Deeper Properties for Programmable Testing

ALPEREN KELES, University of Maryland, College Park, USA
JUSTIN FRANK, University of Maryland, College Park, USA
CEREN MERT, University of Maryland, College Park, USA
HARRISON GOLDSTEIN, University of Maryland, College Park, USA
LEONIDAS LAMPROPOULOS, University of Maryland, College Park, USA

Property-based testing (PBT) is a popular technique for establishing confidence in software, where users write *properties*—i.e. executable specifications—that can be checked many times in a loop by a testing framework. In modern PBT frameworks, properties are usually written in *shallowly embedded* domain specific languages, and their definition is tightly coupled to the way they are tested. This coupling significantly limits the programmability of testing: changing the way the property runners works often necessitates writing a new framework from scratch. Even when the configuration options provided by a framework suffice, the shallowness of the embedding means that such configurations need to be interleaved with the property itself, while users are also limited to precisely what library authors had the prescience to allow for when developing the framework.

We propose a new, deeper—but not entirely deep—language for properties based on a mixed embedding that we call *deferred binding abstract syntax*, allowing us to reify properties as a data structure and decouple them from the property runners that execute them. We implement this language in Rocq and Racket, leveraging the power of dependent and dynamic types in each respectively. Finally, we showcase the flexibility of this new approach by rapidly prototyping a variety of property runners, highlighting domain-specific testing improvements that can be unlocked by more programmable testing.

1 INTRODUCTION

In property-based testing [5], users build confidence in their code using *properties* that describe what it means for a program to be correct, expressed in the form of universally quantified executable predicates: e.g. for all programs, compiling them with and without optimizations yields assembly with the same behavior; for all abstract syntax trees, pretty printing them and parsing the result yields back the original. Property-based testing frameworks then provide both a *property language*, an API for expressing such properties along with (optional) annotations that configure testing behavior (such as how to generate or pretty print test inputs); as well as a *property runner*, a way to actually test a user-specified property taking any annotations into account.

Consider, for example, the print-parse roundtrip property above in Haskell's QuickCheck:

```
prop_roundtrip :: Property                    gen :: Gen Exp
prop_roundtrip =                              shrink :: Exp -> [Exp]
  forAllShrink gen shrink                     pretty :: Exp -> String
    (\e -> parse (pretty e) = Just e)         parse :: String -> Maybe Exp
```

Given some type of expressions `Exp`, users have to write a generator for expressions `gen`–that is, a function from some random seed to a concrete `Exp`; a shrinking function `shrink`—a function from an expression to a list of potentially smaller expressions for minimization purposes; and a predicate on expressions, which in this case is a function that given an expression `e`, prints it, parses the result, and checks that it is equal to the original. To create a `Property` that QuickCheck can test,

Authors' addresses: Alperen Keles, akeles@umd.edu, University of Maryland, College Park, , College Park, Maryland, USA, 20740; Justin Frank, jpfrank@umd.edu, University of Maryland, College Park, , College Park, Maryland, USA, 20740; Ceren Mert, cmert@umd.edu, University of Maryland, College Park, , College Park, Maryland, USA, 20740; Harrison Goldstein, harrygol@umd.edu, University of Maryland, College Park, , College Park, Maryland, USA, 20740; Leonidas Lampropoulos, leonidas@umd.edu, University of Maryland, College Park, , College Park, Maryland, USA, 20740.

users can leverage the `forAllShrink` combinator from QuickCheck' property language API to put everything together.

In turn, QuickCheck provides a property runner to test such a `Property`, consisting of two tight loops: first, it repeatedly generates a random expression using gen and executes the predicate until it succeeds in falsifying the predicate (or a predefined limit is reached); then, if a counterexample `e` has been found, it tries to minimize it by recursively checking if any elements of shrink e also falsify the predicate, and repeating this process until a (local) minimum has been reached which is finally reported to the user. A pictorial depiction of the runner can be seen at Fig. 1.

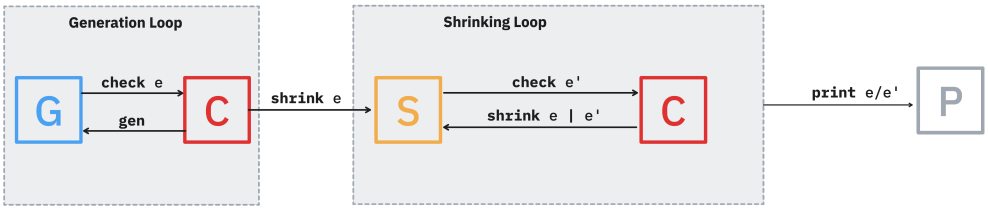

Fig. 1. QuickChick Property Runner

Unsurprisingly, the design of the property runner itself has a tremendous effect on the performance of the random testing process, and the random testing literature is abundant with novel attempts to re-architect the underlying loops. One example of a class of such attempts is mutation-based generation approaches. Such approaches, inspired by the literature on fuzz testing, don't repeatedly sample a generator in a loop; instead, they keep track of inputs that were somehow "interesting" and mutate them slightly in the hopes of uncovering yet more interesting ones. What's considered interesting can also vary: in FuzzChick [11] they instrument the predicate under test and keep track of branch coverage; in FuzzFactory [20] they allow for more general user-provided instrumentation; and in Targeted Property-Based Testing [14] they attempt to maximize some user-provided quantity. Another example lies with integrated shrinking approaches, such as Hedgehog [25] or Hypothesis [16], which don't minimize counterexamples using a best-first search approach; instead, they leverage the generators themselves to generate smaller inputs.

Each of these frameworks introduced new ideas that could greatly improve the effectiveness of testing, but their broad acceptance is limited by the fact that each is implemented as a standalone library, rather than as a modular plugin to existing PBT machinery. That is not an accident: it is a fundamental limitation of a choice in the design of the property language; it is a fundamental limitation of using a *shallow embedding* to express properties.

For concreteness, let's revisit our running example of a simple roundtrip property in QuickCheck. As discussed earlier, to define a property users must specify predicates as well as other aspects of the property such as the way to generate inputs or minimize counterexamples. QuickCheck (and many subsequent PBT frameworks) made a natural and extremely convenient choice: allow users to specify these in the host language, using familiar host features. To achieve this, QuickCheck authors came up with an ingenious design: they baked their property runner in the definition of their type of properties.

A similar design choice is pervasive across the literature. In other words, beginning with QuickCheck, the language of properties is represented using a *shallow embedding*: host-language functions that are familiar to write, easy to execute, but otherwise impossible to inspect (e.g. via pattern matching). As a result, frameworks hard-code property runners by engineering rigid property runners, with only a handful of predefined hooks for user configuration. That is, users are limited to *exactly* what library authors had the prescience to account for when building the tool.

In this work, we challenge this design decision, and present a new way of designing property-language APIs. Instead of using a shallow embedding (like QuickCheck), we introduce a novel style of mixed embeddings which we call *deferred binding abstract syntax*, where rather than binding a variable once at the site of its universal quantification, we will instead bind it at every one of its use sites. Following other work on deeply embedded domain-specific languages [7, 8, 23, etc.], we design a language for properties using this embedding style that is reified as a data structure and then interpreted later on. This allows us to *decouple the specification from the runner.* The property itself expresses only the specification and the runner can be programmed by the user to interpret the property—allowing for maximum programmability.

Concretely, we offer the following contributions:

- We introduce a new style of mixed embeddings which is particularly well-suited for expressing properties for testing, separating specifications from testing configurations and allowing for arbitrary re-interpretation of the way properties are executed (Section 2).
- We implement our property language API in the QuickChick property-based testing framework for Rocq, highlighting the value of dependent types for ensuring that properties are well-formed, and describing how to retain ergonomics in the presence of dependent types (Section 3).
- We also implement our property language API as part of a new PBT framework in Racket, leveraging dynamic typing and macros to hide the internal data structure, providing an identical user interface to existing libraries, while enabling flexibility through the deeper embedding (Section 4).
- We demonstrate the flexibility of this new language by implementing a variety of complex runners from the recent literature—including ones with coverage-guided fuzzing and context-sensitive shrinking—all in user code. We also evaluate our implementations with 4 case studies using and extending the ETNA [24] evaluation platform. We show that testing with the deeper embedding does not incur any performance overhead, while providing significant flexibility to users that can directly translate to more effective testing (Section 5).

We conclude with related (Section 6) and future (Section 7) work.

2 A MIXED PROPERTY LANGUAGE

Properties are simply *universally quantified predicates.* We want to define a layer of such properties on top of some standard host language, equipped with at least booleans:

$$\begin{array}{rcl} \tau & := & \texttt{Bool} \mid \tau \to \tau \\ e & := & x \mid \lambda x.e \mid e\ e \mid T \mid F \mid \ldots \end{array}$$

In practice, the host language needs to also support some kind of randomness for generation, lists for shrinking, strings for pretty printing, etc. For presentation purposes, let's begin by formally describing the core boolean structure:

$$p \quad := \quad \forall x : \tau.\, p \mid e$$

That is, properties are either universal quantifiers or an injection of a host predicate.

