

Fail Faster

ANONYMOUS AUTHOR(S)

[TR: Abstract needs a lot of work.] Property-based testing (PBT) is a software testing approach that verifies a system against a large suite of automatically-generated inputs. Since testing pipelines typically run under strict time constraints, PBT generators must produce inputs as quickly as possible to maximize the likelihood of finding bugs in the time available. However, existing PBT libraries often prioritize generality at the cost of performance. We introduce `waffle_house`, a high-performance generator library that uses staging, a light-weight compilation technique, to eliminate many common generator abstractions. To evaluate `waffle_house`, we design a novel benchmarking methodology that compares generators based on program equivalence, isolating performance improvements from differences in input distribution. Using this methodology, we compare `waffle_house` to a leading generator library, and, through extensive evaluation over a diverse range of generators, demonstrate that `waffle_house` significantly improves generation speed and resource efficiency while matching `base_quickcheck`'s expressiveness.

Additional Key Words and Phrases: Property-based testing, Generators, Staging, Meta-programming

1 Introduction

Property-based testing (PBT) is a technique in which a *generator* produces random inputs to a system under test, which is then evaluated against a set of properties the system is expected to satisfy. Recent studies on PBT usage have shown that practitioners often run tests with very short time budgets—typically between 50 milliseconds and 30 seconds. Given this constraint, generating inputs as quickly as possible is crucial for maximizing the odds of finding a bug in the limited time available. In other words, generators must be treated as performance-sensitive code. A high-leverage way to improve the performance of generators is to optimize *generator libraries*: if programmers are provided with the tools to write fast generators, then they do not themselves need to be performance engineers.

However, the performance of existing generator libraries falls short of what is possible. This is due to two major sources of inefficiency. First, the same high-level design that gives generator libraries their flexibility and expressiveness also introduces overhead: layered abstractions and indirections hinder the compiler's ability to optimize computations, shifting more work to runtime. Second, the process of selecting random values (“random sampling”) can constitute an unexpectedly high proportion of a generator's runtime, depending on one's choice of randomness library and the number of samples needed. This raises a fundamental question: can generator libraries produce code that is both expressive *and* efficient?

To answer this, we propose a two-pronged approach for designing generator libraries that preserves their expressiveness while eliminating unnecessary overhead. First, we describe a method of optimizing generator libraries using *staged metaprogramming*, or “staging”, which has long been used for eliminating overhead from idiomatic functional DSLs [TODO: CITE!!!]. We extend this technique to generators. In unstaged generator libraries, each call to monadic bind constructs and immediately executes a closure at runtime. Meanwhile, a staged approach can resolve the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference acronym 'XX, Woodstock, NY

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-XXXX-X/2025/02

<https://doi.org/XXXXXXX.XXXXXXXX>

bind at compile time, enabling the compiler to inline and eliminate these closures. Secondly, we demonstrate the impact of different choices of randomness library on generator performance by conducting a controlled comparison. Together, these insights provide a recipe for improving even state-of-the-art generator libraries.

To demonstrate the effectiveness of our technique, we apply it to `base_quickcheck`, the fastest property-based testing library in OCaml according to our tests. Our staged version of this library, `base_quickcheck++`, is a direct drop-in replacement for `base_quickcheck` that is *completely semantically equivalent*: that is, given identical random seeds, generators written using `base_quickcheck++` produce exactly the same values as those written in `base_quickcheck`. However, in `base_quickcheck++`, the abstractions provided by the generator DSL are zero-cost, and the randomness library is highly-optimized. The direct equivalence between `base_quickcheck++` and `base_quickcheck` allows us to perform extremely fine-grained comparisons, as generators implemented using both versions of the library will produce exactly the same sequence of values. Therefore, differences in performance are directly attributable to `base_quickcheck++`'s speedup.

To evaluate `base_quickcheck++`'s effectiveness, we implement a series of workloads in each generator library, and show that `base_quickcheck++` runs faster and allocates less than `base_quickcheck`. Further, we show that this performance benefit translates to bug-finding ability by running these workloads in Etna, a platform that simulates real-world PBT usage by injecting bugs into a system and measuring how quickly various generators detect them. Finally, to demonstrate the generality of our technique, we implement `base_quickcheck++` in Scala and benchmark the same generators in the Scala port of the library—thereby highlighting its portability to other strict functional languages and reinforcing its potential for widespread adoption.

In summary, we establish that monadic PBT generator DSLs have a significant performance overhead across languages, and present a cross-language technique for eliminating that overhead while preserving their idiomatic abstractions. We make the following contributions:

- (1) We identify two key sources of inefficiencies in PBT generator libraries, which can significantly impact performance: namely, randomness library choice, and abstraction overhead. §3
- (2) We apply staged metaprogramming to the novel domain of generators. §??
- (3) We implement a version of `base_quickcheck`, `base_quickcheck++`, that showcases our optimization technique while maintaining strict program equality. §??
- (4) We evaluate `base_quickcheck++` by demonstrating the improved performance of generators constructed using our library in a controlled comparison, as well as its impact on bug-finding ability. §5

[JWC: Things to ensure people come away with answers to:

- Is there generality?
- Is this important?
- Did they actually solve the problem?

