in the usual way: for an instance of the reader monad [44]: `return` ignores the size and seed and returns its argument, while `bind` runs g and passes the result to k. The `gen_int` combinator simply calls out to the randomness library `Splittable_random`, aliased as SR here. Different generator DSLs use variations on this basic design, but the basics are the same across the board.

Just how much run-time overhead does this monadic abstraction introduce? To illustrate, let's return to our running example of a constrained pair of integers, written in both `Base_quickcheck` and ScalaCheck. Figure 6 shows two versions of the generator, written in both languages. The first versions (`int_pair` in `Base_quickcheck` and `intPair` in ScalaCheck) are written with the monadic generator combinators from their respective libraries. The second versions (`int_pair_inlined` and intPairInlined) are semantically identical to the first, but have been rewritten by (1) inlining all generator combinator definitions, and then (2) repeatedly reducing simplifiable terms like `(fun x -> e) e'` where an anonymous function is defined and then immediately called, to `let x = e' in e`.

Library	Generator	Average Time per Generation (ns)
Base_quickcheck	int_pair	70
Base_quickcheck	int_pair_inlined	35
ScalaCheck	intPair	458
ScalaCheck	intPairInlined	266

Fig. 5. Microbenchmarks of generators in Base_quickcheck and ScalaCheck. Average over 10000 generations with random seeds and a fixed size.

The performance impact of this simplification is large (Figure 5). In both languages, the inlined version takes on average half as much time to generate a single pair of ints. Microbenchmarks of more realistic generators (see Section 4) show an even more dramatic performance boost. Because the inlined versions of the generator are identical to the un-inlined versions except for mechanical, semantics-preserving transformations, this performance difference is attributable solely to the different machine code generated by the compiler (and not lucky choice of random seeds).

```
let int_pair : (int * int) Bq.t =
  let%bind x = (Bq.gen_int 0 100) in
  let%bind y = (Bq.gen_int 0 x) in
  Bq.return (x,y)

let int_pair_inlined : int -> SR.t -> int * int =
  fun _ sr ->
    let x = SR.int sr ~lo:0 ~hi:100 in
    let y = SR.int sr ~lo:0 ~hi:x in
    (x,y)
```

(a) Simplifying a Generator in OCaml

```
def intPair : Gen[(Long,Long)] = for {
  x <- Gen.choose(0,1000)
  y <- Gen.choose(0,x)
} yield (x,y)

def intPairInlined : (Gen.Parameters, Seed) =>
(Option[(Long,Long)],Seed) = {
 (p,seed) =>
  val (x,seed2) = chLng(0,1000)(p,seed)
  x match {
    case None => (None,seed2)
    case Some(x) =>
      val (y,seed3) = chLng(0,x)(p,seed2)
      y match {
        case None => (None,seed)
        case Some(y) => (Some(x,y),seed3)
      }
    }
  }
```

(b) Simplifying a Generator in Scala

Fig. 6. Simplifying Generators

It might be surprising to readers that this dramatic overhead exists—shouldn't the compiler perform this simple optimization? Compilers of effectful and strict functional languages (including JIT compilers in the case of Scala 3) do in fact have heuristics to determine if and when to perform this particular kind of simplification.[2] However even in cases as simple as Figure 6, the indirection of `return` and `bind` causes these heuristics to not fire. The story is even worse for recursive generators, as the heuristics are necessarily even more conservative for optimizing recursive functions.

More complex generators suffer further performance penalties due to *closure allocation*. In cases where the compiler cannot statically eliminate it, running a monadic bind allocates a short-lived closure for the continuation and then immediately jumps into it. In strict functional languages like OCaml and Scala, each individual closure allocation is relatively cheap; but doing lots of allocation in a generator is very expensive because each allocation brings us closer to the next costly GC pause. This effect is magnified in recursive generators: each iteration through the recursive loop re-allocates closures for `bind`s, so the amount of allocation per generated value scales linearly with the number of recursive generator calls.

Overhead of Choice Combinators. Like `return` and `bind`, combinators like `weighted_union` incur a performance penalty at run time. Aside from the previously-discussed issue that compilers cannot see through the abstraction boundary to optimize these programs, choice combinators like `weighted_union` come with their own particular abstraction overhead.

In practice, `weighted_union` is (almost[3]) always called with an explicitly constructed list, as in `weighted_union [(w1,g1); (w2,g2); (w3,g3)]`. This is because the most common use case for `weighted_union` is to choose between one of the different constructors of an algebraic datatype, the options for which are always known. This list is allocated at the call site and then never needed after the call to `weighted_union` returns. Since the elements of the list and its length `n` are known, a compiler could in principle unroll the loops in `weighted_union` to depth `n` and specialize the function at each call site to avoid allocating the list. Unfortunately, (almost[4]) no compilers perform this kind of optimization. This allocation (or rather, its tendency to cause GC pauses)—as well as the cost of running the code to traverse arbitrary lists compared to unrolled loops—has a significant impact on performance.

Last, many PBT libraries—including both `Base_quickcheck` and ScalaCheck—implement their `weighted_union` combinators in ways that are asymptotically more efficient but slower in common cases than a more naive algorithm. Weighted union uses the Fitness Proportionate Selection algorithm [23], which (1) samples a number r between 0 and the sum of the weights, and then (2) finds the first generator for which the cumulative weights in the list before it exceeds r. This second part can be accomplished in $O(\log_2 n)$ time by a binary search. However, since the lists are short in practice, a linear scan is almost always faster. Moreover, both `Base_quickcheck` and ScalaCheck allocate auxiliary data structures (an array in `Base_quickcheck` and a BST in ScalaCheck) to perform this search, which incurs further run-time overhead.

2.2 Inefficient Randomness Libraries

The core of any PBT generator library is a source of randomness. Different PBT libraries use different randomness libraries implementing different algorithms.[5] Following the original Haskell QuickCheck implementation, `Base_quickcheck` uses the SplitMix algorithm [59], implemented by `Splittable_random`. Meanwhile, ScalaCheck uses the JSF algorithm [3].

[2]As we discuss in Section 5, purity means that Haskell is a slightly different story. GHC can and often does transformations of this form.

[3]There are some generators where `weighted_union` is passed a list which was itself the result of a generator: the well-typed STLC term generator used in Section 4 is an example of this.

[4]GHC is a notable exception here, performing sophisticated list fusion optimizations [20]

[5]The common term for such an algorithm or library is a "Random Number Generator" (RNG). We will avoid this term and instead say "randomness library" to avoid confusing RNG implementations with the PBT generator libraries that use them.

The randomness library is the hottest part of the hot path. Indeed, even basic generators—like ones generating a single `int` or `float` uniformly within a range—can sample *unboundedly many* random numbers, since they usually use versions of rejection sampling [67] to find a value within the range. Moreover, generator combinators like `list` usually make $O(n)$ calls to the `int` or `float` generators. Because of this, the speed of a single sample matters a great deal. Unfortunately, while existing PBT libraries by and large make sensible choices for their randomness libraries, they are not chosen with performance in mind, leading to worse bug-finding power than what is possible.

For example, significantly faster algorithms than SplitMix or JSF exist, such as the Lehmer algorithm [51], WyRand [72], and the xorshiro family of algorithms [7]. These all run between 1.2-1.6x faster per byte than SplitMix in microbenchmarks [39]. Plenty of other known optimizations on top of algorithm choice could also be implemented, including pipelined or even ahead-of-time sampling.

Of course, simply arguing that the randomness library is on the hot path for PBT does not guarantee that a faster sampling leads to measurably faster generation; for that, we need an experiment. The most obvious experiment is to simply swap out the randomness library of either `Base_quickcheck` or ScalaCheck one of the aforementioned faster algorithms. But, as discussed previously, AllegrOCaml and ScAllegro are designed to be 100% semantically equivalent replacements for their unstaged counterparts, ensuring that any bug-finding speedups are solely attributable to generator performance and not lucky seeds. Choosing a different randomness library would break this property, making it much more challenging to assess the performance compared to the baseline. To get around this, we exploit a coincidence: `Splittable_random`—the OCaml implementation of SplitMix that `Base_quickcheck` uses—is slow in a way that can be improved *without* changing the algorithm. In particular, due to implementation details related to the OCaml garbage collector, values of the OCaml type `int64` are not machine words, but rather *pointers* to machine words. This means that *all* `int64` operations (both arithmetic and bitwise) must allocate memory cells to contain their output, which has a significant performance benefit. By building a version that uses much faster "unboxed" 64-bit integer arithmetic, we (a) demonstrate how much faster bugs can be found just by using a more performant randomness library, and (b) explore which generators benefit the most from faster randomness libraries.