The core question we ask in this paper is: *how should we represent this language?* We'll start by reviewing existing embedding solutions from the literature, demonstrating why they are not flexible enough to allow for users to customize or specify their runners, before proposing our solution.

Background: Shallow Embedding. In established property-based testing frameworks, the answer is using a *shallow embedding*. That is, they represent binders for universal quantifiers in properties as a binder in the host language.

$$\begin{array}{lcl} \mathit{forall} & : & \forall \tau.\ (\tau \rightarrow \mathit{prop}) \rightarrow \mathit{prop} \\ \mathit{check} & : & \texttt{Bool} \rightarrow \mathit{prop} \\ \mathit{run} & : & \mathit{prop} \rightarrow \texttt{IO}\ () \end{array}$$

Concretely, the type of properties is some type *prop* in the host language (usually restricted via typeclasses) *check* injects a host boolean into this type, and *forall* takes a host-level function and turns it into a property. Finally, frameworks also provide a *run* function to test properties constructed using this API.

The main advantage of this approach is that it is very convenient for users to write properties (using host-level binders), and extremely easy to test them (simply invoking the framework-provided *run*). However, since the *prop* type is opaque, users can't inspect the structure of the property, can't pattern match on it, and can't customize or write their own runners.

Background: Deeper Embeddings. The polar opposite alternative would be to use a deep embedding, which would necessitate defining an inductive representation of both the host language and the language of properties. Such an option is a non-starter: while it would enable pattern matching and recursively traversing properties, it would be prohibitively cumbersome to host language users.

Naturally, these are not the only two embedding options—researchers have explored a wide range of *mixed embeddings*, that reify parts or aspects of the language being considered. For example, we could attempt a HOAS-style approach, where we define an inductive type of properties, with a constructor corresponding to *forall* and *check*, both indexed by the arguments of their shallow counterparts:[1]

$$\begin{array}{l} \mathit{Inductive\ Prop} := \\ \mid \mathit{Forall} : \forall \tau.\ (f :\ \forall x : \tau.\ \mathit{Prop}\ x).\ \mathit{Prop}\ f \\ \mid \mathit{Check} : \forall b : \texttt{Bool}.\ \mathit{Prop}\ b \end{array}$$

Such a representation still allows us to encode the universal quantifiers using host language binders, but also makes some headway into the issue at hand—given a property we can now pattern match against it! However, there is still a problem: to access the “rest” of the property that follows a universal quantification, we need an element of the type being quantified over; since such elements are going to be randomly generated, we can't actually recurse down the structure of the property without restricting ourselves to something along the lines of QuickCheck's `Gen`erator monad.

The key issue is that we're hiding the definition of the “rest” of the property under a host-language binder. Could we use standard host language constructs to define the predicates at the leaves of the property (the *Check*s), while retaining the ability to pattern match on the property structure?

Proposal: Deferred Binding Abstract Syntax. Our solution is what we call *deferred binding abstract syntax* (DBAS): rather than bind a variable once at the site of its universal quantification, we will instead bind it at every one of its use sites.

$$\begin{array}{l} \mathit{Inductive\ Prop}\ (\mathit{env} : [\mathit{Type}]) := \\ \mid \mathit{Forall} : \forall \tau.\ \mathit{Prop}\ (\tau :: \mathit{env}) \rightarrow \mathit{Prop}\ \mathit{env} \\ \mid \mathit{Check} : (\llbracket \mathit{env} \rrbracket \rightarrow \texttt{Bool}) \rightarrow \mathit{Prop}\ \mathit{env} \end{array}$$

[1]We're abusing notation a bit here as any embedding would probably require additional type information to be admitted by a dependently typed language, but it's enough to showcase the problem.

We still define an inductive type of properties, with one constructor for each combinator in our API. We also index our type of properties by an environment: a list of types that have already been quantified. The key change is that we move the host level binder from the binding site (*Forall*) to its use site (the *Check*). That is, the argument to *Forall* is no longer a function, but simply a property with an extended environment; on the other hand, the argument to *Check* is no longer a simple boolean, but a function that binds *everything* in the environment. In the code above, we denote that as $[\![env]\!]$—we will see how it can be implemented in a statically typed or dynamically typed setting respectively in later sections.

At a first glance, this is a counter-intuitive trade-off: every time you want to use a variable, you have to bind *everything* that was quantified before that point. That would only make sense in a scenario where there are a lot of quantifications and few variable uses—which is precisely the case for the language of properties! Compared to the previous language representations, this DBAS-based one allows us to access the structure of the "rest" of the predicate without having access to a concrete value, which is necessary when such values are to be randomly generated. Finally, compared to the fully shallow representation, the added inductive structure and typing information pose some burden to the user experience, but once again, we'll address these issues in host-language-specific settings.

Adding Annotations. Generalizing the property language above to include annotations for generation, shrinking, or printing of individual elements is straightforward. We simply include an optional extensible list of annotations at the *Forall* constructor:

$$\begin{array}{l} \mathit{Inductive\ Prop}\ (\mathit{env} : [\mathit{Type}]) := \\ \mid \mathit{Forall} : \forall\tau.\ \mathit{as} \rightarrow\ \mathit{Prop}\ (\tau :: \mathit{env}) \rightarrow \mathit{Prop\ env} \\ \mid \mathit{Check} : ([\![\mathit{env}]\!] \rightarrow \mathtt{Bool}) \rightarrow \mathit{Prop\ env} \\ \\ \mathit{as} := \emptyset \mid (k, \forall\tau.\ [\![\mathit{env}]\!] \rightarrow \tau) :: \mathit{as} \\ k\ := \mathit{gen} \mid \mathit{shr} \mid \ \ldots \end{array}$$

At a high level, we can annotate each *Forall* constructor with a (possibly empty) sequence of (host-level) functions that quantify over the context so far (as in *Check*) and return annotation-specific terms (e.g. a generator or a shrinker). We demonstrate the exact implementation of such annotations to the following two host-language-specific sections.

3 A DEEPER DEPENDENTLY TYPED PROPERTY LANGUAGE

Implementing the language of universally quantified properties using deferred binding abstract syntax is straightforward on top of a dependently typed language, but achieving good ergonomics can be a challenge. In this section we will focus on implementing such a language on top of the QuickChick [10] framework for property-based testing in Rocq.

To that end, we explicitly encode contexts in our properties, capturing every input that *will have been generated* by that point. We will use a standard inductive definition of contexts, using ∅ to denote the empty context and · to extend a context by a type. Given a context, we can calculate the type corresponding to it: the type of tuples containing all of its types in order, with `unit` as the base case:

```
Inductive Ctx :=
| ∅ : Ctx
| · : Type → Ctx → Ctx.
```

```
Fixpoint interp (C : Ctx) : Type :=
  match C with
  | ∅ => unit
  | T · C => T * interp C
  end.
```

We will write [[C]] as a shorthand for `interp C`.

Now we can define a deeper version of the property language using DBAS:

```
Inductive Prop : Ctx ⟶ Type :=
| FORALL : forall {A: Type} {C: Ctx}
      (generator : ⟦C⟧ ⟶ G A)          (mutator : ⟦C⟧ ⟶ A ⟶ G A)
      (shrinker  : ⟦C⟧ ⟶ A ⟶ list A) (printer : ⟦C⟧ ⟶ A ⟶ string),
      Prop (A ·C) ⟶ Prop C
| IMPLIES : forall C
      (prop : ⟦C⟧ ⟶ bool),
      Prop C ⟶ Prop C
| CHECK : forall C,
      (⟦C⟧ ⟶ bool) ⟶ Prop C.
```

Just like the shallow approach of QuickCheck, this representation allows us to express, in the host language, type-based generators, mutators, shrinkers, and printers for each quantifier in a property. Just like the shallow approach, we can use typeclasses to automate much of the burden of specifying the property (as we will see below). Crucially, however, unlike the shallow approach we can pattern match on this definition and construct a wide range of methods for interpreting such properties without needing to modify the code of the underlying property-based framework at all.

