Covering All the Bases: Type-based Verification of Test Input Generators

ANONYMOUS AUTHOR(S)

Test input generators are an important part of property-based testing (PBT) frameworks. Because PBT is intended to test deep semantic and structural properties of a program, the outputs produced by these generators can be complex data structures, constrained to satisfy properties the developer believes is most relevant to testing the function of interest. An important feature expected of these generators is that they be capable of producing *all* acceptable elements that satisfy the function's input type and generator-provided constraints. However, it is not readily apparent how we might validate whether a particular generator's output satisfies this *coverage* requirement. Typically, developers must rely on manual inspection and post-mortem analysis of test runs to determine if the generator is providing sufficient coverage; these approaches are error-prone and difficult to scale as generators become more complex. To address this important concern, we present a new refinement type-based verification procedure for validating the coverage provided by input test generators, based on a novel interpretation of types that embeds "*must*-style" underapproximate reasoning principles as a fundamental part of the type system. The types associated with expressions now capture the set of values *guaranteed* to be produced by the expression, rather than the typical formulation that uses types to represent the set of values an expression *may* produce. Beyond formalizing the notion of *coverage types* in the context of a rich core language with higher-order procedures and inductive datatypes, we also present a detailed evaluation study to justify the utility of our ideas.

1 INTRODUCTION

Property-based testing (PBT) is a popular technique for automatically testing deep semantic and structural properties of programs. Originally pioneered by the QuickCheck [1] library for Haskell, PBT frameworks now exist for many programming languages, including JavaScript [8], Rust [32], Python [15], Scala [33], and Coq [22]. The PBT methodology rests on two key components: *executable properties* that capture the expected input-output behaviors of the program under test, and *test input generators* that generate random values of the input types needed to validate these behaviors. In contrast to unit tests, which rely on single examples of inputs and outputs, generators are meant to provide a *family* of inputs against which programs can be tested, with the goal of ensuring the set of generated tests provide good coverage of all possible inputs. In order to prune out irrelevant inputs, PBT frameworks allow users to define custom generators that reflect the specific shape of data that the developer believes is most likely to trigger interesting (aka faulty) behavior. As one simple example, to test a tree compression or balancing function, the developer may want to use a generator that produces n-ary trees with randomly chosen height and arity but whose leaves are ordered according to a user-provided ordering relation.

Given the critical role they play in the assurance case provided by PBT frameworks, it is reasonable to ask what constitutes a "good" specification for a test generator. For our example, one answer could be that it should only produce ordered trees. Of course, this is not a very satisfactory characterization of the behavior we desire: the "constant" generator that always produces trees of height one trivially meets this specification, but it is unlikely to produce useful tests for a compression function! Ideally, we would like a generator to intelligently enumerate the space of *all* possible ordered trees, thereby helping to maximize the likelihood of finding bugs in the function under test. Because defining such an enumeration procedure for arbitrary datatypes can be hard, PBT frameworks instead give developers the ability to assemble generators for complex data structures *compositionally*, building on generators for simpler types where randomly sampling elements of the type is straightforward

2022. XXXX-XXXX/2022/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

and sufficient. For example, we could implement an ordered tree generator in terms of a primitive random number generator that is used to non-deterministically select the height, arity, and elements of a candidate tree, checking (or enforcing) the orderness of the tree before returning it as a feasible test input. However, although the random number generator might provide a guarantee that its underlying probability density function (PDF) is always non-zero on all elements in its sample space, determining that a tree generator that is built using it can actually enumerate all the ordered trees desired is a substantially harder problem. Even if we know the generator is capable of eventually yielding all trees, proving that any filtering operations it uses do not mistakenly prune out valid ordered trees or that any transformations it implements over candidate trees preserve the elements of the random tree being transformed, pose additional challenges. In other words, verifying that the generator is *complete* with respect to our desired orderness property entails reasoning that is independent of the behavior of the primitive generators used to build the tree. Consequently, we require some alternative mechanism to help qualify the part of the target function's input space the generator is actually guaranteed to cover. Devising such a mechanism is challenging precisely because the properties that need to be tested may impose complex structural and semantic constraints on the generated output (e.g., requiring that an output tree be a binary search tree, or that it satisfies a red-black property, etc.); the complexity of these constraints is directly correlated to the sparseness of the function's input space preconditions.