]

2 Background

[JWC: In this paper, we'll use OCaml to illustrate the ideas, but the concepts are portable to other languages: see Section 2.3]

99 2.1 Property-Based Testing

100 [JWC: Usual spiel about pbt and generators: here's a monadic generic generator library (but we're
 101 using BQ to illustrate), it has return and bind, this is what they're for.] [JWC: You often want
 102 generators to generate for sparse preconditions so you can exercise a property reasonably well]

```
103
104 module Bq = struct
105   type 'a t = int -> SR.t -> 'a
106
107   let return (x : 'a) : 'a t = fun size seed -> x
108
109   let bind (g : 'a t) (f : 'a -> 'b t) : 'b t =
110     fun size seed ->
111       let x = g size seed in
112         (f x) size seed
113
114   let gen_int (lo : int) (hi : int) : int t =
115     fun size seed -> SR.int seed lo hi
116   end
```

117 [JWC: gen_int here is really `int_uniform_inclusive`. I also think we should completely omit
 named arguments in the paper so we don't confuse with splices.]

118 [JWC: Here's an example of a generator: you can sample from it and it gives you pairs of ints,
 119 one less than the other.]

```
120
121 let int_pair : (int * int) Bq.t =
122   let%bind x = Bq.gen_int 0 100 in
123   let%bind y = Bq.gen_int 0 x in
124   return (x,y)
```

125 [JWC: This desugars to...]

```
126 let int_pair : (int * int) Bq.t =
127   Bq.bind (Bq.gen_int 0 100) (fun x ->
128     Bq.bind (Bq.gen_int 0 x) (fun y ->
129       Bq.return (x,y)
130     ))
```

131 [JWC: Explain effect ordering here!!

- 132 • Bq.bind has the CBV sampling semantics
- 133 • If you Bq.bind x and then use it twice, both uses refer to the *same* value.
- 134 • Sampling is effectful: it mutates the internal state.
- 135 • Permuting two independent binds gives you a generator that is distributionally equivalent,
 136 but not equivalent when considered up to pointwise equality as *functions* from seed to
 137 value.
- 138 • Note that it's much more sensible to compare the bugfinding performance of two function-
 139 equivalent generators:

140]

141 [JWC:

- 143 • PBT generator libraries include more functions than just the monad interface and sampling,
 144 libraries include helper functions.
- 145 • With `weighted_union`, to make a weighted choices, and `fixed_point` to define recursive
 146 generators.

- 148 • Also `size` and `with_size` to adjust the size parameter — these are useful to ensure we
 149 terminate with recursive generators.

150]

151 [JWC: This section needs to explain approximately how weighted union works — compute a
 152 sum of the weights, sample between 0 and the sum, pick the first element where the partial sums
 153 to the left exceeds the sampled number. (This has been written about a billion times, cite one of the
 154 papers about this.)]

```
155
156   let tree_of g = fixed_point (fun rg ->
157     let%bind n = size in
158       weighted_union [
159         (1, return E);
160         (n,
161           let%bind x = g in
162             let%bind l = with_size (n / 2) rg in
163               let%bind r = with_size (n / 2) rg in
164                 return (Node (l,x,r))
165               )
166         ]
167       )
168   )
```

[JWC: This generator generates binary trees using all the combinators, yay. Explain em.]

2.2 Multi-Stage Programming

[JWC: I think this section should actually go later.]

[JWC:

- In Multi-stage programming (also known more simply as staging), programs execute over the course of multiple stages, with each stage producing the code to be run in the next.
- This is an old idea, with roots going back to quasiquotation in LISP, and picking up steam again in the 1990s with MetaML [CITE].
- For the purposes of this paper, we will only consider two stages: compile time and run time. Staged programs thus execute twice: once at compile time, which produces more code, which is then compiled and run at run time.
- Staging has many applications, but chief among them is its use for *optimization*. We can write programs such that during the compile time stage, they are partially evaluated to eliminate abstraction overhead.
- Many languages have some degree of staging functionality, either provided as a library [CITE] or build directly into the language's implementation [CITE].
- For this paper, we use MetaOCaml [CITE] for OCaml staging, but all of the staging concepts are portable to any other language with multi-stage programming functionality.

187]

[JWC: This section is going to need work.]

188 MetaOCaml's staging functionality is exposed through a type `'a code`. A value of type `t code`
 189 at compile time is a (potentially open) OCaml term of type `t`.

190 Values of `code` type are introduced by *quotes*, written `.<...>..` Brackets delay execution of a
 191 program until run time. For example, the program `.< 5 + 1 >.` has type `int code`. Note that this
 192 is not the same as `.< 6 >..` Because brackets delay computation, the code is not *executed* until the
 193 next stage (run time). Values of type `code` can be combined together using *escape*, written `.~(e)`
 194 (or just `.~x`, when `x` is a variable). Escape lets you take a value of type `code`, and “splice” it directly
 195 into another quote.

197 into a quote. For example, this program `let x = .<1 * 5>. in .< .~(x) + .~(x) >.` evaluates
 198 to `.<(1 * 5) + (1 * 5)>..` MetaOCaml ensures correct scoping and macro hygiene, ensuring
 199 that variables are not shadowed when open terms are spliced.

200 The power of staging for optimization away abstraction overheads comes from defining functions
 201 that accept and return code values. A function `f : 'a code -> 'b code` is a function that takes
 202 a program computing a run-time '`a`' and transforms it into a program computing a run-time '`b`'. In
 203 particular, because `f` itself runs at compile time, the abstraction of using `f` is necessarily completely
 204 eliminated by run time. A code-transforming function `'a code -> 'b code` can also be converted
 205 *code for a function* — a value of type `('a -> 'b) code` — with the following program:

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

207 This program is known as “the trick” in the partial evaluation and multi-stage programming
 208 literature. [CITE] It returns a code for a function that takes an argument `x`, and then it splats in the
 209 result of calling `f` on just the quoted `x`. [JWC: Does this make any sense?]