For example, the standard "generate-and-run" loop of Figure 1, which amounts to interpreting such a property in the original shallow embedding, can be straightforwardly encoded as follows, first defining a simple type of *results* that holds information such as the `inputs` that were generated (calculated recursively from an input property `prop`):

```
Inductive RunResult {C: Ctx} (prop : Prop C) :=
| Normal : ⟦inputs prop⟧ ⟶ bool ⟶ RunResult prop
| Discard : ⟦inputs prop⟧ ⟶ RunResult prop.
```

And then the runner as a straightforward fixpoint: [2]

```
Fixpoint genAndRun (C : Ctx) (cprop : Prop C) : ⟦C⟧ ⟶ G (RunResult cprop) :=
  match prop with
  | FORALL A C name gen mut shr pri prop =>
      (fun A′ C′ name′ gen′ mut′ shr′ pri′ prop′ =>
      fun env =>
         a ⟵ gen′ env;;
         res ⟵ genAndRun (A′ ·C′) prop′ (a, env);;
         match res with
         | Normal env truth => (fun env′ truth′ =>
             ret (Normal (Some a, env′) truth′)) env truth
         | Discard env => (fun env′ =>
             ret (Discard (Some a, env′))) env
         end) A C name gen mut shr pri prop
  | CHECK C prop => (fun C′ prop′ =>
      fun env =>
      ret (Normal tt (prop′ env))) C prop
  | IMPLIES C pre prop => (fun C′ pre′ prop′ =>
      fun env =>
      if pre′ env then
```

[2]Branches follow the standard convoy pattern to enable type inference in dependent pattern matching, and we hide those in *gray*—they are unfortunate artifacts of Rocq's support for dependently typed programming.

```
        res ⟵ genAndRun C′ prop′ env;;
        match res with
        | Normal env truth => (fun env′ truth′ =>
            ret (Normal env′ truth′)) env truth
        | Discard env => (fun env′ =>
            ret (Discard env′) env
        end)
      else ret (Discard (nones prop′))) C pre prop
  end.
```

The flexibility to define such a loop at the hands of users allows for encoding all kinds of interpreters for properties, including pure generators, runners, shrinkers, fuzzers, with fully programmable execution, printing, and benchmarking options. We'll further demonstrate this flexibility by implementing a series of runners from the literature, all on top of this abstract property language.

Usable Defaults. A standard disadvantage of deep embeddings compared to shallow ones, is that they are generally less convenient to work with. Encoding everything in the host language, as in a shallow embedding, allows users to simply reuse a large part of host language infrastructure. The property language described above enables much of that using dependent types. Still, it is desirable to provide as seamless an experience for new users as possible, leveraging the same familiar typeclass-based interfaces of the shallow setting.

For concreteness, without any effort to provide such an experience, users would have to write the following to encode the roundtrip property of the introduction:

```
Definition prop_roundtrip_bad :=
  FORALL (fun tt => gen) (fun tt => mut)
         (fun tt => shrink) (fun tt => pretty) (
  ·CHECK (Expr ·∅) (fun '(e, _) => parse (pretty e) == Some e)).
```

That is, users would have to write a lot of annotations to achieve the same result, both at the type level (`Expr · ∅`) and to annotate individual `FORALL`s with the various generators, shrinkers, and printers.

However, we are not restricted to providing the core property definition as the final user-level interface! To that end, we present a simple surface-level language that allows users to write simplified properties, using typeclasses to fill in the remaining information. For example, the same property can be defined in our framework as in the much more straightforward snippet that follows:

```
  Definition prop_roundtrip :=
    ForAll e :- Expr,
    Check (fun 'e => parse (pretty e) == Some e).
```

In addition, users can override particular aspects of the property easily. For example, specifying a particular generator to be used, such as `gen` can be done as follows:

```
  Definition prop_roundtrip :=
    ForAll e :- Expr gen:gen,
    Check (fun 'e => parse (pretty e) == Some e).
```

Most of this surface language is achieved using Rocq's powerful notation mechanism, including its support for recursive notations. The final piece of the puzzle to simplify `Check` definitions relies on typeclasses. In particular, we associate each predicate with its corresponding context and a proof of that correspondence, in a class we name `Untuple`:

```
Class Untuple (A : Type) :=
  { untuple : Ctx
  ; untuple_correct : ⟦untuple⟧ = A }.
```

We then provide instances for the empty context and the bind:

```
Instance Untuple_empty : Untuple tt :=
  { untuple := ∅
  ; untuple_correct := eq_refl }.
Instance Untuple_pair {A B} `{Untuple B} : Untuple (A * B) :=
  { untuple := A ··untuple B _
  ; untuple_correct := ... }.
```

Before finally providing a convenient user-level wrapper for property conclusions, which we used above:

```
Definition Check {A} `{Untuple A} (p : A → bool) : Prop (·untuple A _).
  refine (Check (·untuple A _) _).
  rewrite untuple_correct.
  exact p.
Defined.
```

Finally, we can also leverage the extensive metaprogramming facilities of QuickChick to provide an even more seamless default when users only want to specify the predicate to be checked, which provides a no-effort starting point for newcomers:

```
Definition roundtrip (e : Expr) := parse (pretty e) == Some e.
Derive Property roundtrip.
(* ==> roundtrip_prop is defined. *)
```

This command constructs the deeper property above using the Rocq predicate itself.

4 A DEEPER DYNAMICALLY TYPED PROPERTY LANGUAGE

Representing the property language with deferred binding abstract syntax is not restricted to a dependently typed setting; in this section, we show how to implement it in a dynamically typed language like Racket. In a such a setting, we no longer have static guarantees about the types of the variables in the context; instead, we fall back on dynamic errors. However, we are also not encumbered by the type system, as we can freely invoke functions on arguments of (statically) unknown types, which we will fully take advantage of to recover most of the convenience of a shallow representation.

The first step is to directly translate the datatype into a series of `struct`s:

```
(struct Forall (var augments body))
(struct Implies (prop body))
(struct Check (prop))
```

In Racket, we cannot rely on typeclasses to automatically discover generators or shrinkers for property defined variables. Instead, we add a dictionary to the `Foralls` that allows us to attach extra information onto each variable that we call *augments*. Concretely, this dictionary maps augment names as Racket keywords to a function that takes the current environment of previously-generated variables and produces the augment value. These augments are fully generic in that they can store any values, though our implementation defines specific uses for three.

- `#:contract` attaches an invariant contract to values bound to the variable.

- `#:gen` attaches a generator for the variable. In order to make the usage of generators from other frameworks e.g. RackCheck easier, we intentionally treat the generator value as opaque. Instead, property interpretations that use the generators take a user-provided sampling function that is applied to the generator.
- `#:shrink` attaches a function used for shrinking counterexamples, it expects a function from values to lists of shrunk values.

However, handling the struct-based definitions directly involves a lot of explicit plumbing that we would rather not need to write. Consider once again the roundtrip property from the introduction:

```
(define roundtrip
  (Forall 'e (hash '#:contract (λ (env) expr?) '#:gen (λ (env) gen-expr))
        (Check (λ (env)
                 (let ([e (dict-ref env 'e)])
                   (equal? (parse (print e)) e))))))
```

There are two main ergonomic issues: the repeated nesting and the explicit environment passing and lookup. We can use Racket's extensive macro capabilities to create a DSL for writing these deeper properties. To that end, we flatten the nested structure by using the fact that properties are isomorphic to a list of `Forall` and `Implies` terminated by a single `Check`. Then, we use Racket's variable transformer macros to define and refer to generator variables as Racket identifiers and insert the dictionary passing plumbing for us.

These features allow us to translate the above roundtrip property into one much closer to how shallow embedding properties are written.

```
(define roundtrip
  (property
    (forall e #:contract expr? #:gen gen-expr)
    (equal? (parse (print e)) e)))
```

Property Runners. Developing property runners in the Racket setting shares much of the structure of the Rocq version, with some extra logic to attach the contract to the generated value if present.

```
(define (gen-and-run p sample . args)
  (let loop ([p p] [env (hash)])
    (match p
      [(Forall var augments body)
       ; Ensure the variable has a generator augment
       (unless (dict-has-key? augments '#:gen) (error 'no-generator))
       ; Generate a value using the sample function
       (define val (apply sample ((dict-ref augments '#:gen) env) args))
       ; Check the contract if present
       (when (dict-has-key? augments '#:contract)
         (invariant-assertion ((dict-ref augments '#:contract) env) val))
       ; Recur
       (loop body (dict-set env var val))]
      [(Implies prop body)
       (if (prop env)                ; Check precondition
           (loop body env)           ; If it passes, recur
           (values 'discard env))] ; If it fails, discard
      [(Check prop)
       (if (prop env)              ; Check result
           (values 'pass env)      ; Success
           (values 'fail env))]))) ; Failure
```

Encoding properties using deferred binding abstract syntax in Racket gives users the same variety in choices of property runners loops that the Rocq version does, as well as the same programmability for expert users. We utilize Racket's contracts to optionally allow users to enforce typed boundaries on generated variables. Racket's powerful macros enable us to write properties in a style that requires little syntactic overhead compared to shallow embeddings without sacrificing any of the programmability provided by deeper embeddings. In Section 5.2.2, we show that the extra programmability does not come at a performance cost, enables the writing of property runners that execute many runs in parallel, and shrinking loops that find significantly smaller counterexamples.