3 How Can I Make My Generator Library Faster?

With a sense of two main inefficiences in generator libraries, we set out in this section to investigate solutions. We begin with a quick tour through multi-stage programming (Section 3.1), then incrementally build AllegrOCaml, layering on features and functionality (Sections 3.2 through 3.7). Last, we present a controlled experiment demonstrating how a faster randomness library leads to failing faster (3.8).

3.1 Background: Multi-Stage Programming

In multi-stage programming (or simply "staging"), programs execute in multiple stages, with each stage producing code to be run in the next. For the purposes of this paper, we just need two stages: compile time and run time.

Staging eDSLs. One of the primary uses of staging is in embedded DSL construction [53, 56, 65]. While embedded DSLs are powerful tools, they have a well-known drawback: the functional abstractions used to build eDSLs tend to prevent compilers from generating efficient machine code. This is *abstraction overhead* [11, 45]: the layers of abstraction that make eDSLs pleasant to use are precisely what prevents them from being fast. Indeed, the root causes of many of the performance issues we discovered in Section 2 are not unique to generator DSLs, but pervasive across eDSLs.

In light of this issue, staging is often used as a lightweight compiler for DSLs. The compile-time evaluation stage transforms the DSL code, eliminating abstractions to produce code that the host language compiler can generate fast machine code for. This recipe has been used to great effect to stage eDSLs for stream processing [30, 45], parser combinators [27, 34, 69, 71], and query processing [4].

Many languages have some degree of staging functionality, including Scala 3 [1], Haskell [57], Racket [18], OCaml [28, 70], and Java [68]. For this paper, we have implemented staged PBT libraries in both OCaml (using MetaOCaml [28]) and Scala 3 (using `scala.quoted`). All staged code presented in the body of this paper is AllegrOCaml code written in OCaml; ScAllegro is very similar. We discuss the potential for staged generator DSLs in other languages in Section 5.

Staging in MetaOCaml. MetaOCaml's staging functionality is exposed through the type `'a code`. A value of type `t code` is an OCaml term of type `t`. Values of `code` type are introduced by *quotes* (written `.<...>.`). Quotes delay execution of a program until run time. For example, the program `.< 5 + 1 >.` has type `int code`. Note that this is not the same as `.< 6 >.`: because brackets delay computation, the code is not *executed* until the next stage (run time). Values of type `code` can be combined using an *escape*, written `.~(e)` (or just `.~x`, when `x` is a variable). Escaping lets you take a value of type `code` and "splice" it directly into a quote. For example, the program `let x = .<1 * 5>. in .< .~(x) + .~(x) >.` evaluates to `.<(1 * 5) + (1 * 5)>.` MetaOCaml enforces correct scoping and macro hygiene, ensuring that variables are not shadowed when open terms are spliced in under binders.

The power of staging for optimizing away abstraction overheads comes from defining functions that accept and return `code` values. A function `f : 'a code -> 'b code` takes a program computing a run-time `'a` and transforms it into a program computing a run-time `'b`. In particular, because `f` itself runs at compile time, the fact that the programmer called `f` does not matter at run time—the abstraction that `f` defines has been eliminated. A code-transforming function `'a code -> 'b code` can also be converted *code for a function*—a value of type `('a -> 'b) code`—with the following program:

```
let eta (f : 'a code -> 'b code) : ('a -> 'b) code = .<fun x -> .~(f .<x>.)>.
```

This program is known as "The Trick" in the partial evaluation and multi-stage programming literature [29]. It returns a code for a function that takes an argument `x` and splices in the result of calling `f` on just the quoted `x`. For example, the following program

```
let is_even x = .< .~x mod 2 == 0 >. in
let succ x = .< 1 + .~x >. in
eta (fun x -> succ (is_even x))
```

reduces at compile time to `.< fun x -> (1 + x) mod 2 == 0 >.` By composing the two code-transforming functions together at compile time and only then turning them into a run-time function, the two functions are fused together. This is the basis of how staging is used to eliminate the abstraction overhead of DSLs. By writing DSL combinators as compile-time functions—and calling `eta` at the end on the completed DSL program—we can ensure that any overhead of using the combinators is eliminated before run time.

3.2 Design of a Staged Generator DSL

To build a staged version of a generator DSL, we want to rewrite the the generator combinators to do as much as possible at compile time. The compile-time stage will then produce simplified code, free of any DSL abstraction, that can be compiled and run with different sizes and seeds. Our job is thus to carefully bisect the DSL, determining which inputs to generator combinators are known statically (and can be part of the compile time stage) and which parts are only known at run time (and must be treated as code). In the staging literature, this task is known as "binding-time analysis," [26].

The crux of our binding time analysis[BCP: the wording here is misleading: it feels like we're implying we've *implemented* a binding-time analysis...] is that the *only* parts of a generator that are not known at compile time are the random seed and size parameters. In practice, generators themselves are entirely known at compile time. This leads us to define our library's generator type `'a Gen.t` as

```
'a Gen.t  =  int code -> SR.t code -> 'a code.
```

That is, a `Gen.t` is a compile-time function from dynamically known size and seed to dynamically determined result.

The basic functionality of the staged monadic generator DSL can be found in Figure 7.

```
module Gen = struct
  type 'a t = int code -> Random.t code -> 'a code

  let return (cx : 'a code) : 'a t = fun size random -> cx

  let bind (g : 'a t) (k : 'a code -> 'b t) : 'b t =
    fun size random ->
      .<
        let a = .~(g size random) in
        .~(k .<a>. size random)
      >.

  let int (lo : int code) (hi : int code) : int t =
    fun size random ->
      .< SR.int .~random .~lo .~hi >.

  let to_bq (g : 'a code Gen.t) : ('a Bq.t) code =
    .<
      fun size random -> .~(g .<size>. .<random>.)
    >.
end
```

Fig. 7. Basic Staged Generator Library

The constant generator `return` runs at compile time. Given `cx : 'a code`, the code for an `'a`, it returns the generator that ignores its `size` and `random` arguments and simply runs[BCP: returns the code for?] `cx`. Similarly, `bind g k` sequences generators by passing the result of running the generator `g` to the continuation `k`. However, instead of getting access to the particular value generated by `g`, the continuation `k` only gets access to code for the value sampled from `g`: we know that at run time the code `g` generates will produce *some* `'a`, but at compile time we cannot inspect the value. Operationally, bind takes code for the size and seed and returns code that (1) let-binds a variable `a` to spliced-in code of `g` and then (2) runs the spliced-in continuation `k`. Both function applications `g size random` and `k .<a>. size random` run at compile time. `Gen.int` is the generator that samples an int from the randomness library. Given any size and random seed, it returns a code block that calls `SR.int` with that random seed. Because the lower and upper bounds might not be known at compile time—they may themselves be the results of calling `Gen.int`—the arguments `lo` and `hi` are of type `int code` and get spliced into the code block as arguments to `SR.int`. Lastly, `to_bq` turns a staged generator into code for a normal `Base_quickcheck` generator. This is just a 2-argument version of "The Trick" from Section 3.1.

Returning to our running example, Figure 8 shows the int-pair generator written with the staged `Gen.t` monad, as well as the inlined code that results from calling `Gen.to_bq` (changing some identifier names for clarity). The code generated is identical to the manually inlined version from Section 2.

```
let int_pair_staged : (int * int) Gen.t =
  Gen.bind (Gen.int .<0>. .<100>.) (fun cx ->
    Gen.bind (Gen.int .<0> cx) (fun cy ->
      Gen.return .<(.~cx,.~cy)>.
    )
  )

let int_pair : (int * int) Bq.t code = Gen.to_bq int_pair_staged
(* int_pair = .< fun size random ->
                 let x = SR.int random 0 100 in
                 let y = SR.int random 0 x in
                 (x,y)
               >.   *)
```

Fig. 8. Pairs of Ints, Staged

3.3 Staging Combinators

In Section 2, we noted that generator combinators like `weighted_union` allocate lists in the hot path of the generator. Even though these lists are usually small—at most a few dozen elements—each allocation takes us closer to the next garbage collection, which is bad for performance.

This is an ideal opportunity to exercise another feature of staging: compile-time specialization. Since we almost always know the particular list of choices at compile time, a staged version of `weighted_union` can generate *different code* depending on the number of generators in the union. If we use weighted union on a compile-time list of generators `g1`, `g2`, and `g3`, we can emit code that picks between the generators without realizing the list at run time.