210 The trick is best illustrated by an example. The following program reduces to `.< fun x -> (1`
 211 `+ x) mod 2 == 0 >.`

```
212
213 let is_even x = .< .~x mod 2 == 0 >. in
214 let succ x = .< 1 + .~x >. in
215 eta (succ . is_even)
```

216 By composing the two code-transforming functions together at compile time, and only then turning
 217 them into a run-time function, the functions are fused together... [JWC: words]. This is the basis of
 218 how staging is used to eliminate the abstraction of DSLs (like a generator DSL). By writing DSL
 219 combinators as compile-time functions — and only calling `eta` at the end on the completed DSL
 220 program — we can ensure that any overhead of using them is eliminated by run time.
 221

2.3 Other Languages and Libraries

[JWC: TODO: Move this section later (cf conversation in WH meeting 3/7/25)]

[TR: Here's where we talk about what's going on in the world, and ultimately make the argument
 225 that `base_quickcheck` is the best tool to reproduce.]

[JWC: Note that this explanation require some prior note of exactly *why* BQ is slow... i.e. the
 227 abstraction overhead of the library is high.]

[JWC:

- 230 • Scala. Functional abstractions like QC generators are known to be costly in Scala, that's why
 231 they have LMS (in Scala 2, and Macros in Scala 3). Example: parser combinators (“On Staged
 232 Parser Combinators for Efficient Data Processing”), functional data structures ([Link](#)), web
 233 programming (“Efficient High-Level Abstractions for Web Programming”). Al of this should
 234 still work in scala. Could easily be incorporated into ScalaCheck, with minor modification:
 235 ScalaCheck uses a state monad to thread around the seed, instead of a stateful one (like BQ),
 236 or a splittable one (like Haskell). So you have to adapt to that. But same diff.
- 237 • Python. Maybe could do it?? Hypothesis + MacroPy
- 238 • Haskell: GHC does a lot of these optimizations already, since the code is pure. Since QC
 239 generators are relatively small programs, GHC has little trouble specializing them. Of course,
 240 this is not guaranteed. A version of this idea can easily be ported to the original QC with
 241 template Haskell, to guarantee the highest-performance generators.
- 242 • Rust: Not GC'd, so no alloc overhead but bind'd generators still dispatch through runtime
 243 data.

244]

246 3 Sources of Inefficiency in Monadic Generator Libraries

247 [JWC: Why is a monadic generator library slow?] [JWC: NOTE: We can simplify this further by
 248 getting rid of the size parameter. Make the story even cleaner, I think!]

249 [JWC: IMPORTANT: We are using OCaml because we have to pick a syntax for this section, but
 250 these abstraction overheads exist in other languages – at least scala, rust.]

252 3.1 Monadic DSL Abstraction Overhead

253 [JWC:

- 254 • It's been long-known that clean functional abstractions have a runtime overhead (this
 255 should be familiar by this time in the paper).
- 256 • How does (simplified) BQ work?
 - 257 – The basic generator type: 'a generator = int -> SR.t -> 'a. Size and random
 258 seed to deterministic value. (note: SR.t is a mutable seed)
 - 259 – This gets a monad instance in the obvious way (show code).
 - 260 – Also show the code for int, how it calls the underlying SR function.
- 261 • Show benchmarks of the running example, versus the version where you inline everything.
 262 ()
- 263 • Let's look at the running example: inline it all the way.

264]

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

270 let int_pair_inlined : (int * int) Bq.t
271   fun sr ->
272     let x = Splittable_random.int sr ~lo:0 ~hi:100 in
273     let y = Splittable_random.int sr ~lo:0 ~hi:x in
274     (x,y)
```

275 [JWC: Show benchmark difference between these two generators: the benchmark is in waffle-house/handw²⁷⁶
 277 It's about 2x. (see comment in latex here)]

278 [JWC: The overhead of the abstraction is 2x. The reality is actually worse: actual base-quickcheck
 279 includes even more indirection in its type.]

280 [JWC: The native code OCaml compiler, even with -O3, fails to specialize this code and eliminate
 281 the overhead of this abstraction – inlining all of the function definitions and then performing
 282 beta-reductions speeds up sampling by a factor of 2.]

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

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