To illustrate this distinction more concretely, consider the input test generator shown in Figure 1 that is intended to generate all binary search trees (BSTs) whose elements are between the interval `lo` and `hi`. If we ignore the commented line, we can conclude this generator always produces a non-empty BST whenever `lo` < `hi`. While the generator is correct - it always generates a well-formed BST - it is also incomplete; the call `bst_gen 0 10`, for example, will never produce a tree containing just `Leaf` or a tree with a shape like `Node(1,Leaf,Leaf)`, even though these instances are valid trees consistent with the constraints imposed by the generator's argument bounds. In fact, this implementation *never* generates a BST that only contains a proper subset of the elements that reside within the interval defined by `lo` and `hi`. By uncommenting the commented expression, however, we allow the generator to non-deterministically choose (via operator ⊕) to either return a `Leaf` or left and right BST subtrees based on value returned by the `int_range` generator, enabling it to potentially produce BSTs containing all valid subsets of the provided interval, thus satisfying our desired desired completeness behavior. The subtleties involved in reasoning about such coverage properties is clearly non-trivial. We reiterate that recognizing the distinction between these two implementations is not merely a matter of providing a precise output type capturing the desired sortedness property of a BST: for example, the incomplete implementation clearly satisfies such a type! Furthermore, simply knowing that the underlying `int_range` generator used in the implementation samples all elements within the range of the arguments it is provided is also insufficient to conclude that the BST generator can yield *all* possible BSTs within the supplied interval. Similar observations have led prior work to consider ways to improve a generator's coverage through mechanisms such as fuzzing [6, 20], or to automatically generate complete-by-construction generators for certain classes of datatypes [21].

```
type 'a tree =
  | Leaf
  | Node of ('a * 'a tree * 'a tree)

let rec bst_gen (lo: int) (hi: int) : int tree =
  if lo + 1 >= hi then Leaf
  else (* Leaf ⊕ *) (
     let (x: int) = int_range (lo + 1, hi - 1) in
       Node (x, bst_gen lo x, bst_gen x hi))
```

Fig. 1. A BST generator. Failing to uncomment the commented line results in the generator never producing trees that contain only a subset of the elements in the interval between `lo` and `hi`, which is inconsistent with the developer's intent.

In contrast to these approaches, this paper embeds the notion of coverage as an integral part of a test input generator's *type* specification. By doing so, a generator's type now specifies the set of behaviors a generator is *guaranteed* to exhibit; a well-typed generator is thus guaranteed to produce *every possible* value satisfying a desired structural property, e.g., that the repaired (complete) version of `bst_gen` is capable of producing every valid BST. By framing the notion of coverage in type-theoretic terms, our approach neither requires instrumentation of the target program to assess the coverage effectiveness of a candidate generator (as in Lampropoulos et al. [20]) nor does it depend on a specific compilation strategy for producing generators (as in Lampropoulos et al. [21]). Instead, our approach can automatically verify the coverage properties of an *arbitrary* test input generator, regardless of whether it was hand-written or automatically synthesized.

Key to our approach is a novel formulation of a *must*-style analysis [11, 12, 16] of a test input generator's behavior in type-theoretic terms. In our proposed type system, we say an expression e has *coverage type* τ if every value contained in τ *must* be producible by e. Note how this definition differs from our usual notion of what a type represents: ordinarily, if e has type τ then we are allowed to conclude only that any value contained in τ *may* be produced by e. Informally, types interpreted in this usual way define an *overapproximation* of the values an expression e may yield, without obligating e to produce any specific such value. In contrast, coverage types define an *underapproximation* - they characterize the values an expression e must produce, potentially eliding other values that e may also evaluate to. In this sense, our solution can be seen a type-theoretic interpretation of recently developed underapproximate program logics [5, 25], in much the same way that refinement-type systems like Liquid Types [34] relate to traditional program logics [14].