5 EVALUATION

In this section, our primary goal is to evaluate the flexibility and the expressiveness of representing properties using DBAS. To that end, we begin by showcasing how to straightforwardly implement a number of different property runners from the recent literature. In particular, we implement a basic generate-and-shrink runner (as in QuickCheck and QuickChick); a feedback-guided mutation runner (generalizing FuzzFactory, FuzzChick, and Targeted Property-Based Testing); and a parallelized property runner (inspired by the recent work in a parallel runtime for QuickCheck) [9]. Crucially, all runners are developed on top of the same property language, without the need to modify the internals of a framework—let alone to write a new one from scratch. We then carry out a sequence of case studies to evaluate various aspects of the performance of these implementations.

In our first case study, we compare the performance of the standard DBAS-based property runner as implemented in Rocq and Racket to the existing property runners of QuickChick and RackCheck. For benchmarking, we turn to the ETNA [24] framework for evaluating property-based testing performance, which comes with a series of Rocq workloads—programs along with injected bugs—in the form of Binary Search Trees, Red-Black Trees, and the Simply Typed Lambda Calculus. For the purposes of this case study we extended the ETNA tool to support Racket and ported these workloads. We found that using the DBAS-based embedding has *no observable performance overhead.*

In our second case study, we explore how different design choices with respect to the representation and sampling of the seed pool affect testing performance, reusing a case study from both FuzzChick [11] and Targeted Property-Based Testing [14]: testing information-flow control abstract machines. In both original case studies, the design choices around the search strategy and the power schedule of the feedback loop are not empirically justified; in large part, because making such choices involved making changes to the underlying infrastructure and would be hard to parameterize and benchmark. Here, we implement six different data structure representations for saving interesting inputs, along with four different energy scheduling strategies to explore the effect of such configurations in the performance of property-based testing tools. Even though we find that a particular heap-based implementation appears to be more efficient and we will nominate it as the default strategy in the next release of QuickChick and RackCheck, the key takeaway is that *such an exploration was not just possible, but very straightforward to undertake.*

In the third case study, we compare the default integrated shrinking capabilities of RackCheck with a simple external shrinker we implemented for the DBAS-style Racket library, using a (port of) the `System F` workload in ETNA for Racket. We show that our *external shrinker is significantly more effective at shrinking counterexamples*, once again illustrating the standard trade-off between usability (no need to write a shrinker) and effectiveness (size of minimized counterexamples) in internal vs external shrinking. Once again, we argue that this choice should be up to the user, and not fixed by the developer of a framework, which is exactly what a representation of properties based on DBAS provides.

In our last case study, we implement a parallel property-runner inspired by a recent work on parallelizing QuickCheck [9]. We implemented a worker queue based parallelization in Racket, outperforming the single threaded property-runner, all in just an evening's work, demonstrating the benefits of DBAS in enabling faster PBT experimentation.

5.1 Flexibility: Encoding Property Runners

Simple Generational Property Runner. We begin demonstrating the flexibility of our approach by re-implementing a series of property runners from the literature, starting with the natural one: the generate-test-shrink-test loop introduced by QuickCheck which was shown in Fig. 1. Throughout this section functions corresponding to the components in the pictorial diagram are overlayed with the same colors.

```
Definition runLoop (fuel : nat) (cprop : Prop ∅) :=
  let fix runLoop' (fuel : nat) (cprop : Prop ∅)
      (passed : nat) (discards: nat) : G Result :=
  match fuel with
  | O => ret (mkResult discards false passed [])
  | S fuel' =>
    input <- gen cprop (log2 (passed + discards));;
    res <- run cprop input ;;
    match res with
    | Normal seed false => (* Fails *)
      let shrunk := shrinkLoop 10 cprop seed in
      let printed := print cprop 0 shrunk in
      ret (mkResult discards true (passed + 1) printed)
    | Normal _ true => (* Passes *)
      runLoop' fuel' cprop (passed + 1) discards
    | Discard _ _ => (* Discard *)
      runLoop' fuel' cprop passed (discards + 1)
    end
  end in
  runLoop' fuel cprop 0 0.
```

Fig. 2. Simple Generational Property Runner in Rocq

Each component of the property runner in Fig. 1 has a clear correspondence to the corresponding overlayed sections in Fig. 2. The functions `gen`, `run`, `shrinkLoop`, and `print` are the building blocks of user level property-runners, but can also be written by users themselves in a straightforward manner as shown in Sections 3 and 4. We use them here to present runners at a higher level of abstraction as enabled by DBAS, focusing on how these components interact with each other in order to showcase users how to implement new runners corresponding to their needs.
The Simple Generational Property Runner composes of two tight loops, the first one runs `gen`, `run`, `gen`, `run` ...until a counterexample is found, or until a predefined limit of tests has been reached. The second loop runs `shrink`, `run`, `shrink`, `run` ...until it is not able to minimize the input further, reporting the smallest input within the shrinking process.

The code in Fig. 3 shows the same runner in Racket, with minor syntactical adjustments.

Mutation-Based Property Runners. Fuzzing techniques, and especially coverage-guided fuzzing, have proven to be very effective at finding bugs by focusing on *mutating* inputs that proved to be interesting rather generating new ones from scratch. Figures 4 and 5 pictorially represent two such runners: one for FuzzChick [11], which relies on branch coverage information obtained via binary instrumentation, and one for Targeted PBT [14], which relies on an explicit user-provided feedback function.

The coverage-guided (fuzzing) property runner, illustrated in Fig. 4, uses AFL instrumented code coverage as feedback, generating a new input from scratch or mutating a previous input based on the feedback, and is complemented with the shrinking loop in Simple Generational Property

```
(define (run-loop tests p)
  (let loop ([n 0] [passed 0] [discards 0])
   (if (= n tests) (result #f passed discards #f)
                   (let ( [env (generate p run-rackcheck-gen (floor (log n 2)))] )
                     (case (check-property p env)
                       [(fail) (result #t passed discards (shrink-eager p env) )]
                       [(pass) (loop (add1 n) (add1 passed) discards)]
                       [(discard) (loop (add1 n) passed (add1 discards))])))))
```

Fig. 3. Simple Generational Property Runner in Racket

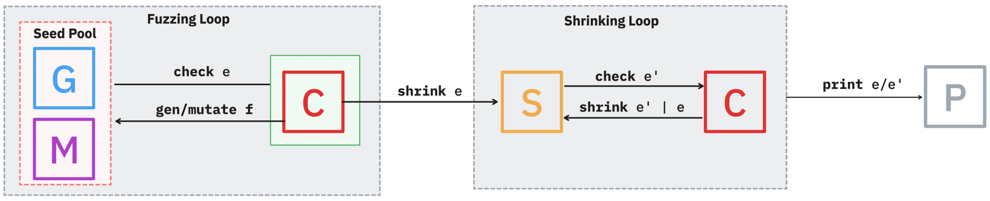

Fig. 4. Coverage-Guided (Fuzzing) Property Runner

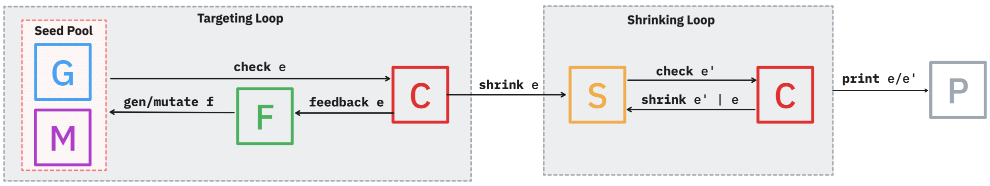

Fig. 5. Custom-Feedback Guided (Targeted) Property Runner

Runner illustrated in Fig. 1. The custom-feedback guided (targeted) property runner, illustrated in Fig. 5, shows our runner to take explicit user-specified feedback into account instead of some predefined notion of coverage.

We can straightforwardly implement such approaches as shown in the fuzzing property runner of Fig. 7. This runner is parameterized over a simple seed pool interface, which abstracts away search strategy (how to select which input to mutate) and power schedule (how long to fuzz it for) concerns. We will revisit this abstraction in just a few sections (Sec 5.3). The targeted runner is similar and can be found in Appendix A.1.

Parallel Runner. Inspired by QuickerCheck [9], we implemented a worker queue based parallelization in Racket, which is depicted in Fig. 6 The workers share a common size variable, which they atomically increment at each concurrent iteration, allowing near-linear parallel scalability. The code for the Racket parallel runner can be found in the appendix (Sec A.2).