```

295     y match {
296       case None => (None, seed)
297       case Some(y) => (Some(x,y), seed3)
298     }
299   }
300 }
```

[JWC: Same story in scala. Approximately 2x difference for the same generator (though an order of magnitude slower overall than bq)... (scala uses a slightly different generator type, but the principles are the same) This doesn't even account for more inlining we could do to eliminate the boxing/unboxing of results. (Scalacheck generators can fail, though this adds significant overhead)]

[JWC: The problem is plain: while the monadic abstraction that PBT libraries like base quickcheck provide are indispensable for writing idiomatic generators, the performance overhead of using them is dramatic. Modern compilers use heuristics to decide when to specialize and inline code [CITE], and these heuristics cannot guarantee that generator abstractions are zero-cost. These heuristics are also necessarily conservative about inlining and specializing recursive functions, which all generators of recursive data types must be.]

[TR: Each of these consist of an explanation of the problem and pseudocode outlining the solution.]

[JWC:

- While it's the "correct" abstraction for generators, using a monadic interface prevents the compiler from specializing generator code. Using monadic bind and return obscures the control and data flow of a generator from the compiler. This is compounded when handling recursive functions.
- In cases where the compiler cannot statically eliminate them, running a monadic bind allocates a short-lived closure: we allocate a closure for the continuation, and then immediately jump into it.
- Each closure allocation is relatively cheap, but doing lots of allocation in a generation hot loop adds up fast.
- Similar things have been noticed about monadic parsing DSLs in the past.

]

3.2 Combinator Abstraction Overhead

[JWC: Aside from the usual fact that the compiler can't "see through" the abstraction boundary to specialize, combinators like union and weighted union necessarily allocate a lot in the generation hot path. Even without the dumb BQ array thing, to call union and weighted union, the types mean that you have to allocate a list. These allocations are extremely costly, and happen every sample from the generator. If the generator is recursive (like the tree generator), they occur at each recursive call.]

[JWC: In practice, the possible options are almost always statically known: in the binary tree generator from earlier, we can tell from the text of the code what the weights are, what the generators are, and how many there are. Put more simply, you basically always call weighted union with an *explicit list*, which is then immediately traversed by the weighted union. For this reason, you should not have to incur the cost of allocating this list at runtime. However, almost no compilers (GHC is a notable exception) can figure this out and eliminate the list [CITE](list fusion).]

]

343

344 **3.3 Choice of Randomness Library**

345 [JWC: We really need to have explained the effect ordering equivalence aspect of the evaluation
 346 before here.]

347 The core of any PBT generator library is a source of randomness: to generate random values
 348 of some datatype, we some access to random numbers! Different PBT libraries use different ran-
 349 domness libraries implementing different algorithms ¹. Following the original Haskell QuickCheck
 350 implementation, `base_quickcheck` uses the SplitMix algorithm [CITE], implemented as a (stateful)
 351 OCaml library called `Splittable_random`. Meanwhile, ScalaCheck uses the JSF algorithm [CITE].

352 The randomness library sits at the heart of the hot path. Even basic generators — like ones
 353 generating a single `int` or `float` uniformly within a range — can sample *unboundedly many*
 354 random numbers, since they usually use versions of rejection sampling to find a value within the
 355 range [CITE]. Moreover, generator combinators like `list` usually make $O(n)$ calls to even those
 356 basic generators. Because of this, the speed of a single sample matters a great deal. Unfortunately,
 357 existing PBT libraries make relatively inefficient choices on this front, leading to worse bugfinding
 358 power than what is possible.

359 [HG: This paragraph feels a bit clumsy right now. What if we started with a table of the RNGs
 360 used by a bunch of popular PBT frameworks and then argued that things like Lehmer are faster?]

361 For example, significantly faster algorithms than SplitMix or JSF exist, such as the Lehmer
 362 algorithm [CITE]. In microbenchmarks, the Lehmer algorithm runs almost 2x as fast as SplitMix
 363 [CITE]. Moreover, PBT libraries could even consider eschewing the requirement that a source
 364 of entropy pass statistical tests like BigCrush [CITE]. [HG: This needs more discussion] This is
 365 common practice in other areas of testing already: fuzzers often simply use a buffer full of arbitrary
 366 bytes as a source of entropy [CITE]. Such approaches are also faster than algorithms like SplitMix:
 367 bumping a pointer and reading from memory (which can be pipelined trivially) will always be
 368 faster on modern CPUs than any random sampling algorithm with data-dependent arithmetic
 369 instructions. [HG: Should we just always be doing this? I think we want to argue that this is a
 370 bridge too far because you still need to generate the buffer ahead of time and you can run out of
 371 randomness?]

372 Of course, simply arguing that the randomness library is on the hot path does not guarantee
 373 that a faster sampling leads to measurably faster generation; for that, we need an experiment. The
 374 most obvious experiment is to simply swap out the randomness library for `base_quickcheck` with
 375 a totally different, faster one. But, as discussed previously, generators with different generation
 376 orders can be difficult to compare. To get around this, we exploit a “natural experiment”: OCaml’s
 377 `Splittable_random` library is slow in a way that can be improved *without* changing its extensional
 378 behavior. In particular, due to implementation details related to the OCaml garbage collector, values
 379 of the OCaml type `int64` are not machine words, but rather *pointers* to machine words. This means
 380 that *all* `int64` operations (both arithmetic and bitwise) must allocate memory cells to contain their
 381 output, which has a significant performance benefit. By building a version that uses much faster
 382 “unboxed” 64-bit integer arithmetic, we can demonstrate just how much bug-finding performance
 383 can be improved just by using a more performant randomness library.

385 **4 Eliminating Abstraction Overhead of Generator DSLs by Staging and Faster
 386 Randomness Libraries**

387 [JWC: This section needs a much better title...] [JWC: Emphasize that this is an *COMPLETELY
 388 EQUIVALENT drop in replacement!* We din’t just build a different library with different distributions.]

389 ¹The common term for such an algorithm or library is a “Random Number Generator” (RNG) we will avoid this term and
 390 instead say “randomness library” to avoid confusing RNG implementations with the PBT gnenerator libraries that use them.

393 [JWC: Usual introduction to this section, corresponding to how we talked about it in the intro.
 394 “We present a library that XYZ”.]

395 [HG: TODO: Signposting – I got a few paragraphs in and realized I wasn’t sure what I was
 396 reading]

398 4.1 Design of a Staged Generator DSL

399 Recall that staging a DSL involves changing the combinators to run at compile time by carefully
 400 annotating their types with codes. Deciding which types `t` can instead be `t code` – in other
 401 words, figuring out which parts of the DSL can be determined statically (and can be part of the
 402 compile-time stage), and which parts are only known dynamically (and hence must be code) – is
 403 an art known as “binding-time analysis” [CITE].

404 The crux of our binding time analysis is that the particular random seed and size parameter
 405 [JWC: if explained] are only known at run time (the later stage), but the code of the generator
 406 itself is known at compile time.[HG: Are we going to talk more about how we did this analysis?] [JWC: Well it’s sort of right here: you just think about it, it’s the same thing as staging a parser too.] Generators – values of type `'a Bq.t` – are always constructed statically in practice, so all of
 409 the combinators we use to build them can run at compile time.

410 This means our library’s generator type `'a Gen.t` should have the type `int code -> SR.t`
 411 `code -> 'a code`: a compile-time function from dynamically-known size and seed to dynamically-
 412 determined result.

413 This type, along with basic monadic generator DSL functionality can be found in Figure ???. The
 414 monadic interface is given by a return and bind, as usual. Return is the constant generator, but this
 415 time it runs at compile time. Given `cx : 'a code`, the code for a `'a`, it returns the generator which
 416 always generates that value. bind `g k` sequences generators by passing the result of running the
 417 generator `g` to a continuation `k`. However, instead of getting access to the particular value generated
 418 by `g`, the continuation `k` gets access to code for the value sampled from `g`: at compile time, we
 419 only know `g` will generate *some* `'a`, but not which one ². Operationally, bind takes code for the
 420 size and seed, and returns code that (1) let-binds a variable `a` to spliced-in code that runs `g`, and
 421 then (2) runs the spliced-in continuation `k`. Both function applications `g size_c random_c` and `k`
 422 `.<a>. size_c random_c` run at compile time. `Gen.int` is the generator that samples an int from
 423 the randomness library. Given any size and random seed, it returns a code block that calls `SR.int`
 424 with that random seed. Because the lower and upper bounds might not be known at compile time
 425 – they may themselves be the results of calling `Gen.int` – the arguments `lo` and `hi` are of type
 426 `int code`, and get spliced into the code block as arguments to `SR.int`. Lastly, `to_bq` turns a staged
 427 generator into code for a normal `base_quickcheck` generator. This function is just a 2-argument
 428 version of “The Trick” (eta from Section 2.2). [HG: TODO: Be really careful with formatting of the
 429 above paragraph. It’s very easy for some inline code to get split weirdly over a line break or just be
 430 difficult to parse in general, and that could throw the reader off when the content is already pretty
 431 low-level]

432 Returning to our running example, Figure ?? shows the int-pair example written with the staged
 433 `Gen.t` monad, as well as the inlined code that results from calling `Gen.to_bq` (changing some
 434 identifier names for clarity). The code generated is identical to the manually inlined version from
 435 Section 3, and of course runs equally fast.

436 [JWC: Is there anything more we need to say here?]

438 ²Readers familiar with OCaml may notice that `return : 'a code -> 'a Gen.t` and `bind : 'a Gen.t -> ('a code ->
 439 'b Gen.t) -> 'b Gen.t` do not have the correct types for a monad instance, preventing us from using `let%bind` notation.
 440 We rectify this issue in Section 4.4 by importing some clever ideas from the staging literature.