Stated another way, a coverage type characterizes the *subset* of values an expression is *guaranteed* to generate, while a normal type characterizes the *entire* set of values an expression *may* yield. When the cardinality of an input generator's (underapproximate) coverage type matches that of its (overapproximate) normal type, however, we can soundly assert that the generator is complete. As we illustrate in the remainder of the paper, this characterization allows us to reason about a program's coverage behavior on the same formal footing as its safety properties.

This interpretation leads to a fundamental recasting of how types relate to one another: ordinarily, we are always allowed to assert that $\tau <: \top$. This means that any typing context that admits an expression with type τ can also admit that expression at a type with a logically weaker structure. In contrast, the subtyping relation for coverage types inverts this relation, so that $\top <: \tau$. Intuitively, $\top$ represents the coverage type that obligates an expression ascribed this type to be capable of producing *all* elements in τ. But, any context that requires an expression to produce all such elements can always guarantee that the expression will also produce a subset of these elements. In other words, we are always allowed to weaken an overapproximation (i.e., grow the set of values an expression may evaluate to), and strengthen an underapproximation (i.e., shrink the set of values an expression must evaluate to). In our setting, a random number generator over natural numbers has coverage type $\top_{\mathsf{nat}}$ under the mild assumption that its underlying PDF provides a non-zero likelihood of returning every natural number, while a faulty computation like `1 div 0` has coverage type $\bot$ since there are no guarantees provided by the computation on the value(s) it must return. Here, $\bot$ represents a type that defines a degenerate underapproximation, imposing no constraints on the values an expression ascribed this type must produce.

In summary, this paper makes the following contributions:

(1) It introduces the notion of *coverage types*, types that characterize the values an input test generator is guaranteed to (i.e., *must*) yield.

(2) It formalizes the semantics of coverage types in an ML-like functional language with support for higher-order functions and inductive datatypes[1].
(3) It develops a bi-directional type-checking algorithm for coverage types in this language.
(4) It incorporates these ideas in a tool (Poirot) that operates over OCaml programs equipped with input generators and typed using coverage types, and presents an extensive empirical evaluation justifying their utility, by verifying the coverage properties of both hand-written and automatically synthesized generators for a rich class of datatypes and their structural properties.

The remainder of the paper is structured as follows. In the next section, we present an informal overview of the key features of our type system. Section 3 presents the syntax and semantics for a core call-by-value higher-order functional language with inductive datatypes that we use to formalize our approach. Section 4 presents a type system for coverage types; a bidirectional typing algorithm is then given in Section 5. We describe details about the implementation of Poirot and provide benchmark results in Section 6. Related work and conclusions are given in Sections 7 and 8.

2 OVERVIEW

Before presenting the full details of our type system, we begin with an informal overview of its key features.

Base types. In the following, we write $[v{:}b \mid \phi]$ to denote the coverage type that qualifies the base type b using the predicate ϕ. As described in the previous section, an application of the primitive built-in generator for random numbers: `int_gen` : `unit` → `int` has the coverage type `int_gen` () : $[v{:}int \mid \top_{\text{int}}]$. We use brackets $[...]$ to emphasize that a coverage type has a different meaning from the types typically found in other refinement type systems [17, 34] where a qualified type b, written as $\{v{:}b \mid \phi\}$, uses a predicate ϕ to constrain the set of values a program *might* evaluate to. To illustrate this distinction, consider the following combinations of expressions and types[2] (the constant `err` represents a special error value, that when encountered, causes the program to halt):