5.2 Comparison of Deep and Shallow Embeddings

Background: ETNA Bucket Charts. Throughout the sequence of case studies that follow, our performance results will use ETNA bucket charts: each bucket represents the tasks (mutant-property pairs) solved within a certain time limit in the average of 10 trial runs. The leftmost bucket

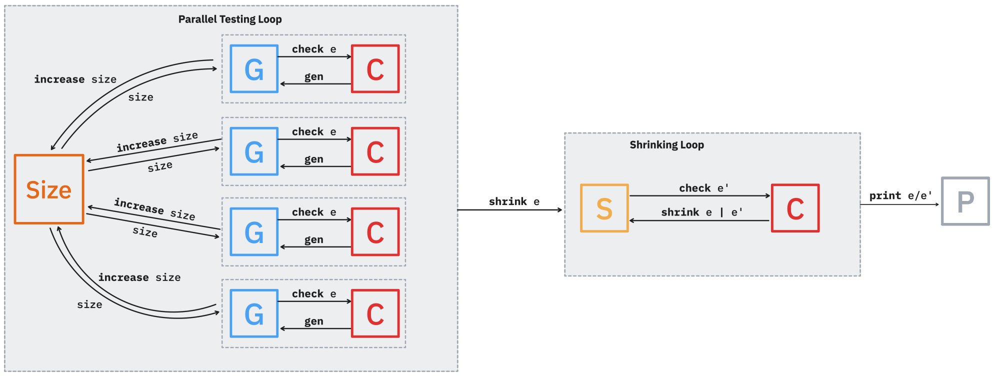

Fig. 6. Parallel Property Runner

with the darkest color denotes the tasks solved within 0.1 seconds, where the remaining buckets progressively denote the tasks solved within 1, 10, 60 seconds, and the last bucket denotes the tasks that were not solved within 60 seconds for at least one of the 10 trial runs. The legend for these charts is shown in Fig. 8.

5.2.1 Comparison of Deep and Shallow Embeddings in Rocq. Our first case study focuses on the performance implications of using deferred binding abstract syntax instead of the standard generational runner. We benchmark our implementations of this loop for both Rocq and Racket against the existing loops of the QuickChick and RackCheck libraries in 3 ETNA workloads: binary search trees (BST), red-black trees (RBT), and the simply-typed lambda calculus (STLC). Our results show that libraries implemented via DBAS is on par with both QuickChick and RackCheck, where the property-runner we implemented is a user level construct as shown in Fig. 1 in § 1.

The bucket charts in Fig. 12 show that using DBAS does not incur a performance penalty compared to QuickChick in the BST, RBT, and STLC workloads. In a total of 12 strategy/workload pairs, DBAS-style Rocq library outperforms QuickChick in 9 of them, while QuickChick has a better performance in 3 of them. Yet, there are no significant differences in the results of the two libraries in terms of mean time to solve the tasks.

5.2.2 Comparison of Deep and Shallow Embeddings in Racket. In our Racket experiments, we focused on comparing the DBAS-style Racket library with the shallow embedding implemented by RackCheck. We have conducted our experiments on the same BST, RBT, and STLC workloads by porting the existing workloads in Haskell to Racket. In the process of porting these workloads, we have also discovered and reported a bug in the RackCheck core, which the authors have fixed in the latest version of the library. [3]

Fig. 16 shows the results of the comparison of the deep and shallow embeddings in Racket. Once again, we find no significant differences in performance between the two versions. However, during the course of these experiments we found that the default configuration of RackCheck in terms of size of generated terms led to *significant* performance degradation in the RBT case study. As size is configurable in most PBT APIs, we have changed the RackCheck size function to be logarithmic with respect to number of tests, as our property-runner does. Still, this further reinforces our point on programmability: if sizes and similar aspects of generation are configurable (and severely impact testing performance), why shouldn't the runners themselves?

[3]The deanonymized version of our paper will have a citation to the bug report

```
Definition fuzzLoop (fuel : nat) (cprop : Prop ∅) {Pool}
  {pool: SeedPool } (seeds : Pool) : G Result :=
  let fix fuzzLoop' (fuel passed discards: nat) seeds :=
  match fuel with
  | O => ret (mkResult discards false passed [])
  | S fuel' =>
    let directive := sample seeds in
    input ⟵ match directive with
      | Generate => gen cprop (log2 (passed + discards))
      | Mutate source => mutate cprop source
    end;;
    res <- instrumentedRun cprop withInstrumentation;;
    let '(res, feedback) := res in
    match res with
    | Normal seed false => (* Fails *)
      let shrunk := shrinkLoop 10 cprop seed in
      let printed := print cprop 0 shrunk in
      ret (mkResult discards true (passed + 1) printed)
    | Normal seed true => (* Passes *)
      match useful seeds feedback with
      | true =>
        let seeds' := invest (seed, feedback) seeds in
        fuzzLoop' fuel' (passed + 1) discards seeds'
      | false =>
        let seeds' := match directive with
          | Generate => seeds
          | Mutate _ => revise seeds
        end in
        fuzzLoop' fuel' (passed + 1) discards seeds'
      end
    | Discard _ _ => (* Discard *)
      match directive with
      | Generate => fuzzLoop' fuel' passed (discards+1) seeds
      | Mutate source =>
        match useful seeds feedback with
        | true =>
          fuzzLoop' fuel' passed (discards+1) seeds
        | false =>
          fuzzLoop' fuel' passed (discards+1) (revise seeds)
        end
      end
    end
  end in
  fuzzLoop' fuel 0 0 seeds.
```

Fig. 7. Coverage-Guided Fuzzing Property Runner in Rocq

The coverage-guided fuzzing property runner introduces some complexity on top of the simple generational property runner, this complexity is mainly related to the orchestration logic that manages the seed pool, which is parametric over the SeedPool interface. At each iteration of the loop, the seed pool produces a directive, either to generate an input from scratch, or mutate a previous input. The generated input is then passed into `instrumentedRun` function that is also parameterized over a custom instrumentation function.

In classic fuzz testing, this instrumentation function is branch or path coverage, yet our Rocq library can accommodate any function that observes information about the state of the executed program, as in Padhye et al. [20]. This is reflected in the fuzzing loop in Fig. 7 where feedback is received from the execution of the `instrumentedRun`. Such feedback can range from traditional branch or path coverage (as in coverage-guided fuzzing) to timing or memory usage (as in performance fuzzing [12]).

This way, we view the fuzzing property runner presented in Fig. 7 as (1) a more generalized version of the classic coverage-guided fuzzing, and (2) a property-based testing version of the FuzzFactory [20], which allowed for arbitrary instrumentation functions to guide the search similar to the fuzzing property runner we present here.

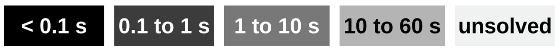

Fig. 8. ETNA Style Bucket Chart Legend

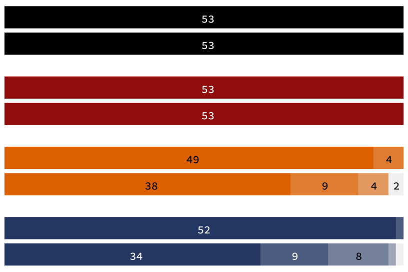

Fig. 9. Binary Search Trees

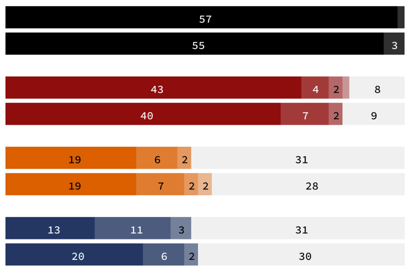

Fig. 10. Red-Black Trees

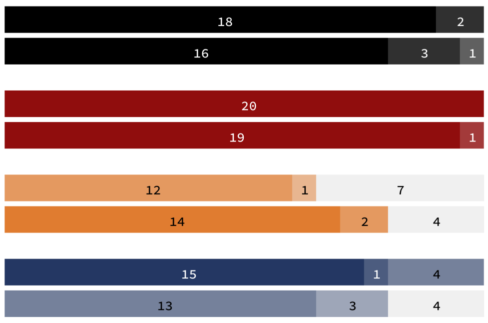

Fig. 11. Simply-Typed Lambda Calculus

Fig. 12. Comparison of Shallow and Deep Embeddings in Rocq. Each color denotes a strategy, where the top bar is DBAS and the bottom bar is the shallow behavior.
⬛ = Bespoke Generator, 🟥 = Specification-Based Generator,
🟧 = Type-Based Fuzzer, 🟦 = Type-Based Generator.

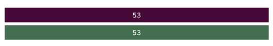

Fig. 13. BST

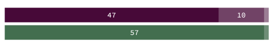

Fig. 14. RBT

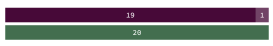

Fig. 15. STLC

Fig. 16. Comparison of Shallow(Rackcheck) and Deep(Tartarus) Embeddings in Racket. The purple bar on the top is Tartarus with deep embedding, the green bar on the bottom is Rackcheck with the shallow embedding. 🟪 = Bespoke Generator used with Tartarus Library, 🟩 = Bespoke Generator used with Rackheck Library.