```

442 module Gen = struct
443   type 'a t = int code -> Random.t code -> 'a code
444
445   let return (cx : 'a code) : 'a t = fun size_c random_c -> cx
446
447   let bind (g : 'a t) (k : 'a code -> 'b t) : 'b t =
448     fun size_c random_c ->
449       .<
450         let a = .~(g size_c random_c) in
451         .~(k .<a>. size_c random_c)
452       >.
453
454   let int (lo : int code) (hi : int code) : int t =
455     fun size_c random_c ->
456       .< SR.int .~random_c .~lo .~hi >.
457
458   let to_bq (g : 'a code Gen.t) : ('a Bq.t) code =
459     .<
460       fun size random -> .~(g .<size> . <random>.)
461     >.
462 end

```

Fig. 1. Basic Staged Generator Library
??

```

463
464
465
466 let int_pair_staged : (int * int) Gen.t =
467   Gen.bind (Gen.int .<0>..<100>.) (fun cx ->
468     Gen.bind (Gen.int .<0> cx) (fun cy ->
469       Gen.return .<(.~cx,.~cy)>.
470     )
471   )
472
473 let int_pair : (int * int) Bq.t code = Gen.to_bq int_pair_staged
474 (* .< fun size random ->
475   let x = SR.int random 0 100 in
476   let y = SR.int random 0 x in
477   (x,y)
478   >.
479 *)

```

Fig. 2. Pairs of Ints, Staged
??

4.2 Staging Combinators

In Section 3, we noted that generator combinators like `weighted_union` must allocate lists in the hot path of the generator. Even though these lists are often small — usually at most a few dozen elements in practice — each allocation takes us closer to the next garbage collection.

This is an ideal opportunity to exercise another use of staging: compile-time specialization. Since we almost always know the particular list of choices at compile time, a staged version of