`int_gen ()`	$\vdash [v{:}int \mid \top_{\text{int}}]$ $\vdash \{v{:}int \mid \top_{\text{int}}\}$	$\vdash [v{:}int \mid v = 1 \vee 2]$ $\nvdash \{v{:}int \mid v = 1 \vee 2\}$	$\vdash [v{:}int \mid v = 1]$ $\nvdash \{v{:}int \mid v = 1\}$	$\vdash [v{:}int \mid \bot]$ $\nvdash \{v{:}int \mid \bot\}$
1	$\vdash [v{:}int \mid v = 1]$ $\vdash \{v{:}int \mid v = 1\}$	$\vdash [v{:}int \mid \bot]$ $\nvdash [v{:}int \mid \top_{\text{int}}]$	$\vdash \{v{:}int \mid \top_{\text{int}}\}$ $\nvdash [v{:}int \mid v = 1 \vee 2]$	$\vdash \{v{:}int \mid v = 1 \vee 2\}$ $\nvdash \{v{:}int \mid \bot\}$
`err`	$\vdash [v{:}int \mid \bot]$ $\nvdash \{v{:}int \mid \top_{\text{int}}\}$	$\nvdash [v{:}int \mid \top_{\text{int}}]$ $\nvdash \{v{:}int \mid v = 1 \vee 2\}$	$\nvdash [v{:}int \mid v = 1 \vee 2]$ $\nvdash \{v{:}int \mid v = 1\}$	$\nvdash [v{:}int \mid v = 1]$ $\nvdash \{v{:}int \mid \bot\}$

These examples illustrate the previous observation that it is always possible to strengthen the refinement predicate used in an underapproximate type and weaken such a predicate in an overapproximate type. As a consequence, the bottom type $[v{:}int \mid \bot]$ is the universal supertype in our type hierarchy, as it places no restrictions on the values a term *must* produce. Thus, we sometimes abbreviate $[v{:}int \mid \bot]$ as `int`, since the information provided by both types is the same. Importantly, the coverage type for the error term (`err`) can *only* be qualified with $\bot$, since an erroneous computation is unconstrained with the respect to the values it is obligated to produce.

Coverage types can also qualify inductive datatypes, like lists and trees. In particular, the complete generator for BSTs presented in the introduction can be successfully type-checked using the following result type:

$$[v{:}int\ tree \mid bst(v) \wedge \forall u, mem(v, u) \implies \texttt{lo} < u < \texttt{hi}]$$

[1] The formalization of the typing rules and their soundness is mechanized in Coq and provided in the supplemental material.
[2] We use $\vdash$ and $\nvdash$ marks to indicate whether a term can or cannot be assigned the corresponding type, resp.

where $bst(\nu)$ and $mem(\nu, u)$ are *method predicates*, i.e., uninterpreted functions used to encode semantic properties of the datatype. In the type given above, the qualifier requires that `bst_gen`'s result is a BST and that every element u stored in the tree (as given by predicate $mem(\nu, u)$) is between `lo` and `hi`; the coverage type thus constrains the implementation to produce *all* trees that satisfy this qualifier predicate. In contrast, the incomplete version of the generator (i.e., the implementation that does not allow prematurely terminating tree generation with a `Leaf` node) could only be type-checked using the following (stronger) type:

$$[\nu{:}int\ tree \mid bst(\nu) \land \forall u, mem(\nu, u) \iff \texttt{lo} < u < \texttt{hi}]$$

This signature asserts that all trees produced by the generator are BSTs, that any element contained in the tree is within the interval bounded by `lo` and `hi`, *and* moreover, any element in that interval *must* be included in the tree. The subtle difference between the two implementations, reflected in the different implication constraints expressed in their respective refinements, precisely captures how their coverage properties differ.