5.3 An Exploration of Seed Pool Design Choices

Our second case study focuses on the programmability of the feedback-guided property runners. Fuzzing research is ripe with exploring different strategies and power schedules [1, 2, 4, 6], with researchers coming up with better and better designs. On the other hand, property-based testing frameworks that support feedback-guided generation of inputs such as FuzzChick or Hypothesis, which pride themselves in the power of testing arbitrary user-defined specifications, do not provide an option to configure such crucial parameters of their feedback-guided property runners. We illustrated two such feedback-guided runners earlier in this section, that abstract away such concerns into a small API, which we implement in Rocq using typeclasses (Fig. 17).

```
Class SeedPool {A F Pool: Type} := {
  (* Creates an empty pool. *)
  mkPool  : unit → Pool;
  (* Adds a useful seed into the pool. *)
  invest  : (A * F) → Pool → Pool;
  (* Decreases the energy of a seed after
     a useless trial. *)
  revise  : Pool → Pool;
  (* Samples the pool for an input. *)
  sample  : Pool → ·Directive A F;
  (* Returns the best seed in the pool. *)
  best    : Pool → option (·Seed A F);
}.
```

```
Class Utility {A F Pool: Type}
              `{SeedPool A F Pool} := {
  (* Returns true if the feedback
     is interesting. *)
  useful  : Pool → F → bool;
  (* Returns a metric of how interesting
     the feedback is. *)
  utility : Pool → F → Z;
}.
```

Fig. 17. SeedPool and Utility typeclasses used in Feedback-Guided Property Runners

The configurability enabled by DBAS allows the users to rely on a set of community-accepted defaults chosen by framework developers, but also to explore if different choices from the literature or novel ones they devised fit their testing needs better. In this case study, we explore six different data structure representations to hold interesting seeds, as well as four different power schedules, leading to a total of 21 different configurations. We picked these configurations to explore different parts of the design space, such as the queuing strategy, the size/cardinality of the pool, whether the pool is monotonic, and how many times a given seed is reused. We have conducted our experiments on the IFC workload in ETNA, and we have used a type-based generator alongside a type-based mutator to conduct our experiments. While our experiment reveals a clear winner for this case study, we reiterate that our primary goal is not the exploration itself, but rather to demonstrate that such an exploration is not only feasible, but natural to carry out with DBAS-style PBT libraries.

More concretely, we explore the following data structure representations, three that hold collections of seeds (as in FuzzChick), and three that only hold a single seed (as in Targeted PBT):

(1) *FIFO Queue Seed Pool*: A pool that holds a queue of seeds, and reduces the energy of the current seed after usage. The pool only generates a new seed from scratch when the queue is empty, and mutates the seed otherwise. When the energy of the current seed drops to 0, it is removed from the queue. The next seed is chosen from the front of the queue. This was the default behavior of FuzzChick.
(2) *FILO Queue Seed Pool*: The same as the *FIFO Queue Seed Pool*, but the next seed is chosen from the back of the queue.
(3) *Heap Seed Pool*: Similar to the FIFO and FILO Queues, but the seeds are stored in a heap, creating a priority queue.
(4) *Static Singleton Pool*: A pool that holds a single seed, and does not reduce its energy after usage. The pool generates a new seed from scratch at the first iteration, and mutates its seed for the subsequent iterations. The seed is only updated when a new seed with a better feedback is generated via mutation. This essentially devolves the search to hill climbing, as in the original Targeted PBT work [14].

(5) *Dynamic Monotonic Singleton Pool*: A pool that holds a single seed, and reduces its energy after usage. The pool generates a new seed from scratch at the first iteration and when the energy of the current seed is 0, mutates the seed otherwise. The seed is only updated when a new seed with a better feedback is generated.
(6) *Dynamic Resetting Singleton Pool*: A pool that holds a single seed, and reduces its energy after usage. As opposed to the *Dynamic Monotonic Singleton Pool*, once the current seed's energy is 0, the seed is effectively discarded and a new seed is generated from scratch.

All of the queues except the *Static Singleton Pool* have been tested with 4 different energy schedules, where the energy of the seed was respectively up to 1, 10, 100, 1000, depending on its interestingness. We report the experiments over 5 trials for each configuration within 65 tasks in the IFC workload in ETNA. For brevity, the graphs omit 34 tasks none of the configurations have solved, and only show the remaining 31 tasks. Each bucket represents the tasks solved within a certain time limit for at least one of the 5 trials, where the leftmost bucket with the darkest color denotes the tasks solved within 0.1 seconds, where the remaining buckets progressively denote the tasks solved within 1, 10, 60 seconds, and the last bucket denotes the tasks that were not solved within 60 seconds.

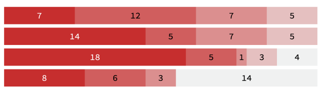

Fig. 18. Heap Seed Pool

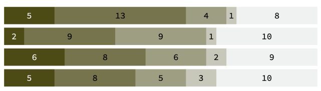

Fig. 19. FILO Queue Seed Pool

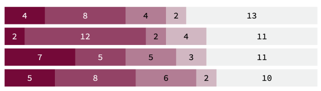

Fig. 20. FIFO Queue Seed Pool

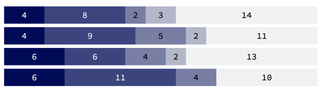

Fig. 21. Dynamic Motononic Singleton Seed Pool

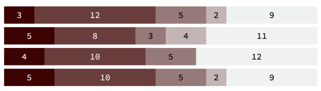

Fig. 22. Dynamic Resetting Singleton Seed Pool

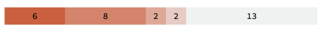

Fig. 23. Static Singleton Seed Pool

Fig. 24. Comparison of Seedpool Strategies in Rocq. Each color denotes a strategy, where the bars are ordered from energy levels 1, 10, 100, and 1000; except for the static singleton pool that ignores energy.

These results mark a task as "solved" for the purposes of a bucket chart if at least 1 out of 5 fuzzing campaigns finds a bug. If we switched to requiring *all* fuzzing campaigns to find the bug, the results paint an even more compelling argument: *only* Heap-based pools consistently find counterexamples—we omit the graphs because all other options fail to consistently solve a task across runs, indicating a very high variance that heap seed pool does not exhibit. As a result, we plan to advise QuickChick authors to use the effective configuration as a default going forward.

5.4 Comparison of Integrated Shrinking with External Shrinking

RackCheck uses an integrated shrinking mechanism that shrinks inputs by leveraging the generators, rather than passing the inputs to an external shrinker. In this case study, we show that the convenience of integrated shrinking sometimes comes at the cost of shrinking effectiveness, suggesting that developers can benefit from a DBAS-style property language that allows for integrated shrinking but does not enforce it.

The structure of the property runners for frameworks using integrated shrinking instead of external shrinking is very similar to the simple generational property runner, with the small change that they leverage generators with smaller sizes instead of an explicit shrinking function.

We have conducted our experiments on the `System F` workload in ETNA. We used the same generator in both RackCheck and DBAS-style Racket library, equipping our library with a simple type-based external shrinker we implemented in place of the integrated shrinker. Our results show that the the external shrinker successfully shrinks System F terms to a minimal counterexample that is an average of 2.66 times smaller than the original input with a standard deviation of 1.23, while the integrated shrinker of RackCheck only shrinks the inputs to an average rate of 1.04 times smaller than the original input with a standard deviation of 0.37. RackCheck only shrunk the inputs to smaller inputs in 66 out of the 360 trials, kept the size the same in 77, grew the inputs in 68, and failed to shrink in 149 trials. Fig. 25 depicts these results, which might be surprising at first glance: does the internal shrinker really only shrink the inputs in %20 of the trials? It turns out that in this case, it does. Its notion of size is based on the structure of the generator rather than on the input itself, so "smaller randomness" may not actually lead to smaller input values.

The point of this experiment is not to demonize internal shrinking—for many testing situations, it is perfectly sufficient and much more user-friendly than a bespoke shrinker. However, in pathological cases, programmers need an escape hatch, and DBAS provides the necessary configurability.

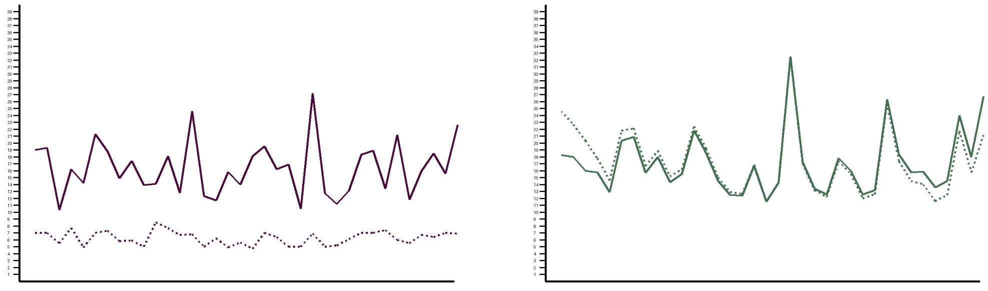
Fig. 25. Original sizes (continuous) and shrunk sizes (dashed) of external (left) and internal shrinker (right).