```

491 module Gen =
492 ...
493 let pick (acc : int code) (weighted_gens : (int code * 'a t) list) (size : int code) (random : Sr.t code)
494   match weighted_gens with
495   | [] -> .< failwith "Error" >.
496   | (wc,g) :: gens' ->
497     .<
498       if .~acc <= .~wc then .~(g size random)
499     else
500       let acc' = .~acc - .~wc in
501       .~(pick .<acc'>. gens' size random)
502     >.
503
504 let weighted_union (weighted_gens : (int code * 'a t) list) : 'a t =
505   let sum_code = List.foldr (fun acc (w,_) -> .<.~acc + .~w>.) .<>. weighted_gens in
506   fun size random ->
507     .<
508       let sum = .~sum_code in
509       let r = SR.int .~random_c 0 sum in
510       .~(pick .<r>. weighted_gens size random)
511     >.
512
513

```

Fig. 3. Staged Weighted Union

514 weighted_union can generate *different code* depending on the number of generators in the union.
 515 If we use weighted union on a compile-time list of generators g1, g2, and g3, we can emit code that
 516 picks between the generators without realizing the list at run time.

517 Figure 3 shows the code for such a staged weighted union. Crucially, it takes a *compile-time*
 518 list weighted_gens of generators and weights. The weights themselves might only be known at
 519 run time — it is common to use the current size parameter as a weight, for instance — so they
 520 are codes. Gen.weighted_union begins by computing sum_code, an int code that is the sum of
 521 the weights. Note that this happens at compile time: we fold over a list known at compile time to
 522 produce another code value. We then call SR.int to sample a random number r between 0 and
 523 the sum. Finally, we splice in the result of calling the helper function pick. pick produces a tree
 524 of ifs by again traversing the list of generators at compile time. This tree of ifs “searches” for
 525 the generator corresponding to the sampled value r, and then runs it. [HG: I'm not sure this helps
 526 me. I think I'd prefer a less detailed explanation of the code that conveys the intuition and let the
 527 reader actually read the code if they want to know specifics. As it stands the explanation is just too
 528 dense for me to process]

529 Figure 4 demonstrates a use of this staged weighted union. Given a list (in this case constant)
 530 generators with weights “the current size parameter”, 2, and 1, the generated code first computes
 531 the sum of these numbers, samples between 0 and the sum, and then traverses a tree of three ifs
 532 to find the correct value to return.

533 4.3 Let-Insertion and Effect Ordering

535 Careful readers might note that the definition of bind in Section 4.1 was more complicated than
 536 one might expect. In particular, why not define bind in a more standard way: as let bind' g
 537 k = fun size random -> k (g size random) size random, without the code block that
 538 let-binds the spliced code .~(g size random)?[HG: Need to spell this out – the vast majority of
 539

```

540 let grades : char Bq.t = Gen.to_bq (
541   Gen.bind size (fun n ->
542     Gen.weighted_union [
543       (n, Gen.return .<'a'>);
544       (.<2>,, Gen.return .<'b'>);
545       (.<1>,, Gen.return .<'c'>);
546     ]
547   )
548 (*
549   .< fun size random ->
550     let sum = size + 2 + 1 + 0 in
551     let r = SR.int random 0 sum in
552     if r <= size then 'a'
553     else
554       let r' = r - size in
555       if r' <= 2 then 'b'
556       else
557         let r'' = r' - 2 in
558         if r'' <= 1 then 'c'
559         else
560           failwith "Error"
561     >.
562   *)
563
564
565
566
567

```

Fig. 4. Use of Staged Weighted Union

568 readers won't have that precise question in their heads. Might even be worth comparing the two
 569 implementations here side by side] Unfortunately, using Gen.bind' leads to incorrect code being
 570 generated. For example, consider Gen.bind' (Gen.int .<0>. .<1>.) (fun x -> Gen.return
 571 .<(. x,. x)>.). This generates the run time code fun size random -> (SR.int random 0 1,
 572 int SR.random 0 1), which is incorrect. This is not equivalent to the behavior of writing the
 573 same code with base quickcheck [JWC: Which is a property we want to hold for our eval]. Instead
 574 of generating a single integer and returning it twice, it samples two different integers.

575 This problem is intimately related to nondeterministic effects in the presence of CBN/CBV
 576 evaluation ordering [CITE]. In essence, the behavior of splice $\sim cx$ in a staged function $f(cx : 'a$
 577 code) = ... is to copy the entire block of code, effects and all. To ensure that the randomness effects
 578 of the first generator are executed only once, but that the value can be used in the continuation
 579 multiple times, bind let-binds the result of generation to a variable, and then passes that to the
 580 continuation.

581 The library [JWC: ensure this is consistent with the way we talk about the project, cf cross-
 582 language] is carefully designed to preserve exactly the effect order of base quickcheck [JWC: and the
 583 scala version to preserve the effect ordering of ScalaCheck]. [JWC: This fact should go earlier. Not
 584 really sure about where this subsection should actually land, but it needs to be said somewhere.]
 585 [HG: +1, I think this feels sort of out of place here, but I also think moving it up would make that
 586 part harder to understand too. Is there a way to tie this in to our discussion about maintaining the
 587 precise set of choices? The two issues are different, but they're related]

589 4.4 CodeCPS and a Monad Instance

590 This is all great so far,[HG: too informal] but there's a subtle issue that prevents the version of the
 591 library design discussed so far from being used as a proper drop-in replacement for an existing
 592 generator DSL: the types of `return : 'a code -> 'a Gen.t` and `bind : 'a Gen.t -> ('a`
 593 `code -> 'b Gen.t) -> 'b Gen.t` aren't quite right. For the type '`a Gen.t` to actually be a
 594 monad, the these types cannot mention `code`. This is not just a theoretical issue, it is a significant
 595 usability concern: the syntactic sugar for monadic programming (`let%bind` in OCaml, `foreach`
 596 in Scala, `do` in Haskell, etc) that makes it so appealing can *only* be used if the types involved are
 597 actually the proper monad function types. [HG: A pedantic reader may ask: Are we concerned with
 598 the monad laws as well, or just the type signature?]

599 To support real monadic programming, we'll need to adjust the type of '`a Gen.t` slightly. An
 600 initial attempt is to try type '`'a t = int code -> SR.t code -> 'a`. If we strip the `code` off the
 601 result type, the functions `return (x : 'a) = fun _ _ -> x` and `bind (g : 'a t) (k : 'a ->`
 602 '`b t) = fun size random -> k (g size random) size random` have the proper types for a
 603 monad instance. Then, any combinators of type '`a Gen.t` before simply become '`a code Gen.t`
 604 with this new version. [JWC: do we want a different name for the first cut?]

605 However, this definition of `bind` doesn't have call-by value effect semantics, as discussed in the
 606 previous section! And because the type of `g size random` is just '`a` (not necessarily '`a code`), we
 607 cannot perform the let-insertion needed to preserve the CBV effects. To solve this problem, we turn
 608 to a classic technique from the multistage programming literature: writing our staged programs in
 609 continuation-passing style [1]. [HG: Slow down! This is all very interesting and very technical! Try
 610 making each of these sentences two sentences, add some citations, and maybe spell this out with
 611 an example] [JWC: I could, but i'm just not sure this is critical.]

612 In Figure 5, we follow prior work [2, 3] and define the type '`'a CodeCps.t = 'z. ('a -> 'z
 613 code) -> 'z code`: a polymorphic continuation transformer with the result type always in `code`³.
 614 The monad instance for this type is the standard instance for a CPS monad with polymorphic return
 615 type. In prior work, this type is often referred to as the "code generation" monad (and sometimes,
 616 ironically, called "Gen"). This is because a value of type '`'a code` `CodeCps.t` is like an "action"
 617 that generates `code`: `CodeCps.run` passes the continuation transformer the identity continuation
 618 to produce a '`a code`. To avoid confusion with random data generators, we refer to this type as
 619 `CodeCps.t`. Most importantly, the `CodeCps` type supports a function `let_insert`, which, given `cx`
 620 : '`a code`, let-binds `let x = .~cx`, and then passes `.<x>`. to the continuation. [HG: I'm getting
 621 lost in the inline code again]

622 We can then redefine our staged generator monad type to be '`'a Gen.t = int code -> SR.t
 623 code -> 'a CodeCps.t`, as shown in Figure 5. [JWC: any types that were '`a Gen.t` before are
 624 now '`a code Gen.t`]. This gives us the best of both worlds. First, we get a monad instance for
 625 '`a Gen.t` with the correct types, which lets us use the monadic syntactic sugar of our chosen
 626 language. Moreover, we also get to maintain the correct effect ordering: effectful combinators like
 627 `Gen.int` do their *own* let-insertion, ensuring that a program like `Gen.bind Gen.int (fun x ->`
 628 `...)` generates a let-binding for the result of sampling the randomness library. For example, `bind`
 629 `(int .<0>. .<1>) (fun cx -> return .<(. cx, . cx)>.)` now correctly generates `.< fun
 630 size random -> let x = SR.int random 0 1 in (x,x) >..` [JWC: Should we do out the whole
 631 reduction sequence here? It might be explanatory, but it also might be boring.]

632 This design is less obviously correct, and does require some care. Rather than `bind` ensuring
 633 correct evaluation order once and for all, individual combinators must be carefully written to ensure
 634

635 636 ³This is an instance of the *codensity* monad [4], a fact which deserves further investigation.