Control Flow. Just as underapproximate coverage types invert the standard overapproximate subtyping relationship, they also invert the standard relationship between a control flow construct and its subexpressions. To see how, consider the simple program shown on the right that defines an even number generator in terms of an integer number generator. When the random number generator yields an odd number, the program faults; otherwise it simply returns the generated number. Consider the following type judgment that arises when type checking this program:

```
let even_gen () =
  let (n: int) = int_gen () in
  let (b: bool) = n mod 2 == 0 in
  if b then n else err
```

$$\mathsf{n}{:}[\nu{:}int \mid \top_{\mathsf{int}}], \mathsf{b}{:}[\nu{:}bool \mid \nu \iff n \ \mathsf{mod}\ 2 = 0] \vdash \texttt{if b then n else err} : [\nu{:}int \mid \nu \ \mathsf{mod}\ 2 = 0] \tag{1}$$

Intuitively, this judgment asserts that the `if` expression covers all even numbers (i.e., has the type $[\nu{:}int \mid \nu \ \mathsf{mod}\ 2 = 0]$) assuming that the *local variable* `n` can be instantiated with an arbitrary number, and that the variable `b` is true precisely when n is even. Notice how the typing context encodes the potential control-flow path that *must* reach the non-faulting branch of the conditional expression.

Notably, enforcing the requirement that the conditional be able to return all even numbers does not require each of its branches to be a subtype of the expected type, in contrast to standard type systems. Our type system must instead establish that, *in total*, the values produced by each of the branches cover the even numbers. Indeed, because the `false` branch of the conditional faults, it is only typeable at the universal supertype, i.e., $[\nu{:}int \mid \bot]$. Thus, if the standard subtyping relationship between this conditional and its branches held, it could only be typed at $[\nu{:}int \mid \bot]$! This is not the case, however, in our setting, as the `true` branch contributes all the desired outputs. Formally, this property is checked by the following assumption of the coverage typing rule for conditionals:

$$\mathsf{n}{:}[\nu{:}int \mid \top_{\mathsf{int}}], \mathsf{b}{:}[\nu{:}bool \mid \nu \iff \mathsf{n}\ \mathsf{mod}\ 2 = 0] \vdash [\nu{:}int \mid (\mathsf{b} \land \nu = \mathsf{n}) \lor (\neg\mathsf{b} \land \bot)] <: [\nu{:}int \mid \nu \ \mathsf{mod}\ 2 == 0]$$

The $\mathsf{b} \land \nu = n$ and $\neg\mathsf{b} \land \bot$ subformulas correspond to the types of the `true` and `false` branches[3], respectively. Taking the disjunction of these two formulas describe the set of values produced by either branch; this subtyping relationship guarantees this type is at least as large as the type expected by the entire conditional.

To check that this subtyping relationship holds, our type checker generates the following formula:

$$\forall \nu, (\nu \ \mathsf{mod}\ 2 = 0) \implies (\exists \mathsf{n}, \top \land \exists \mathsf{b}, \mathsf{b} \iff \mathsf{n}\ \mathsf{mod}\ 2 = 0 \land (\mathsf{b} \land \nu = \mathsf{n}) \lor (\neg\mathsf{b} \land \bot)) \tag{2}$$

[3] As is standard in dependent type systems, the types of both branches have been refined to reflect the path conditions under which they will be executed.

This formula aligns with the intuitive meaning of (1): in our type system, coverage types of variables in the typing context tell us what values they must (at least) produce. When checking whether a particular subtyping or typing relationship holds, we are free to choose *any* instantiation of the variables that entails the desired property. Accordingly, in (2), the variables n and b are *existentially* quantified to indicate there exists an execution path that instantiates these local variables in a way that produces the output v, instead of being universally quantified as they would be in a standard refinement type system.