Figure 9 shows the code for such a staged weighted union. Crucially, it takes a *compile-time* list `weighted_gens` of generators and weights. The weights themselves might only be known at run time—it is common to use the current size parameter as a weight, for instance—so they are codes. Instead of building a data structure representing a histogram of the distribution described by the weights at run time and then traversing it, the compile-time `weighted_union` combinator generates a tree of `if`s, specialized to the list of weights known at compile time, that picks out the selected generator.

Figure 10 demonstrates a use of this staged weighted union. Given a list of (in this case constant) generators with weights "the current size parameter", `2`, and `1`, the generated code first computes the sum of these numbers, samples between `0` and the sum, and traverses the tree of three `if`s to find the correct value to return.

```
module Gen =
...
  let pick (acc : int code) (weighted_gens : (int code * 'a t) list) size random : 'a code =
    match weighted_gens with
    | [] -> .< failwith "Error" >.
    | (wc,g) :: gens' ->
      .<
        if .~acc <= .~wc then .~(g size random)
        else
          let acc' = .~acc - .~wc in
          .~(pick .<acc'>. gens' size random)
      >.

  let weighted_union (weighted_gens : (int code * 'a t) list) : 'a t =
    let sum_code = List.foldr (fun acc (w,_) -> .<.~acc + .~w>. ) .<0>. weighted_gens in
    fun size random ->
      .<
        let sum = .~sum_code in
        let r = SR.int .~random_c 0 sum in
        .~(pick .<r>. weighted_gens size random)
      >.
```

Fig. 9. Staged Weighted Union

```
let grades : char Bq.t = Gen.to_bq (
  Gen.bind size (fun n ->
    Gen.weighted_union [
      (n, Gen.return .<'a'>.);
      (.<2>., Gen.return .<'b'>.);
      (.<1>., Gen.return .<'c'>.);
    ]
  )
)
(*
.< fun size random ->
    let sum = size + 2 + 1 + 0 in
    let r = SR.int random 0 sum in
    if r <= size then 'a' else
      let r' = r - size in
      if r' <= 2 then 'b' else
        let r'' = r' - 2 in
        if r'' <= 1 then 'c' else failwith "Error"
>.
*)
```

Fig. 10. Use of Staged Weighted Union

3.4 Let-Insertion and Effect Ordering

Careful readers might note that the definition of `bind` (Figure 11a) was more complicated than one might expect. Why not define bind in the standard way for a reader monad (Figure 11b)? Unfortunately, the latter definition is wrong in our context as it leads to incorrect code being generated. For example, consider `Gen.bind' (Gen.int .<0>. .<1>.) (fun x -> Gen.return .<(. x,. x)>.)`. This generates the run-time code `fun size random -> (SR.int random 0 1, int SR.random 0 1)`, which is incorrect. Instead of generating a single integer and returning it twice, it samples two different integers. This matters because, as described in Section 1, AllegrOCaml and ScAllegro are intended to be equivalent to their unstaged counterparts.

```
let bind (g : 'a t) (k : 'a code -> 'b t) : 'b t =
    fun size random ->
      .<
        let a = .~(g size random) in
        .~(k .<a>. size random)
      >.
```

(a) `bind`, with a let-binding

```
let bind' (g : 'a t) (k : 'a code -> 'b t) : 'b t =
    fun size random -> k (g size random) size random
```

(b) `bind'`, the “standard” bind for the reader monad

Fig. 11. `bind`, two ways

In essence, the behavior of splice `.~cx` in a staged function `f(cx : 'a code) = ...` is to *copy* the entire block of code, effects and all. To ensure that the randomness effects of the first generator are executed only once but that the value can be used in the continuation multiple times, the correct `bind` let-binds the result of generation to a variable and then passes it to the continuation.

3.5 CodeCPS and a Monad Instance

Another subtle issue prevents the version of the library design discussed so far from being used as a drop-in replacement for an existing generator DSL: the types of `return : 'a code -> 'a Gen.t` and `bind : 'a Gen.t -> ('a code -> 'b Gen.t) -> 'b Gen.t` aren't quite right. For the type `'a Gen.t` to actually be a monad, these types cannot mention code. This is not just a theoretical issue; it is a significant usability concern. The syntactic sugar for monadic programming (`let%bind` in OCaml, `foreach` in Scala, do in Haskell, etc) that makes it smooth can *only* be used if return has type `'a -> 'a Gen.t` and bind has type `'a Gen.t -> ('a -> 'b Gen.t) -> 'b Gen.t`

To support convenient monadic programming, we need to adjust the type of `'a Gen.t` slightly. An initial attempt is to try `type 'a t = int code -> SR.t code -> 'a`. If we strip the code off the result type, the functions `return (x : 'a) = fun _ _ -> x` and `bind g k = fun size seed -> k (g size seed) size seed` have the proper types for a monad instance. Then, any combinators of type `'a Gen.t` before simply become `'a code Gen.t` with this new version.

However, this definition of bind doesn't have call-by value effect semantics, as discussed in the previous section. And because the type of `g size seed` is now just `'a` (not necessarily `'a code`), we cannot perform the let-insertion needed to preserve the CBV effects. To solve this problem, we turn to a classic technique from the multistage programming literature: writing our staged programs in continuation-passing style [8].

In Figure 12, following prior work [11, 32], we define the type `'a CodeCps.t = 'z. ('a -> 'z code) -> 'z code`: a polymorphic continuation transformer with the result type always in code.[6] The monad instance for this type is the standard instance for a CPS monad with polymorphic return

[6] This is an instance of the *codensity* monad [66]—a fact that deserves further investigation.

```
module CodeCps = struct
  type 'a t = { cps : 'z. ('a -> 'z code) -> 'z code }

  let return x = {cps = fun k -> k x}

  let bind (x : 'a t) (f : 'a -> 'b t) : 'b t =
    {cps = fun k -> x.cps (fun a -> (f a).cps k)}

  let run (t : ('a code) t) : 'a code = t.cps (fun x -> x)

  let let_insert (cx : 'a code) : 'a code t =
    {cps = fun k -> k .< let x = .~cx in .~(k .<x>.) >.}
end

module Gen = struct
  type 'a t = int code -> SR.t code -> 'a CodeCps.t

  let return (x : 'a) : 'a t = fun _ _ -> Codecps.return x

  let bind (g : 'a t) (f : 'a -> 'b t) =
    fun size random ->
      CodeCps.bind (g size random) (fun x -> (f x) size random)

  let int (lo : int code) (hi : int code) : int code t =
    fun size random -> let_insert .< SR.int .~random .~lo .~hi >.
end
```

Fig. 12. CodeCPS and the Final Gen Monad

type. In prior work, this type is often referred to as the "code generation" monad. This is because a value of type `('a code) CodeCps.t` is like an "action" that generates code: `CodeCps.run` passes the continuation transformer the identity continuation to produce a `'a code`. To avoid confusion with random data generators, we refer to this type as `CodeCps.t`. Most importantly, the `CodeCps` type supports a function `let_insert`, which, given `cx : 'a code`, let-binds `let x = .~cx`, and then passes `.<x>.` to the continuation. [HG: I'm getting lost in the inline code again]

We can then redefine our staged generator monad type to be `'a Gen.t = int code -> SR.t code -> 'a CodeCPS.t`, as shown in Figure 12. The (old) type `'a Gen.t` is now written as the (new) type `'a code Gen.t`, and this type change carries through all of our combinators. For example, `Gen.int` now returns `int code Gen.t`.

This gives us the best of both worlds. First, we get a monad instance for `'a Gen.t` with the correct types, which lets us use the monadic syntactic sugar of our chosen language. Moreover, we also get to maintain the correct effect ordering: effectful combinators like `Gen.int` do their *own* let-insertion, ensuring that a program like `Gen.bind Gen.int (fun x -> ...)` generates a let-binding for the result of sampling the randomness library. For example, `bind (int .<0>. .<1>.) (fun cx -> return .<(. cx,. cx)>.)` now correctly generates `.< fun size random -> let x = SR.int random 0 1 in (x,x) >.`. This design is less obviously correct, and does require some care. Rather than `bind` ensuring correct evaluation order once and for all, individual combinators must be carefully written to ensure that `'a code` values that contain effects are `let_inserted`. To validate the library, we built a PBT harness to compare staged generators to their `Base_quickcheck` equivalents over 1,000 random seeds. By differentially testing [41] a large suite of generators in this way, we gained confidence that AllegrOCaml is equivalent to `Base_quickcheck`.