```

638 module CodeCps = struct
639   type 'a t = { cps : 'z. ('a -> 'z code) -> 'z code }
640
641   let return x = {cps = fun k -> k x}
642
643   let bind (x : 'a t) (f : 'a -> 'b t) : 'b t =
644     {cps = fun k -> x.cps (fun a -> (f a).cps k)}
645
646   let run (t : ('a code) t) : 'a code = t.cps (fun x -> x)
647
648   let let_insert (cx : 'a code) : 'a code t =
649     {cps = fun k -> k .< let x = .~cx in .~(k .<x>.) >.}
650 end
651
652 module Gen = struct
653   type 'a t = int code -> SR.t code -> 'a CodeCps.t
654
655   let return (x : 'a) : 'a t = fun _ _ -> CodeCps.return x
656
657   let bind (g : 'a t) (f : 'a -> 'b t) =
658     fun size random ->
659       CodeCps.bind (g size random) (fun x ->
660         (f x) size random
661       )
662
663   let int (lo : int code) (hi : int code) : int code t =
664     fun size random -> let_insert .< SR.int .~random .~lo .~hi >.
665
666 end
667

```

Fig. 5. CodeCPS and The Final Gen Monad

```

668 let int_or_zero : int Bq.t =
669   let%bind n = size in
670   if n <= 5 then return 0 else Bq.int
671
672 let split_bool (b : bool code) : bool Gen.t =
673   fun _ _ -> _
674

```

Fig. 6. ??

677 that 'a code values that contain effects are let_insert'd. In Section 5.1, we discuss how we use
678 PBT to ensure that the OCaml is written correctly.
679

[JWC: ... and all of this works identically in Scala, too.]

681 *4.4.1 The Trick at Other Types.* To write more interesting generators, we also need the ability to
682 generate code that manipulates runtime values. For instance, consider this generator

[JWC: need a better example here]

[JWC: Describe split — this section is approximately experts-only, but you can just point at the
685 Andras paper and the things it cites.]