Function types. To type functions, most refinement type systems add a restricted form of the *dependent function types* found in full-spectrum dependent type systems. Such types allow the qualifiers in the result type of a function to refer to its parameters, allowing the expression of rich safety conditions governing the arguments that may be supplied to the function. To see how this capability might be useful in our setting, consider the test generator `bst_gen` from the introduction. The complete version of this function produces all BSTs whose elements fall between the range specified by its two parameters, `lo` and `hi`. For the bounds `0` and `3`, the application `bst_gen 0 3` can be typed as: $[v{:}int \mid bst(v) \land \forall u, mem(v,u) \implies 0 < u < 3]$. Using the standard typing rule for functions, the only way to encode this relationship in the type of `bst_gen` is:

$$[v{:}int \mid v = 0] \to [v{:}int \mid v = 3] \to [v{:}int\ tree \mid bst(v) \land \forall u, mem(v,u) \implies 0 < u < 3]$$

Of course, this specification fails to account for the behaviors of `bst_gen` when supplied with different bounds: for example, the application `bst_gen 2 7` will fail to typecheck against this type.

Since the desired coverage proprty of `bst_gen` fundamentally depends on the kinds of inputs given to it, our type system includes dependent products of the form:

$$\texttt{lo}{:}\{v{:}int \mid \top_{\texttt{int}}\} \to \texttt{hi}{:}\{v{:}int \mid \texttt{lo} \leq v\} \to [v{:}int\ tree \mid bst(v) \land \forall u, mem(v,u) \implies \texttt{lo} < u < \texttt{hi}]$$

We use the notation $\{\ldots\}$ to emphasize that the argument types of a dependent arrow have a similar purpose and interpretation as in standard refinement type systems. Thus, the above type can be read as "if the inputs `lo` and `hi` are *any* number such that `lo` ≤ `hi`, then the output *must* cover all possible BSTs whose elements are between `lo` and `hi`". Using this type for `bst_gen` allows our system to seamlessly type-check both (`bst_gen 0 3`) and (`bst_gen 2 7`). Our typing algorithm will furthermore flag the call (`bst_gen 3 1`) as being ill-typed, since the function's type dictates that the generator's second argument (`1`) *may* only be greater than or equal to its first (`3`).

```
let bst_gen_low_bound (low: int) =
    let (high: int) = int_gen () in
    bst_gen low high
```

Fig. 2. This function generates a BST with a supplied lower bound, `low`.

Function Application. Since the type of a function parameter is interpreted as a normal (overapproximate, "may") refinement type, while arguments in an application may be typed using (underapproximate, "must") coverage types, we need to be able to bridge the gap between may and must types when typing function applications. Intuitively, our type system does so by ensuring that the set of values in the coverage type of the argument has a nonempty overlap with the set of possible values expected by the function. We establish this connection by using the fact that the typing context captures the control flow paths that may and must exist when the function is called. To illustrate this intuition concretely, consider the function `bst_gen_low_bound` shown in Figure 2. This function generates all non-empty BSTs whose elements are numbers with the lower bound given by its parameter. The judgment we need to check is of the form:

$$\texttt{low}{:}\{v{:}int \mid \top_{\texttt{int}}\}, \texttt{high}{:}[v{:}int \mid \top_{\texttt{int}}] \vdash \texttt{bst\_gen low high} : \ldots$$

Note that the type for `low` is a normal refinement type that specifies a safety condition for function `bst_gen_low_bound`, namely that `low` *may* be any number. In contrast, the type for `high` is a coverage type, representing the result of `int_gen()` that indicates that it *must* (i.e., guaranteed to) be any possible integer. However, the signature for `bst_gen` demands that parameter `hi` only be supplied values greater than its first argument (`lo`); we incorporate this requirement by *strengthening* `high`'s type (via a subsumption rule) to reflect this additional constraint when typing the body of the let expression in which `high` is bound. This strengthening, which is tantamount to a more refined underapproximation, allows us to typecheck the application (`bst_gen low high`) in the following context:

$$\texttt{bst\_gen} : \texttt{lo}{:}\{v{:}int \mid \top_{\texttt{int}}\} \rightarrow \texttt{hi}{:}\{v{:}int \mid \texttt{lo} \leq v\} \rightarrow [v{:}int\ tree \mid ...], \texttt{low}{:}\{v{:}int \mid \top_{\texttt{int}}\}, \texttt{high}{:}[v{:}int \mid \texttt{low} \leq v]$$