3.6 Recursive Generators

Generating values of recursive datatypes requires recursive generators. Different generator DSLs support recursive generators differently. Some allow recursive generators to be defined as recursive functions, while others (including both `Base_quickcheck` and ScalaCheck) expose a fixed-point combinator to construct recursive generators. Given a step function that takes a "handle" to sample from a recursive generator call, it ties the knot and builds a recursive generator.

In our setting, letting programmers define recursive generators as recursive functions is out of the question. With staged programming, recursion must be handled with care: it is far to easy to accidentally recursively define an infinite `code` value and have the program diverge at compile time, when trying to write a `code` representing a recursive program. To this end, we develop a staged recursive generator combinator.[7], whose API is shown in Figure 13 The recursion API consists of an opaque type `'a handle`, and a function `recurse` to perform recursive calls. Programmers can then define recursive generators by `fixed_point`, which ties the recursive knot.

```
type 'a handle
val recurse : 'a handle -> 'a code Gen.t
val fixed_point : ('a handle -> 'a code Gen.t) -> 'a code Gen.t
```

Fig. 13. Staged Recursive Generator Combinator API

3.7 Staging Type-Derived Generators

Generators are traditionally handwritten, but some PBT libraries allow users to synthesize them automatically from type definitions. Type-derived generators are convenient—the derivation process requires no manual effort—but also limited: they are unable to account for constraints not encoded in the type. For example, they can generate arbitrary trees, but not binary search trees. When such constraints are present, type-derived generators are usually less effective than hand-crafted ones, since most generated values will be invalid.

The speed of generation becomes particularly important in this setting. In a generator that produces only valid values, only a subset of them will trigger a bug; in a type-derived generator, however, only a subset of generated values will be valid, and only a subset of *those* will find a bug. As a result, many more values must be produced, making it important to do so as quickly as possible.

The type-deriving algorithm follows a compositional pattern. Generators for complex types are synthesized by structurally composing generators for their subtypes: base types are mapped to primitive generators included in the generator library; product types such as tuples and records are handled by sampling each component using `bind` and aggregating the results; sum types, or variants, are generated using a `weighted_union` of the generators for each case; for recursive types, the entire generator is wrapped in a fixed-point combinator and a recursive handle is used as the generator for all recursive occurrences in the type. This compositional approach maps naturally to staged generation: to derive a staged generator, we replace each standard combinator with its staged counterpart. The derivation algorithm remains unchanged.

[7]We actually provide a more general API that allows programmers to define *parameterized* recursive combinators, of type `'r code -> 'a code Gen.t`, for any type `'r`.

Our implementation in OCaml uses the PPX (PreProcessor eXtension) system to synthesize staged generators from type definitions. However, any language that supports type-derived generators can implement a staged version using a similar implementation to the original. In Scala, for instance, the same strategy could be realized using type class resolution. In both settings, the result is a three-stage process: a metaprogram—via PPX or type classes—constructs a generator expression composed of staged combinators; this expression is evaluated at compile time to produce a specialized generator; and finally, the generator is executed at run time to produce values.

We implemented this in AllegrOCaml; it is not yet implemented in ScAllegro. We benchmark the results in §4.

3.8 Performance Opportunities in Randomness Libraries

As we discussed in Section 2.2, choosing an inefficient randomness library is another impediment to finding bugs fast. To demonstrate that faster random sampling can significantly impact bug-finding power, we use the controlled experiment suggested by OCaml's inefficient implementation of SplitMix. By replacing this relatively slow randomness library with a faster but semantically equivalent implementation, we can precisely quantify the bug-finding speedup that a better randomness library gives across a range of PBT scenarios.

Note that the performance intervention described here is OCaml-specific. In most other PBT frameworks, the randomness library used operates on machine integers, so this *particular* inefficiency does not exist. However, the insights that we will derive from this experiment in Section 4—about which kinds of generators benefit the most from faster randomness and how it leads to faster bug-finding— are applicable to all languages.

The precise details of the SplitMix algorithm [59] are not important for present purposes; the key fact is that all of its operations are defined in terms of arithmetic and bitwise operations on 64-bit integers. In OCaml, because of details related to the garbage collector, the 64-bit integer type `int64` is represented at run time as a *pointer* to an unscanned block of memory containing (among other things) a 64-bit integer [?]. This means that all operations that return an `int64` must allocate this block of memory, which has a significant impact on performance. A single call to one of the `Base_quickcheck` library functions—like generating an arbitrary integer—may call into the `Splittable_random` library times. Each call into `Splittable_random` may sample many times from the core SplitMix sampling routine `next_int64`, i.e., using rejection sampling to find a value within a range. Finally, each call to `next_int64` allocates 9 times, and each allocation brings us closer to the next garbage collection pause. While small allocations like these are *very* fast to perform and subsequently collect in OCaml[8] , we will see in Section 4 that they can still have a large performance impact on generators that spend most of their time sampling data. To avoid these allocations, we reimplement SplitMix in C and call out to it with the OCaml FFI. The C version of the library uses proper `int64_t` arithmetic, only boxing and unboxing integers at the call boundaries between OCaml and C code.[9]

4 Evaluation

We evaluate the raw generator performance and bug-finding speed of Allegro generators across a range of benchmarks. Our experiments compare generators built using our technique—with and

[8]The OCaml GC is a generational collector [43], and since these allocations are small and mostly very short lived, they will all be minor allocations, never to be promoted.

[9]Ideally in the future, one would not need to call out to C for this: the Jane Street bleeding-edge OCaml compiler has support for unboxed types [17], which (among other things) would let us implement a version of SplitMix that does not allocate, directly in OCaml. Unfortunately, the Jane Street branch of the compiler is incompatible with MetaOCaml, which we use to implement the metaprogramming discussed in the previous sections.

without our improved SplitMix ("CSplitMix")—against those implemented with existing generator libraries and their default randomness mechanisms. Specifically, we implement semantically equivalent staged generator libraries that replicate the behavior of `Base_quickcheck` in OCaml and ScalaCheck in Scala, allowing us to assess the effectiveness of our technique across different languages and runtime environments.

This section presents our experimental setup and addresses the following research questions:

- **RQ1**: Do generators written using our technique run faster than those written with regular generator combinators?
- **RQ2**: Do observed generation speedups translate to better bug-finding speed?

All experiments were run on a 64-bit Linux machine with 264 GB RAM and a 128-core Intel Xeon Platinum 8375C CPU, running Ubuntu 24.04.1 LTS. AllegrOCaml and all OCaml benchmarks were compiled with 4.14.1+BER MetaOCaml `ocamlopt`, the native code compiler for OCaml, using compiler flag `-O3`. The baseline OCaml generators were written with `Base_quickcheck` 0.16. ScAllegro and all Scala benchmarks were run on Scala 3.6.3 and OpenJDK 21.0.6, using ScalaCheck version 1.17 as the baseline. We used `core_bench` 0.16 [62] in OCaml and `jmh` 0.4.7 [48] for performance microbenchmarking. For assessing and comparing PBT techniques, we used the Etna platform [58].

4.1 Benchmarking Generator Speed

To answer **RQ1**, we microbenchmark generators, comparing generators written in AllegrOCaml to semantically identical ones in `Base_quickcheck` and generators in ScAllegro to those in ScalaCheck. We vary the choice of randomness library in our AllegrOCaml generators, using both `Base_quickcheck`'s default `splittable_random` and CSplitMix, as discussed in §3.8. Our test cases consist of generators for boolean lists, binary search trees (BSTs), and simply typed lambda-calculus (STLC) terms. We implement generators for these benchmarks using a variety of *strategies*, varying in structure and sophistication.

In AllegrOCaml, our strategies include: type-derived generators for BSTs and STLC terms, following the approach described in §3.7; two custom BST generators—one that builds a tree incrementally by repeatedly inserting values into an initially empty structure, and another that constructs the entire tree in a single pass by generating keys, values, and subtrees at each step; a boolean list generator that mirrors the single-pass BST strategy; and a generator for STLC terms that is correct by construction (i.e., it produces only well-typed terms).

For each strategy, we compare three treatments: our `Base_quickcheck` baseline; a AllegrOCaml version using `Base_quickcheck`'s randomness library, `splittable_random`; and a AllegrOCaml version using CSplitMix. For each treatment, we measure the time to generate a value (i.e., a BST, STLC term, or boolean list), using a random seed, varying generation sizes (10, 100, 1,000, 10,000).[10] We run each treatment for 5 seconds and compute the average generation time of a value produced in that interval.