5.5 Parallelizing Property-Based Testing

An important recent work on novel property runners is QuickerCheck [9], where authors implement and evaluate a parallel run-time for QuickCheck [5], achieving massive performance gains in a variety of workloads. The underlying idea the authors propose is rather simple, they provide an alternative, parallel property runner `quickCheckPar`, where multiple worker threads share a common variable that controls the size of the generated inputs. Using this strategy, the authors achieve an almost linear speed-up with respect to the number of physical cores used in testing. Unfortunately, due to the rigid structure of shallow embedding based PBT libraries, creating such a parallel runner requires intensive engineering effort. The commit history of the project shows that the implementation of this new runner is an effort spanning 2 years and more than 50 commits.

In contrast, by leveraging the ability to implement property runners in user space, we were able to implement a naive version of the QuickerCheck algorithm in a matter of hours for our DBAS-style Racket library, with no prior knowledge of multicore Racket. Our implementation uses worker threads with 2 shared atomic variables, one keeping the current number of tests across threads, and one indicating if the process of testing is finished or not.

We have compared this parallel runner with 4 threads against the single threaded runner across 131 tasks (53 BST, 58 RBT, 20 STLC). As the results of the simple generational property runner evaluation in Racket 5.2.2 shows, the majority of the tasks in ETNA [24] are trivial for bespoke generators, they can be solved within 0.1 seconds. In order to measure the impact of the parallelization in the presence of such variation across tasks, we apply a set of cutoffs of time differences between task solving performance, and report the average ratio of time to solve for each cutoff.

- For all tasks where the difference between time-to-failure for single and parallel runners is greater than a millisecond, the average of their ratios is 1.2, where the parallel runner is *on average* takes 20% more time than of the single threaded runner across 49 of the 131 tasks.
- For all tasks where the difference between time-to-failure for single and parallel runners is greater than 0.1 seconds, the average of their ratios is 0.33, where the parallel runner is *on average* 3 times faster than the single threaded runner across 10 of the 131 tasks.
- Only 1 task has a difference of more than 1 second, which the average time-to-failure for single threaded runner is 3.56 seconds, and the average time-to-failure for the parallel runner is 1.16 seconds, resulting in again a 3 times speed-up due to parallelization.

6 RELATED WORK

Throughout this paper, we have thoroughly discussed various property-based testing frameworks, their property languages, and the property runners that they come bundled with. Here, we briefly summarize related work in PBT and we also discuss related work in deeply embedded domain-specific languages.

Property-Based Testing Frameworks. To our knowledge, there are no existing PBT frameworks that give users full access to re-program the property runner. However, there are multiple frameworks that make distinctly different choices in what capabilities they provide to users. Table 1 summarizes the status quo in popular tools.

Framework	**Language**	**Shrinking**	**Feedback**
QuickCheck [5]	Haskell	External	
Hedgehog [25]	Haskell	Internal	
QuickChick [21]	Rocq	External	Coverage [11]
Hypothesis [16]	Python	Internal [15]	Coverage*
Zest [18]	Java	AFL Trimming	Customizable[20]
QCheck	OCaml	Both*	
Crowbar	OCaml	AFL Trimming	Coverage
RackCheck	Racket	Internal	
QuviQ QuickCheck	Erlang	Both	
PropEr	Erlang	Both	Targeting [14]

Table 1. Shrinking and feedback options in popular frameworks; * denotes experimental or partial support.

Free Generators. The design of our deeply embedded property language builds on a rich literature of embedded DSLs [8]. In particular, our approach parallels work on *free generators* [7], which present a deeply embedded DSL for writing random generators. Deeply embedded properties and generators are largely orthogonal—free generators may be able to simplify the implementation of some of the different runners described in Section 5, but they do not allow the developer to re-program the loop itself. This follows a more general trend of using free structures to increase expressivity or usability in the programming languages community. For example, itrees [28] introduced a general-purpose Rocq structure which is essentially a coinductive variant of free monads, which allows them to represent and reason about interactive recursive programs. In follow up work, Li and Weirich [13] explored how other free structures (such as applicative functors) can be used as an alternative to free-monad-based embeddings.

Mixed Embeddings. There is also a long line of related work in attempting to bridge the benefits of shallow embeddings (ease of use, as in the current property language) with those of deep ones (extensibility). For example, Carrete et al. [3] showed how to use typeclasses in Haskell to allow for shallow embeddings that can be interpreted in different ways, hinting at a way of incorporating a deeper property language in a type system such as Haskell's. More recently, Matsuda et al. [17] showed how to convert between embedding representations by *unembedding* alleviating some of the problems with Higher-Order Abstract Syntax representations. Finally, Prinz et al. [23] introduced a hybrid embedding where typing derivations are represented as a deep embedding indexed by shallow terms in the host language, offering pattern matching capabilities.

7 CONCLUSION AND FUTURE WORK

We have presented deferred binding abstract syntax (DBAS), a new approach to writing properties for PBT that enables more flexible and programmable testing. The key advance made by DBAS is to reify properties as a free data structure; allowing them to be written in a clear and readable way, separate from the property runner that tests them. These more deeply embedded properties can then be inspected and interpreted by user-defined property runners. With the help of DBAS, developers in Rocq, Racket, and hopefully soon other programming languages, can tailor and experiment with their setup to achieve optimal testing in their domain.

In the future, we intend to make DBAS convenient to program in languages with more standard type systems than Rocq and Racket. In particular, Haskell is conspicuously missing from the list. The main challenge in languages like Haskell is implementing contexts in a user-friendly manner. In Haskell we can't represent contexts with dependent types the way we implement them in Rocq, but it is also too strongly typed for the looser approach we took in Racket. Modern advances in dependently typed Haskell [27] may actually provide the power we need, but there are other mainstream programming languages that lack this expressive power. Additional work might be necessary to bring the full potential of DBAS to more users.

8 DATA AVAILABILITY STATEMENT

All our work will be made publicly available. We intend to submit an artifact for artifact evaluation that includes the implementations of the property language in Rocq and Racket, as well as scripts to re-execute the experiments carried out.

REFERENCES

[1] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. 2020. Boosting fuzzer efficiency: an information theoretic perspective. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 678–689. https://doi.org/10.1145/3368089.3409748

[2] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2019. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Trans. Software Eng.* 45, 5 (2019), 489–506. https://doi.org/10.1109/TSE.2017.2785841

[3] Jacques Carrete, Oleg Kiselyov, and Chung-Chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 5 (2009), 509–543. https://doi.org/10.1017/S0956796809007205

[4] Jinfu Chen, Shengran Wang, Saihua Cai, Chi Zhang, Haibo Chen, Jingyi Chen, and Jianming Zhang. 2023. A Novel Coverage-guided Greybox Fuzzing based on Power Schedule Optimization with Time Complexity. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 172, 5 pages. https://doi.org/10.1145/3551349.3559550

[5] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*. ACM, 268–279. https://doi.org/10.1145/351240.351266

[6] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies (WOOT'20)*. USENIX Association, USA, Article 10, 1 pages.

[7] 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

[8] Paul Hudak. 1996. Building domain-specific embedded languages. *Comput. Surveys* 28, 4es (Dec. 1996), 196. https://doi.org/10.1145/242224.242477

[9] Robert Krook, Nicholas Smallbone, Bo Joel Svensson, and Koen Claessen. 2024. QuickerCheck: Implementing and Evaluating a Parallel Run-Time for QuickCheck. arXiv:2404.16062 [cs.PL] https://arxiv.org/abs/2404.16062

[10] Leonidas Lampropoulos. 2018. *Random Testing for Language Design*. Ph. D. Dissertation. University of Pennsylvania.

[11] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage Guided, Property Based Testing. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*. https://doi.org/10.1145/3360607

[12] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/3213846.3213874

[13] Yao Li and Stephanie Weirich. 2022. Program adverbs and Tlön embeddings. *Proc. ACM Program. Lang.* 6, ICFP (2022), 312–342. https://doi.org/10.1145/3547632

[14] Andreas Löscher and Konstantinos Sagonas. 2017. Targeted Property-Based Testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 46–56. https://doi.org/10.1145/3092703.3092711

[15] David Maciver and Alastair F. Donaldson. 2020. Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer (Tool Insights Paper). In *34th European Conference on Object-Oriented Programming (LIPIcs, Vol. 166)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 13:1–13:27. https://doi.org/10.4230/LIPIcs.ECOOP.2020.13

[16] David R. MacIver. 2016. Hypothesis: Property-Based Testing for Python. https://hypothesis.works/.