The coverage type associated with `high` guarantees that `int_gen()` is guaranteed to produce values greater than `low` (along with possibly other values). To ensure that the result type of the call reflects the underapproximate (coverage) dependences that exist between `low` and `high`, we introduce existential quantifiers in the type's qualifier:

$$[v{:}int\ tree \mid bst(v)\ \land\ \exists \texttt{high}, \texttt{low} \leq \texttt{high} \land \forall u, mem(v, u) \implies \texttt{low} < u < \texttt{high}]$$

This type properly captures the behavior of the generator: it is guaranteed to generate all BSTs characterized by a lower bound given `low` such that `low` ≤ `hi` and in which every element in the tree is contained within these bounds.

Summary. Coverage types invert many of the expected relationships that are found in a normal refinement type system. Here, qualifiers provide an *underapproximation* of the values that an expression may evaluate to, in contrast to the typically provided *overapproximation*. This, in turn, causes the subtyping relation to invert the standard relationship entailed by logical implication between type qualifiers. Our coverage analysis also considers the *disjunction* of the coverage guarantees provided by the branches of control-flow constructs, instead of their *conjunction*. Finally, when applying a function with a dependent arrow type to a coverage type, we check semantic inclusion between the overapproximate and underapproximate constraints provided by the two types, and manifest the paths that witness the elements guaranteed to be produced by the coverage type through existentially-quantified variables in the application's result type.

3 LANGUAGE

In order to formalize our typed-based verification approach of input test generators, we introduce a language for test generators, $\boldsymbol{\lambda}^{TG}$. The language, whose syntax is summarized in Figure 3, is a call-by-value lambda-calculus with pattern-matching, inductive datatypes, and well-founded (i.e., terminating) recursive functions whose argument must be structurally decreasing in all recursive calls made in the function's body. The syntax of $\boldsymbol{\lambda}^{TG}$ is expressed in monadic normal-form (MNF) [13], a variant of A-Normal Form (ANF) [10] that allows nested let-bindings. The language additionally allows faulty programs to be expressed using the error term `err`. As discussed in Section 2, this term is important in our investigation because coverage types capture an expression's reachability properties, and we need to ensure the guarantees offered by such types are robust even in the presence of stuck computations induced by statements like `err`. The language is also equipped with primitive operators to generates natural numbers, integers, etc. (`nat_gen ()`, `int_gen()`, *etc.*) that can be used to express various kinds of non-deterministic behavior relevant to test input generation. As an example, the ⊕ choice operator used in Figure 1 can be defined as:

$$e_1 \oplus e_2 \doteq \texttt{let}\ n = \texttt{nat\_gen}\ ()\ \texttt{mod}\ 2\ \texttt{in match}\ n\ \texttt{with}\ 0 \longrightarrow e_1 \mid \_ \longrightarrow e_2$$

Note that the primitive generators of $\boldsymbol{\lambda}^{TG}$ are completely agnostic to the specific sampling strategy they employ, as long as they ensure every value in their range has a nonzero likelihood of being generated. Indeed, $\boldsymbol{\lambda}^{TG}$ does not include any operators to bias the frequency at which values are produced, e.g., QuickCheck's `frequency`. While we could include such an operator, it would not