Our results are summarized in Figure 14. The type-derived AllegrOCaml BST generator achieves speedups ranging from $1.30 - 1.38\times$, which increase dramatically, to $2.13 - 7.76\times$, when combined with CSplitMix. The insertion-based BST generator sees $1.18 - 1.22\times$ speedups with staging alone and $3.83 - 6.31\times$ when also using CSplitMix. The single-pass BST generator benefits more from staging, with speedups of $2.22 - 2.25\times$, rising to $9.05 - 9.82\times$ when combined with CSplitMix.

[10]The specific meaning of generation size is a domain-specific implementation detail—in a list generator, size might correspond to the desired length of the list, whereas in a tree generator it refer to number of nodes, number of leaves, depth of tree, etc. Regardless, our goal is to show that performance trends scale.

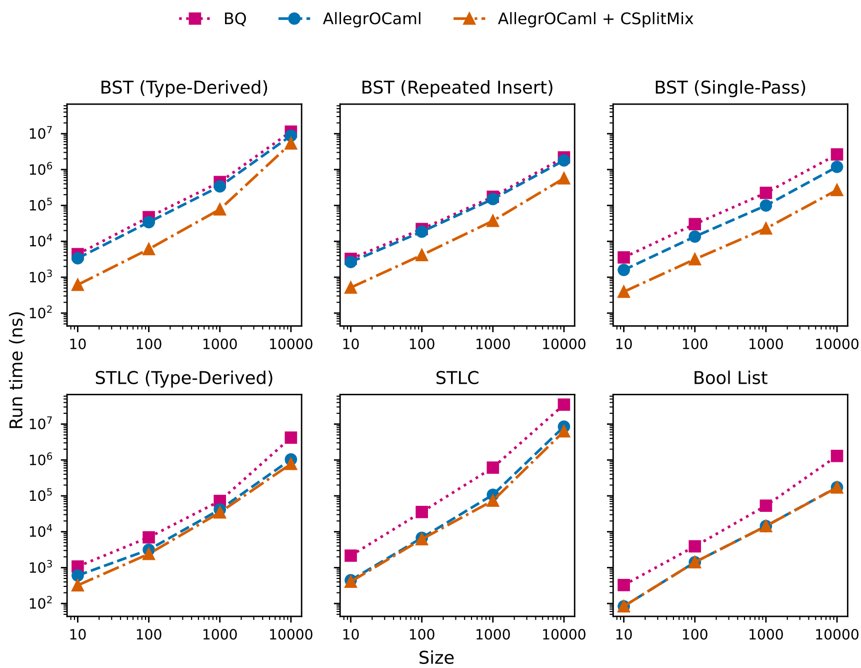

Fig. 14. Time to generate values of varying sizes using each AllegrOCaml strategy. Lower is better. Both axes are logarithmic. BQ is `Base_quickcheck`.

For STLC, staging accounts for larger performance gains, yielding 1.73 – 4.05× speedups for the type-derived STLC generator and 4.10 – 5.71× for the well-typed generator. Adding CSplitMix, these numbers increase to 2.90 – 5.39× and 5.33 – 8.33×.

The boolean list generator experiences 2.77 – 7.47× speedups with staging, and its performance changes only minimally when combined with CSplitMix (2.77 – 7.57×). This is likely because sampling booleans is cheap enough that the overhead of crossing the FFI barrier is comparable to that of generating values directly in OCaml.

The variation in speedups raises a natural question: what determines how much a generator benefits from staging, randomness library optimizations, or both? One likely explanation is that the two techniques address different performance bottlenecks: generators that sample more often benefit more from improved random sampling, while those that rely heavily on generator library combinators gain more from staging. To test this idea, we use measurable proxies that approximate a generator's reliance on its randomness and generator libraries. For the former, we count the number of times a generator invokes the SplitMix sampling routine `next_int64`. For the latter, we use the number of calls to `bind`—the central monadic generator function, extensively used both directly and within other combinators.

We select four strategies with disparate speedup profiles: insertion-based BSTs, single-pass BSTs, boolean lists, and well-typed STLC terms. To determine how often `bind` is invoked, we generate 10,000 values using a fixed generation size of 100 and record the average number of `bind` calls. To determine the number of random samples, we repeat the same test but use Intel ProcessorTrace [25] to capture a 4ms snapshot of processor activity. We then count the number of invocations of `next_int64` and average this over the number of generator calls in the trace.

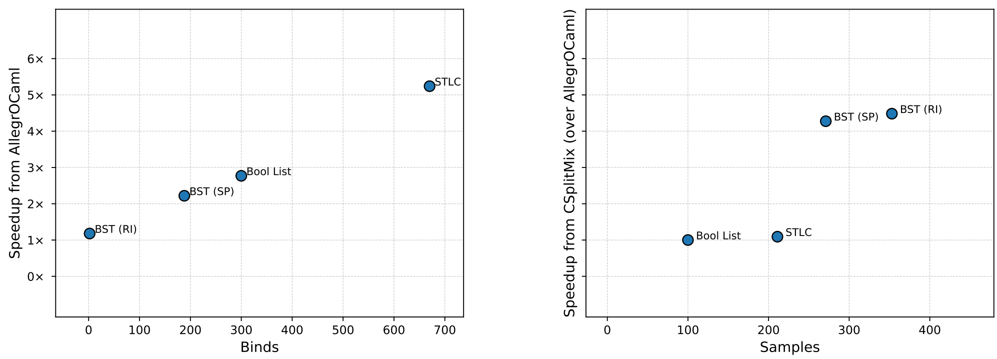

Fig. 15. Left: Speedup from staging (compared to Base_quickcheck) versus the number of bind calls. Right: Speedup from CSplitMix (compared to AllegrOCaml alone) versus the number of random samples.

Figure 15 summarizes the results. The left plot shows a clear linear relationship between performance benefit from staging and number of calls to `bind`. The right plot shows a separation between generators that sample heavily (BST strategies) and those that do not (STLC and Bool List), with the former seeing significantly greater speedups from CSplitMix.

From these results, we conclude that staging and randomness library choice play distinct and complementary roles in generator performance. Sampling-heavy generators see greater improvements from faster randomness libraries, while combinator-heavy ones benefit more from staging. Since both factors influence performance significantly, generator libraries should use both in order to handle diverse workloads.

In ScAllegro, we implement a subset of the AllegrOCaml strategies: the boolean list generator, single-pass BST generator, and well-typed STLC term generator.[11] We did not implement an optimized version of ScalaCheck's randomness library, but our experimental setup was otherwise the same as in OCaml.

As shown in Figure 16, ScAllegro achieves an even greater performance gain—purely from staging—over ScalaCheck than AllegrOCaml does over `Base_quickcheck`. In particular, the single-pass BST strategy is $4.89 - 6.51\times$ faster, the boolean list generator is $6.86 - 13.41\times$ faster, and STLC is $5.43 - 9.70\times$ faster. This difference arises from ScalaCheck's representation of generators as functions of type `size -> seed -> Option[A]`: each generator combinator must construct and then pattern-match on these `Option` values, introducing significant boxing and unboxing overhead at each step. Staging eliminates this overhead. These results show that the Allegro approach generalizes well across languages.

4.2 Benchmarking bug-finding speed

To answer **RQ2**, we evaluate the bug-finding speed of our BST and STLC case studies using Etna [58]. Etna allows users to measure the effectiveness of different generator implementations by injecting bugs into the system under test and recording the time taken for a relevant property to fail in response. Each case study includes a diverse set of *tasks*, where a task consists of a specific bug-property pair designed to test a specific aspect of the system (e.g., BST includes tasks that test insertion, deletion, and union operations). In particular, BST has 37 tasks, and STLC has 20.

[11]The type-derived strategies are excluded, since ScAllegro does not currently support type derivation.

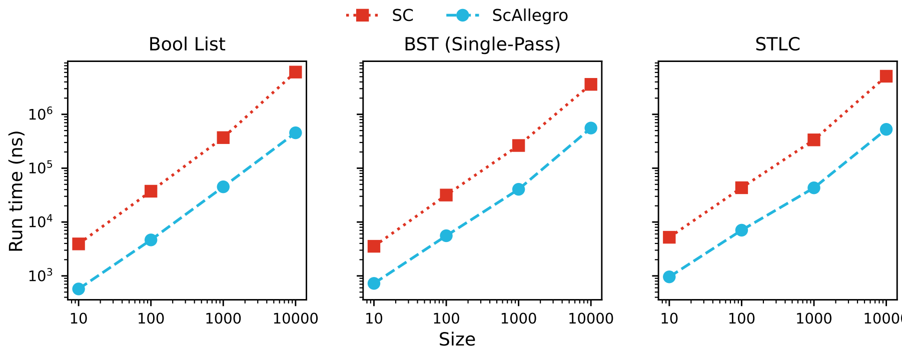

Fig. 16. Time to generate values of varying sizes using each ScAllegro strategy. Lower is better. Both axes are logarithmic. SC is ScalaCheck; Staged is ScAllegro.