[17] Kazutaka Matsuda, Samantha Frohlich, Meng Wang, and Nicolas Wu. 2023. Embedding by Unembedding. *Proc. ACM Program. Lang.* 7, ICFP, Article 189 (Aug. 2023), 47 pages. https://doi.org/10.1145/3607830

[18] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 329–340. https://doi.org/10.1145/3293882.3330576

[19] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Validity fuzzing and parametric generators for effective random testing. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 266–267. https://doi.org/10.1109/ICSE-COMPANION.2019.00107

[20] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: domain-specific fuzzing with waypoints. *Replication Package for "FuzzFactory: Domain-Specific Fuzzing with Waypoints"* 3, OOPSLA (Oct. 2019), 174:1–174:29. https://doi.org/10.1145/3360600

[21] Zoe Paraskevopoulou, Catalin Hritcu, Maxime Denes, Leonidas Lampropoulos, and Benjamin C. Pierce. 2015. Foundational Property-Based Testing. In *6th International Conference on Interactive Theorem Proving (ITP)*. https://doi.org/10.1007/978-3-319-22102-1_22

[22] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2021. Generative type-aware mutation for testing SMT solvers. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–19. https://doi.org/10.1145/3485529

[23] Jacob Prinz, Alex Kavvos, and Leonidas Lampropoulos. 2022. Deeper Shallow Embeddings. In *13th International Conference on Interactive Theorem Proving (ITP) (Lecture Notes in Computer Science)*. https://doi.org/10.4230/LIPIcs.ITP.2022.28

[24] 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

[25] Jacob Stanley. 2019. Hedgehog: Release with Confidence. https://hackage.haskell.org/package/hedgehog/.

[26] Jingling Sun, Ting Su, Jiayi Jiang, Jue Wang, Geguang Pu, and Zhendong Su. 2023. Property-Based Fuzzing for Finding Data Manipulation Errors in Android Apps. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 1088–1100. https://doi.org/10.1145/3611643.

3616286

[27] Stephanie Weirich, Pritam Choudhury, Antoine Voizard, and Richard A. Eisenberg. 2019. A role for dependent types in Haskell. *Proceedings of the ACM on Programming Languages* 3, ICFP (July 2019), 1–29. https://doi.org/10.1145/3341705

[28] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119

A APPENDIX

A.1 Custom-Feedback Guided (Targeted) Property Runner

In recent years, PBT tools and mutation based fuzzers have begun to find common ground. On one hand, fuzzing tools have been trying to move towards more and more structured generation of inputs as well as incorporate and encode more complex properties than simply "the program doesn't crash"[19, 22, 26]. On the other hand, we have seen a rise in the popularity of property-based testing tools that are able to guide the generation of inputs using feedback [11, 14, 18]. Following this trend, we have used DBAS to implement mutation-based generation, developing two property runners leveraging mutation and feedback.

A mutation-based targeted property runner has two important differences from the simple generational property runner described in Fig. 2: the feedback and the targeting. It uses a genetic algorithm based on a user provided custom feedback function for guiding the search towards interesting inputs, and it employs a user-provided or type-derived mutator functions to mutate the input it currently focuses on. The main benefit of such a runner is that it means developers can potentially avoid writing complex generators for complex types and preconditions.

The runner is further parameterized by a seed pool (to keep track of interesting inputs) and a utility function (to accommodate different types of feedback). As shown in the pioneering work of Padhye et al. [20], customizable feedback under user control can lead to very effective testing, and DBAS allows even more relevant choices to be made without needing to modify the internals of a framework. A pictorial depiction of the targeted property runner is illustrated in Fig. 5.

In Fig. 26, we provide an annotated Coq implementation of the custom-feedback guided (targeted) property runner in Coq. Similar to the simple generational runner or coverage-guided (fuzzing) runner in Section 5, this runner reuses the building blocks we have provided for writing novel property runners.

It is important to note that this specific runner is not a fixed part of our library, but merely a default behavior that is convenient and can be readily used. Alternative implementations may change the feedback behavior to accommodate feedback in the discard cases, or change the printing or shrinking behaviors. Even the seed pool and utility typeclasses are provided as sensible defaults and building blocks, rather than static design choices as is the case in shallow embedding based PBT frameworks.

```
Definition targetLoop (fuel : nat) (cprop : Prop \em)
  (feedback_function : ⟦{cprop}⟧− > Z) {Pool : Type}
  {pool : SeedPool} (seeds : Pool)
  (utility : Utility) : G Result :=
  let fix targetLoop' (fuel : nat)
  (passed : nat) (discards: nat)
  (seeds : Pool) : G Result :=
  match fuel with
  | O => ret (mkResult discards false passed [])
  | S fuel' =>
    let directive := sample seeds in
    input ⟵ match directive with
      | Generate => gen cprop (log2 (passed + discards))
      | Mutate source => mutate cprop source
      end;;
    res <- run cprop input;;
    match res with
    | Normal seed false => (* Fails *)
      let shrunk := shrinkLoop 10 cprop seed in
      let printed := print cprop 0 shrunk in
      ret (mkResult discards true (passed + 1) printed)
    | Normal seed true => (* Passes *)
      let feedback := feedback_function seed in
      match useful seeds feedback with
      | true =>
        let seeds’ := invest (seed, feedback) seeds in
        targetLoop' fuel' (passed + 1) discards seeds'
      | false =>
          let seeds' :=
            match directive with
            | Generate => seeds
            | Mutate source => revise seeds
            end in
          targetLoop' fuel' (passed + 1) discards seeds'
        end
    | Discard _ _ => (* Discard *)
        targetLoop' fuel' passed (discards + 1) seeds
    end
  end in
  targetLoop' fuel 0 0 seeds pool utility.
```

Fig. 26. Custom-Feedback Guided (Targeted) Testing Loop in Rocq

The structure of the targeted property runner depicted in Fig. 5 is also reflected in the structure of the `targetLoop` in Fig. 26. This loop performs all the usual bookkeeping we discussed in the simple loop of Fig. 2, but adds mutation and feedback mechanisms guiding the search towards interesting inputs. The loop is parameterized by the `feedback function`, `the seed pool`, and `the utility function`, where the seed pool can be configured with different data structures such as a priority, FIFO, or FILO queue, and the utility function can be configured with different strategies such as a threshold, or a more complex stateful model. The loop uses the `Seed Pool` and `Utility` typeclasses to orchestrate the search.

Walking through the runner, we see that it diverges from the simple generational property runner in Fig. 2 in its input generation, where it might either `generate` from scratch, or `mutate` by `sampling` the seedpool. This input is then `run` through the property, and the failure and discarded cases are handled exactly the same as the simple generational property runner. If the test succeeds, depending on the feedback calculated by the user-provided feedback function, the seed might be `invested` in the seed pool, or the seed pool might be `revised` to reduce the energy of the seed. The loop then continues with the updated seed pool, and the passed and discarded counts.

A.2 Parallel Property Runner

Below is a slightly abridged version of our Racket parallel testing runner. Parallelism is done through Racket futures. We use a lock-free shared counter and a flag that is set if a counterexample is found, which is necessary because Racket futures are not able to be halted arbitrarily. Each worker thread grabs a test number from the counter, and loops until it either finds a counterexample or the counter exceeds the test number, with the main thread waiting for results from the workers and combining them when they finish.

Like above this loop is not a fixed part of the library, and more efficient or sophisticated parallel runners can be implemented without changing the underlying property representation.

```
(define (parallel-run-loop tests prop [num-workers (processor-count)])
  ; atomic counter for the test number
  (define counter (box 0))
  ; flag set if a thread finds a counterexample
  (define found-counterexample? (box #f))
  ; function called by each thread
  (define (worker-thunk)
    ; each thread creates its own random number generator
    (define rng (make-pseudo-random-generator))
    (let worker-loop ([passed 0]
                      [discards 0])
      ; fetch and increment the test number counter
      (define n (box-faa! counter 1))
      (cond
        ; if the number of tests has exceeded the total, return the thread results
        [(>= n tests) (results #f passed discards #f)]
        ; if another thread has found a counterexample, return the thread results
        [(unbox found-counterexample?) (results #f passed discards #f)]
        [else
        ; run a single test
         (let-values ([(res env) (gen-and-run prop run-rackcheck-generator rng n)])
           (case res
             ; if a counterexample was found, set the found flag
             ; and return the thread results
             [(fail)
              (set-box! found-counterexample? #t)
              (results #t passed discards env)]
             ; on pass or discard, increment the relevant counter and recur
             [(pass) (worker-loop (add1 passed) discards)]
             [(discard) (worker-loop passed (add1 discards))]))])))
  ; spawn workers
  (define workers
    (for/list ([_ (in-range num-workers)])
      (future worker-thunk)))
  ; read results from workers
  (for/fold ([res (results #f 0 0 #f)])
            ([worker workers])
    ; get results from this worker
    (define worker-res (touch worker))
    ; combine with previous worker results
    (results (or (results-foundbug? res) (results-foundbug? worker-res))
             (+ (results-passed res) (results-passed worker-res))
             (+ (results-discards res) (results-discards worker-res))
             (or (results-counterexample res) (results-counterexample worker-res)))))
```