```

687 type 'a handle
688 val recurse : 'a handle -> 'a code Gen.t
689 val fixed_point : ('a handle -> 'a code Gen.t) -> 'a code Gen.t
690

```

Fig. 7. Staged Recursive Generator Combinator API

4.5 Recursive Generators

[JWC: Generating recursive datatypes requires recursive generators!]

Different generator DSLs handle defining recursive generators differently. Some allow recursive generators to be defined as recursive functions (or in Haskell's case, recursive values), while others (like `base_quickcheck`) expose a fixpoint combinator to construct recursive generators. Generator fixpoint combinators are similar to a usual `fix` : (`'a -> 'a`) -> `'a` combinator, except that they operate at monadic type `fixed_point` : (`'a Bq.t -> 'a Bq.t`) -> `'a Bq.t`. 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 case, 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 too 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 combinator⁴, whose API is shown in Figure 7. The code for this combinator can be found in Appendix [JWC: appendix]. 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.

4.6 Staged Type-Derived Generators

[JWC:

- In PBT in practice, we learned that in many cases, programmers don't even write custom generators, instead relying on type-derived generators.
- These generators just produce arbitrary values of a given type, not necessarily enforcing validity conditions.
- If
- Let's talk about how type-deriving works. In languages with typeclasses, it works by typeclass resolution. In OCaml it works by PPX system, but the principle is the same. You define rules to go from generators of a subtypes to a generator of the larger type, and then apply those rules to build up a full generator.
 - For base types, you just call the associated generator
 - For product types, you sample from all the component generators, bind the values, and then tuple up the values, and return the tuple
 - For variant types, you use a weighted union to choose one of the component generators with a weighted union.
 - For recursive types, wrap the whole thing in a fixedpoint and then use the recursive handle as the "component generator" for all recursive instance of the type.
- This kind of generic deriving of generators works just as well for staged generators. Just replace the standard combinators in question with staged ones!

⁴In reality, we actually have a more general API that allows programmers to define *parameterized* recursive combinators, of type `'r code -> 'a code Gen.t`, for any type `'r`. See Appendix [JWC: appendix] for details.

- 736 • We built this in OCaml, but you can easily use typeclasses to do it in scala too.
 737 • Note that this is (essentially) 3-stage metaprogramming. You're using either the PPX mecha-
 738 nism or typeclass resolution to generate a bunch of staged combinator calls, which then
 739 run at compile time, which then run at run time.

740] [JWC: Todo: Thia]

741

742 4.7 Faster SplitMix with Unboxed int64s

743 [JWC: Unclear what we should call this section] As we discussed in Section 3, choosing an inefficient
 744 randomness library is another bottleneck for finding bugs fast. While generator libraries by and
 745 large use sensible sources of randomness, they are not explicitly chosen with performance in mind.
 746 Indeed, much faster randomness libraries [CITE] and fast sources of entropy [CITE] exist. [JWC: ...
 747 more here?] To demonstrate that faster random sampling can significantly impact bugfinding power,
 748 we use the natural experiment provided by OCaml's inefficient implementation of SplitMix. By
 749 replacing this slow randomness library with a faster but extensionally equivalent implemetnation,
 750 we are able to precisely quantify the bugfinding speedup that using a randomness library gives
 751 across a range of PBT scenarios.

752 We emphasize that while we believe that the insight that faster sampling translates to faster bug
 753 finding is a cross-langauge one, the specific technical contents of this section are OCaml-specific.
 754 In the case of most other PBT frameworks [CITE], the randomness library used operates on by
 755 machine integers, so this *particular* inefficiency does not exist.

756 The precise details of how the SplitMix algorithm works [CITE] are unimportant for the present
 757 paper, but the critical component is that all of its operations are defined in terms of arithmetic
 758 bitwise operations on 64-bit integers. In OCaml, because of details related to the garbage collector,
 759 the 64-bit integer type `int64` is represented at run time as a *pointer* to an unscanned block of
 760 memory containing (among other things) a 64-bit integer [CITE]. This means that all operations
 761 that return an `int64` must allocate this block of memory. This has a significant impact on the
 762 performance of generators. A single call to one of the `base_quickcheck` library functions – like
 763 generating an integer uniformly in a range – may call the `splittable_random` algorithm multiple
 764 times. Each sample from `splittable_random` allocates 9 times [JWC: this is a call to `next_int64`],
 765 and each allocation brings us closer to the next garbage collection pause. While small allocations
 766 like these are *very* fast to perform and subsequently collect in OCaml,⁵, we will see in Section 5 that
 767 this can have a large performance impact on some generators that spend most of their time sampling
 768 data. To circumvent this allocation and provide an equivalent version of `splittable_random`, we
 769 reimplement SplitMix in C, and call out to it with the OCaml FFI. The C version of the library uses
 770 proper `int64_t` arithmetic, only boxing and unboxing integers at the call boundaries between
 771 OCaml and C code. Ideally in the future, one would not need to call out to C for this: the Jane
 772 Street bleeding-edge OCaml compiler has support for unboxed types [CITE], which (among other
 773 things) would let us implement a version of SplitMix that does not allocate, directly in OCaml.
 774 Unfortunately, the Jane Street branch of the compiler is incompatible with MetaOCaml, which we
 775 use to implement the metaprogramming discussed in the previous sections.

776

777 5 Evaluation

778

779 5.1 Implementing and Testing Generators

780 [JWC: Talk about the difftesting to ensure staging maintains semantics, and also different RNGs
 781 have identical semantics.]

782 ⁵The OCaml GC is a generational collector [CITE], and since these allocations are small and mostly very short lived, they
 783 will all be minor allocations, never to be promoted.

785 5.2 Benchmarking speed & resource usage

786 [JWC: NOTE: we should test generator speed across both languages, but speed -> bugfinding ability
787 in only OCaml.] [JWC: Baseline generators to test speed in both languages:

- 788 • Single int
- 789 • Pair of ints, constrained
- 790 • List of ints without bind (use map): lots of sampling, minimal binds.
- 791 • Unableled Trees of a fixed size (no weighted union) minimal sampling, lots of binds.

792]

793 5.3 Impact on bug-finding ability

794 [JWC: Staging type-derived generators (which are the most commonly used ones) can turn a
795 timeout into a found bug!]

796 6 Conclusion & Future Work

797 7 Related Work

800 [JWC: A generator is a parser of randomness, and people have implemented lots of staged parser
801 libraries!]

802 References

- 803 [1] 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>
- 804 [2] 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.
- 805 [3] 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>
- 806 [4] 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.

807 Received –; revised –; accepted –

808

809

810

811

812

813

814

815

816

817

818

819

820

821

822

823

824

825

826

827

828

829

830

831

832

833