Strategies for BST and STLC are implemented in AllegrOCaml; they are unchanged from their description in §4.1. Etna does not support Scala, so we were unable to test ScAllegro's bug-finding speed in Etna, but we expect the results in this section to extend to ScAllegro.

It is worth noting that the time-to-failure of a given strategy on a given seed is not necessarily representative of the strategy's average time-to-failure over a large number of trials. We normalize by computing the relative performance, or the "speedup." This works in most cases, but it is not perfect. For example, it is theoretically possible to choose a seed such that the *first* value produced by a generator discovers the bug, which would eliminate any differential speedup, and likewise if both generators fail to find the bug. To account for these cases, we repeat the above process over 30 random seeds—that is, all strategies run on the same seed so that they produce identical sequences of values, and this process is repeated 30 times. Although the variance *between* seeds in Etna can be vast, timing results are replicable for a *given* seed. Across 1,000 trials using the same seed, the observed variance in time-to-failure was less than a nanosecond.

We run each strategy on all tasks using a 60-second timeout. If a bug is not found within this limit, the task is considered "unsolved." We exclude tasks where all strategies fail to find the bug from our dataset, as they provide no basis for comparison. Similarly, we exclude tasks where the `Base_quickcheck` strategy completes in under 5ms, as such negligible run times do not yield meaningful insights into relative performance. Across all 30 seeds, these filters remove 28/600 (4.67%) of type-derived STLC's tasks, 168/600 (28%) of STLC's tasks, 149/1110 (14.42%) of type-derived BST's tasks, 268/1110 (24.1%) of single-pass BST's tasks, and 371/1110 (33.42%) of insertion-based BST's tasks. More sophisticated strategies tend to find bugs very quickly, leading to a higher number of filtered tasks. By applying these filters, we ensure that our reported speedups reflect optimizations that meaningfully impact performance.

Our results are summarized in Figure 17, which shows the geometric average of individual-task speedups for each strategy and benchmark. Trends in bug-finding speed reflect trends in performance from §4.1. The distribution of speedups in Figure 18 reveals substantial variability, with most tasks clustering near the median and a long tail of outliers achieving much larger gains. STLC shows a more bimodal distribution of speedups than the other benchmarks, which we attribute to the heterogenous difficulty of its tasks: some tasks regularly hit the 60-second timeout, while others finish in a fraction of a second. We find that "easier" tasks—those that run on the order of milliseconds—regularly achieve speedups in the range of $4 - 9\times$, whereas tasks that run on the

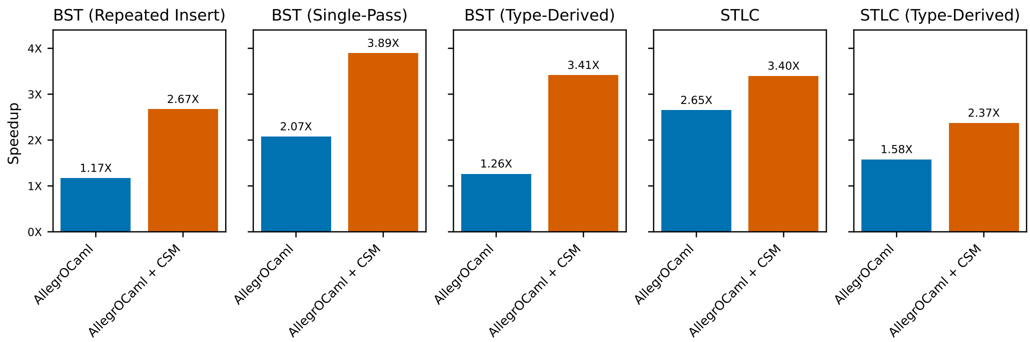

Fig. 17. Geometric average of all speedups—relative to `Base_quickcheck`—for each strategy and benchmark, showing that staging leads to better bug-finding speed across the board. CSM is CSplitMix.

order of seconds achieve more moderate speedups of up to 3.5×. Overall, these results give strong evidence that AllegrOCaml consistently benefits bug-finding speed, sometimes drastically.

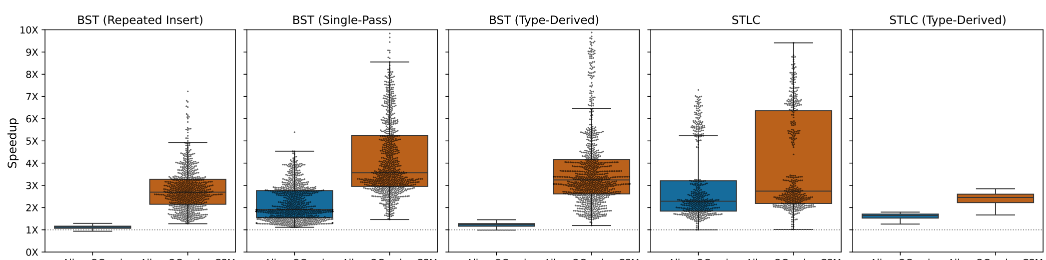

Fig. 18. Distribution of individual-task speedups across strategies and benchmarks. Gray dots represent tasks. Plots with extremely low variance have no swarm overlay. CSM is CSplitMix.

In addition to AllegrOCaml's overall bug-finding speed, we evaluate the performance of its type-derived generators—specifically, whether they can transform previously intractable tasks into tractable ones. Out of 78 tasks that initially timed out, 30 (38.4%) completed successfully using AllegrOCaml alone, and 50 (64.1%) completed successfully after adding CSplitMix. This shows that staged type-derived generators can provide significant performance improvements for free, without any additional effort from the user.

5 Beyond OCaml and Scala

In the years since the original QuickCheck paper [13], PBT has had remarkable success in languages outside of the Haskell world from which it came [2, 24, 61]. For this reason, readers who are users and developers of PBT libraries in *other* languages—including non-strict or procedural ones—might be curious about how the techniques brought to bear on OCaml's `Base_quickcheck` and Scala's ScalaCheck in this paper might be imported to their favorite language.

We begin by noting that the principle of choosing fast random number generators is completely language-agnostic. In languages like OCaml, where runtime value representations make natively-implemented randomness library inefficient, calling out to a C implementation or using standard libraries written in C is a surefire win. In other languages, serious thought should be put into using as fast of a randomness library as possible, as opposed to picking up any suitable option off the shelf. Next, we discuss language-by-language the degree to which the staging-related insights of this paper are portable.

Racket is the next target that we intend to test on. The entire Racket philosophy is intertwined with using macros to build small eDSLs like the ones we use to write generators, and so it seems a natural fit. However, the racket macros literature, while extensive, does not usually concern itself with staging for performance purposes. For this reason, we—the authorship team of this paper, devoid of much Racket expertise—decided to not use Racket as a test case in this paper.

Haskell is home to the original PBT implementation, and so it is natural to ask why we have not built a Allegro in it yet. The basic answer is that the Haskell compiler (GHC) is designed specially to eliminate the run-time overhead of monadic abstractions! Indeed, because Haskell is pure, GHC can aggressively inline and beta-reduce nearly anything as part of its normal optimiation steps. This means that in many cases, the impact of a staged PBT library (and indeed, all staged monadic DSLs) negligible. For complex enough programs however, the heuristics GHC uses can degrade, leading to performance overheads for monadic code. For this reason, Haskell does have a multi-stage programming system called Template Haskell [57], which is sometimes to build eDSLs in contexts where one does not want to rely on the optimizer's heuristics. In short: it is possible to build a Allegro version of Haskell QuickCheck, but the benefits would be much less pronounced.

F# has both multi-stage programming capabilities [42] and a well-used PBT library called FsCheck [19]. The internals of FsCheck's generator eDSL are very similar to both `Base_quickcheck` and ScalaCheck, so we expect that building an Allegro in F# is primarily an engineering effort.

Rust shares some similarities with functional languages, and indeed, it is host to a number of property-based testing libraries [10, 52]. However Rust does not structurally encourage monadic eDSLs—it does not have a special monadic syntax—and so PBT libraries in Rust ask programmers to define generators directly as `seed -> value` functions. For this reason, staging does not seem directly applicable to any of the PBT libraries in Rust.

6 Related Work

Speeding up Property-Based Testing. Our work is unique in its approach to speeding up PBT, but it is certainly not the only work focused on making PBT faster.

Perhaps the best-explored approaches to speeding up bug-finding with PBT is by changing the order in which test inputs are sampled. Feedback-driven mechanisms like Targeted PBT [40] and Coverage-Guided PBT [37] have the potential to speed up testing by changing the generation order to try to find "interesting" inputs faster. Separately, enumerative approaches to PBT [9, 54] try to get to bugs faster by leveraging the "small-scope hypothesis" and attempting to quickly exhaust the list of small inputs to the program under test. All of these approaches can have significant performance benefits in practice, but they are largely orthogonal to our contributions; we expect Allegro could be used to speed these existing techniques to varying degrees.

Similarly orthogonal to our approach are techniques for quickly filtering out inputs that are invalid for testing. Some of these approaches work statically, by automatically deriving generators that produce valid inputs by construction [38], while others filter dynamically via laziness [12] or by solving satisfiability problems [36, 55, 60]. In all cases, these approaches could be further improved by insights from Allegro.

The most direct comparison to our work is QuickerCheck [35], an implementation of Haskell's QuickCheck that exploits the inherent parallelism of PBT to achieve significant performance gains. QuickerCheck, like Allegro, was designed with performance engineering in mind, taking seriously the idea that PBT is a performance critical task. Still, despite their similar motivations and considerations, the solutions are entirely different and complementary.

Multi-Stage Programming. Staging's roots come from quasiquotation in LISP [6], where the unified representation of data and code allows for sophisticated metaprogramming. Quasiquation-style macros were introduced to the ML family of languages by Nielson and Nielson's Two-Level Functional Languages [47] and MetaML [63], which in turn quickly inspired MetaOCaml [28] (and more recently MacoCaml [70]) in OCaml, Lightweight Modular Staging in Scala [53], Template Haskell in Haskell [57], and more. In addition to implementations, researchers have studied the type-theoretic foundations of multi-stage languages [15, 16, 46], as well as ways of combining multi-stage types with other sophisticated features like dependent types [32]. Meanwhile, the Racket community continues the LISP tradition with a long and fruitful line of work studying how macros and metaprogramming can be used to build extensible DSLs [18, 33, 64].

Back in the ML-family world, the primary use of staging is building embedded DSLs with minimal performance overhead, an idea which has come to be known as "abstraction without regret" [49, 50, 53, 56, 65]. This technique has been applied across a range of domains, including Big data processing [5], stream processing [30, 45], query processing [4], and parser combinators [27, 34, 69, 71]. This work on staged parsers is the closest analogue to ours—indeed, PBT research has drawn an explicit link between parsing and generation, framing generators as "parsers of randomness" [22]. However, this equivalence is primarily theoretical. PBT generators are (a) not actually implemented as parsers of random sequences, and (b) support choice-based combinators like `weighted_union` that parsers do not. For this reason, we must build staged generator libraries from scratch and cannot simply use existing staged parser libraries.

7 Conclusion & Future Work

In this paper, we identified, studied, and proposed potential solutions to two important sources of inefficiency in PBT generator libraries: abstraction overhead and choice of randomness library. In the future, we hope to continue to investigate and push the boundaries of PBT generator performance.

On the abstraction overhead side, we hope to employ some of the many tricks for better code generation that have been written about in the staging literature. For example, MetaOCaml includes a primitive floating let-insertion [31] that yields optimal let placement, which we did not use in AllegrOCaml. Another trick we hope to use is GADT-based techniques for unpacking the states of recursive functions. Currently, the AllegrOCaml recursion combinator introduces some overhead from boxing and unboxing the accumulator values at each recursive call. As noticed in other work [32], this overhead can be eliminated using a type-level heterogenous list to represent the state at compile time.

On the randomness libraries side we plan to investigate fast randomness libraries that are not equivalent to SplitMix, such as Lehmer [51], WyHash [72], and variants of xoroshiro [7]. We also want to try techniques to speed up sampling such as pipelining, using SIMD instructions, or even generating random numbers ahead of time. Last, we hope to investigate the degree to which actual statistical randomness matters in PBT.

Data Availabilty Statement

For artifact evaluation we will submit (1) the source code of AllegrOCaml and ScAllegro as well as (2) the code for our version of Etna, which we have modified slightly to (a) support parallelism, and (b) pass deterministic seeds to all versions of a single strategy. These programs are not available publicly or anonymized at the time of submission. The dataset for our evaluation is produced by a combination of Etna and microbenchmarks in the AllegrOCaml and ScAllegro sources. It is also not available publicly at the time of submission, but the artifact will contain the means and instructions to reproduce it.

References

[1] [n. d.]. Macros. https://dotty.epfl.ch/docs/reference/metaprogramming/macros.html. Accessed: 2025-03-17.

[2] [n. d.]. ScalaCheck. https://scalacheck.org/. Accessed: 2025-03-24.

[3] [n. d.]. A small noncryptographic pseudorandom number generator. https://burtleburtle.net/bob/rand/smallprng.html. Accessed: 2025-03-24.

[4] Supun Abeysinghe and Tiark Rompf. 2023. Rhyme: A Data-Centric Expressive Query Language for Nested Data Structures. In *Practical Aspects of Declarative Languages*, Martin Gebser and Ilya Sergey (Eds.). Springer Nature Switzerland, Cham, 64–81.

[5] Stefan Ackermann, Vojin Jovanovic, Tiark Rompf, and Martin Odersky. 2012. Jet: An Embedded DSL for High Performance Big Data Processing. https://infoscience.epfl.ch/handle/20.500.14299/85985

[6] Alan Bawden. 1999. Quasiquotation in Lisp. In *Partial Evaluation and Semantic-Based Program Manipulation.* 4–12. citeseer.ist.psu.edu/bawden99quasiquotation.html

[7] David Blackman and Sebastiano Vigna. 2022. Scrambled Linear Pseudorandom Number Generators. arXiv:1805.01407 [cs.DS] https://arxiv.org/abs/1805.01407

[8] Anders Bondorf. 1992. Improving binding times without explicit CPS-conversion. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming* (San Francisco, California, USA) *(LFP '92)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/141471.141483

[9] Rudy Braquehais, Michael Walker, José Manuel Calderón Trilla, and Colin Runciman. 2017. A simple incremental development of a property-based testing tool (Functional Pearl). (2017).

[10] Bytheway, Cameron. 2025. Bolero. https://camshaft.github.io/bolero/. Accessed: 2025-03-24.

[11] Jacques Carette and Oleg Kiselyov. 2005. Multi-stage Programming with Functors and Monads: Eliminating Abstraction Overhead from Generic Code. In *Generative Programming and Component Engineering*, Robert Glück and Michael Lowry (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 256–274.

[12] Koen Claessen, Jonas Duregård, and Michal H. Palka. 2015. Generating Constrained Random Data with Uniform Distribution. *J. Funct. Program.* 25 (2015). https://doi.org/10.1017/S0956796815000143

[13] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

[14] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, Montreal, Canada, 268–279. https://doi.org/10.1145/351240.351266

[15] Rowan Davies. 2017. A Temporal Logic Approach to Binding-Time Analysis. *J. ACM* 64, 1, Article 1 (March 2017), 45 pages. https://doi.org/10.1145/3011069

[16] Rowan Davies and Frank Pfenning. 2001. A modal analysis of staged computation. *J. ACM* 48, 3 (May 2001), 555–604. https://doi.org/10.1145/382780.382785

[17] Richard A. Eisenberg, Stephen Dolan, and Leo White. 2022. Unboxed types for OCaml. https://www.youtube.com/watch?v=Vevld4cXSYk.

[18] Matthew Flatt. 2012. Creating languages in Racket. *Commun. ACM* 55, 1 (Jan. 2012), 48–56. https://doi.org/10.1145/2063176.2063195

[19] FsCheck Team. 2025. FsCheck: Random Testing for .NET. https://fscheck.github.io/FsCheck/. Accessed: 2025-03-24.

[20] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (Copenhagen, Denmark) *(FPCA '93)*. Association for Computing Machinery, New York, NY, USA, 223–232. https://doi.org/10.1145/165180.165214

[21] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 187, 13 pages. https://doi.org/10.1145/3597503.3639581

[22] Harrison Goldstein and Benjamin C. Pierce. 2022. Parsing Randomness. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (Oct. 2022), 128:89–128:113. https://doi.org/10.1145/3563291

[23] John H. Holland. 1992. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* MIT Press, Cambridge, MA, USA.

[24] Hypothesis Team. 2025. Hypothesis. https://hypothesis.works/. Accessed: 2025-03-24.

[25] Intel Corporation. 2025. Intel® Processor Trace. https://edc.intel.com/content/www/us/en/design/products/platforms/processor-and-core-i3-n-series-datasheet-volume-1-of-2/001/intel-processor-trace/. Accessed: 2025-03-24.

[26] Neil D Jones, Carsten K Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation.* Peter Sestoft.

[27] Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. 2014. Staged parser combinators for efficient data processing. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 637–653. https://doi.org/10.1145/2660193.2660241

[28] Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 86–102.

[29] Oleg Kiselyov. 2023. MetaOCaml Theory and Implementation. arXiv:2309.08207 [cs.PL] https://arxiv.org/abs/2309.08207

[30] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream fusion, to completeness. *SIGPLAN Not.* 52, 1 (Jan. 2017), 285–299. https://doi.org/10.1145/3093333.3009880

[31] Oleg Kiselyov and Jeremy Yallop. 2022. let (rec) insertion without Effects, Lights or Magic. arXiv:2201.00495 [cs.PL] https://arxiv.org/abs/2201.00495

[32] András Kovács. 2024. Closure-Free Functional Programming in a Two-Level Type Theory. *Proc. ACM Program. Lang.* 8, ICFP, Article 259 (Aug. 2024), 34 pages. https://doi.org/10.1145/3674648

[33] Shriram Krishnamurthi and Matthias Felleisen. 2001. *Linguistic reuse.* Ph. D. Dissertation. AAI3021152.

[34] Neelakantan R. Krishnaswami and Jeremy Yallop. 2019. A typed, algebraic approach to parsing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 379–393. https://doi.org/10.1145/3314221.3314625

[35] Robert Krook, Nicholas Smallbone, Bo Joel Svensson, and Koen Claessen. 2024. QuickerCheck: Implementing and Evaluating a Parallel Run-Time for QuickCheck. In *Proceedings of the 35th Symposium on Implementation and Application of Functional Languages (IFL '23)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3652561.3652570

[36] Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner's Luck: A Language for Property-Based Generators. *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017* (2017), 114–129.

[37] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage Guided, Property Based Testing. *PACMPL* 3, OOPSLA (2019), 181:1–181:29. https://doi.org/10.1145/3360607

[38] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2017. Generating Good Generators for Inductive Relations. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30.

[39] Daniel Lemire. 2023. testingRNG. https://github.com/lemire/testingRNG.

[40] Andreas Löscher and Konstantinos Sagonas. 2017. Targeted Property-Based Testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 46–56. https://doi.org/10.1145/3092703.3092711

[41] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[42] Microsoft. 2025. Code Quotations — F# | Microsoft Learn. https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/code-quotations. Accessed: 2025-03-24.

[43] Yaron Minsky and Anil Madhavapeddy. 2022. *Real World OCaml: Functional Programming for the Masses.* " O'Reilly Media, Inc.".

[44] Eugenio Moggi. 1991. Notions of computation and monads. *Information and computation* 93, 1 (1991), 55–92.

[45] Anders Møller and Oskar Haarklou Veileborg. 2020. Eliminating abstraction overhead of Java stream pipelines using ahead-of-time program optimization. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 168 (Nov. 2020), 29 pages. https://doi.org/10.1145/3428236

[46] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Trans. Comput. Logic* 9, 3, Article 23 (June 2008), 49 pages. https://doi.org/10.1145/1352582.1352591

[47] Flemming Nielson and Hanne Riis Nielson. 1992. *Two-level functional languages.* Cambridge university press.

[48] openjdk. 2023. Java Microbenchmarking Harness. https://github.com/openjdk/jmh.

[49] Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. 2017. Quoted staged rewriting: a practical approach to library-defined optimizations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Vancouver, BC, Canada) *(GPCE 2017)*. Association for Computing Machinery, New York, NY, USA, 131–145. https://doi.org/10.1145/3136040.3136043

[50] Lionel Emile Vincent Parreaux. 2020. *Type-Safe Metaprogramming and Compilation Techniques For Designing Efficient Systems in High-Level Languages.* Ph. D. Dissertation. EPFL, Lausanne. https://doi.org/10.5075/epfl-thesis-10285

[51] W. H. Payne, J. R. Rabung, and T. P. Bogyo. 1969. Coding the Lehmer pseudo-random number generator. *Commun. ACM* 12, 2 (Feb. 1969), 85–86. https://doi.org/10.1145/362848.362860

[52] Proptest Contributors. 2025. The Proptest Book. https://altsysrq.github.io/proptest-book/. Accessed: 2025-03-24.

[53] Tiark Rompf and Martin Odersky. 2012. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM* 55, 6 (June 2012), 121–130. https://doi.org/10.1145/2184319.2184345

[54] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values. *ACM SIGPLAN Notices* 44, 2 (Sept. 2008), 37–48. https://doi.org/10.1145/1543134.1411292

[55] Eric L. Seidel, Niki Vazou, and Ranjit Jhala. 2015. Type Targeted Testing. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Jan Vitek (Ed.). Springer, Berlin, Heidelberg, 812–836. https://doi.org/10.1007/978-3-662-46669-8_33

[56] Tim Sheard, Zine el-abidine Benaissa, and Emir Pasalic. 1999. DSL Implementation Using Staging and Monads. In *2nd Conference on Domain-Specific Languages (DSL 99)*. USENIX Association, Austin, TX. https://www.usenix.org/conference/dsl-99/dsl-implementation-using-staging-and-monads

[57] Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (Pittsburgh, Pennsylvania) *(Haskell '02)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/581690.581691

[58] Jessica Shi, Alperen Keles, Harrison Goldstein, Benjamin C. Pierce, and Leonidas Lampropoulos. 2023. Etna: An Evaluation Platform for Property-Based Testing (Experience Report). *Proc. ACM Program. Lang.* 7, ICFP, Article 218 (Aug. 2023), 17 pages. https://doi.org/10.1145/3607860

[59] Guy L. Steele, Doug Lea, and Christine H. Flood. 2014. Fast splittable pseudorandom number generators. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 453–472. https://doi.org/10.1145/2660193.2660195

[60] Dominic Steinhöfel and Andreas Zeller. 2022. Input Invariants. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 583–594. https://doi.org/10.1145/3540250.3549139

[61] Jane Street. 2021. Base Quickcheck. https://github.com/janestreet/base_quickcheck.

[62] Jane Street. 2023. Core bench. https://github.com/janestreet/core_bench.

[63] Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theoretical computer science* 248, 1-2 (2000), 211–242.

[64] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 132–141. https://doi.org/10.1145/1993498.1993514

[65] Laurence Tratt. 2008. Domain specific language implementation via compile-time meta-programming. *ACM Trans. Program. Lang. Syst.* 30, 6, Article 31 (Oct. 2008), 40 pages. https://doi.org/10.1145/1391956.1391958

[66] Janis Voigtländer. 2008. Asymptotic Improvement of Computations over Free Monads. In *Mathematics of Program Construction*, Philippe Audebaud and Christine Paulin-Mohring (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 388–403.

[67] John Von Neumann. 1951. Various Techniques Used in Connection with Random Digits. *Appl. Math Ser* 12, 36-38 (1951), 3.

[68] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. 2010. Mint: Java multi-stage programming using weak separability. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 400–411. https://doi.org/10.1145/1806596.1806642

[69] Jamie Willis, Nicolas Wu, and Matthew Pickering. 2020. Staged selective parser combinators. *Proc. ACM Program. Lang.* 4, ICFP, Article 120 (Aug. 2020), 30 pages. https://doi.org/10.1145/3409002

[70] Ningning Xie, Leo White, Olivier Nicole, and Jeremy Yallop. 2023. MacoCaml: Staging Composable and Compilable Macros. *Proc. ACM Program. Lang.* 7, ICFP, Article 209 (Aug. 2023), 45 pages. https://doi.org/10.1145/3607851

[71] Jeremy Yallop, Ningning Xie, and Neel Krishnaswami. 2023. flap: A Deterministic Parser with Fused Lexing. *Proc. ACM Program. Lang.* 7, PLDI, Article 155 (June 2023), 24 pages. https://doi.org/10.1145/3591269

[72] Wang Yi. 2021. wyhash. https://github.com/wangyi-fudan/wyhash.

Received –; revised –; accepted –