$$
\begin{array}{rrl}
\textbf{Variables} & & x, f, u, \ldots \\
\textbf{Data constructors} & d ::= & () \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{O} \mid \mathsf{S} \mid \mathsf{Cons} \mid \mathsf{Nil} \mid \mathsf{Leaf} \mid \mathsf{Node} \\
\textbf{Constants} & c ::= & \mathbb{B} \mid \mathbb{N} \mid \mathbb{Z} \mid \ \ldots \ \mid d\ \overline{c} \\
\textbf{Operators} & op ::= & d \mid + \mid == \mid < \mid \mathsf{mod} \mid \mathsf{nat\_gen} \mid \mathsf{int\_gen} \mid \ldots \\
\textbf{Values} & v ::= & c \mid op \mid x \mid \lambda x{:}t.e \mid \mathsf{fix} f{:}t.\lambda x{:}t.e \\
\textbf{Terms} & e ::= & v \mid \mathsf{err} \mid \mathtt{let}\ x = e\ \mathtt{in}\ e \mid \mathtt{let}\ x = op\ \overline{v}\ \mathtt{in}\ e \mid \mathtt{let}\ x = v\ v\ \mathtt{in}\ e \\
 & & \mid \mathtt{match}\ v\ \mathtt{with}\ \overline{d\ \overline{y} \longrightarrow e} \\
\textbf{Base Types} & b ::= & \mathit{unit} \mid \mathit{bool} \mid \mathit{nat} \mid \mathit{int} \mid b\ \mathit{list} \mid b\ \mathit{tree} \mid \ \ldots \\
\textbf{Basic Types} & t ::= & b \mid t \rightarrow t \\
\textbf{Method Predicates} & mp ::= & \mathit{emp} \mid \mathit{hd} \mid \mathit{mem} \mid \ldots \\
\textbf{Literals} & l ::= & c \mid x \\
\textbf{Propositions} & \phi ::= & l \mid \bot \mid \top_b \mid op(\bar{l}) \mid mp(\overline{x}) \mid \neg\phi \mid \phi \land \phi \mid \phi \lor \phi \mid \phi \Longrightarrow \phi \mid \forall u{:}b.\ \phi \mid \exists u{:}b.\ \phi \\
\textbf{Refinement Types} & \tau ::= & [v{:}b \mid \phi] \mid \{v{:}b \mid \phi\} \mid x{:}\tau \rightarrow \tau \\
\textbf{Type Contexts} & \Gamma ::= & \emptyset \mid \Gamma, x{:}\tau
\end{array}
$$

Fig. 3. $\boldsymbol{\lambda}^{TG}$ syntax.

change anything fundamentally about our type system or its guarantees. The operational semantics of $\boldsymbol{\lambda}^{TG}$ is otherwise standard and can be found in the supplementary material.

3.1 Types

Like other refinement type systems [17, 34], $\boldsymbol{\lambda}^{TG}$ supports three classes of types: *base types*, *basic types*, and *refinement types*. Base types (b) include primitive types such as `unit`, `bool`, `nat`, etc., and inductive datatypes (e.g., `int list`, `bool tree`, `int list list`, etc.). Basic types (t) extend base types with function types. Refinement types (τ) qualify base types with both underapproximate and overapproximate propositions, expressed as predicates defined in first-order logic (FOL). Function parameters can also be qualified with overapproximate refinements that specify when it is safe to apply this function. In contrast, the return type of a function can only be qualified using an underapproximate refinement, reflecting the coverage property of the function's result and thus characterizing the values the function is guaranteed to produce. The erasure of a type τ, $\lfloor \tau \rfloor$, is the type that results from erasing all qualifiers in τ.

Refinements and Logic. To express rich shape properties over inductive datatypes, we allow propositions to reference method predicates, as it is straightforward to generate verification conditions using these uninterpreted functions that can be handled by an off-the-shelf theorem prover like Z3 [4]. As we describe in Section 5, our typechecking algorithm imposes additional constraints on the form propositions can take, in order to ensure that its validity is decidable. In particular, we ensure that Z3 queries generated by our typechecker to check refinement validity are always over effectively propositional (EPR) sentences (i.e., prenex-quantified formulae of the form $\exists^*\forall^*\varphi$ where φ is a quantifier-free sentence).

4 TYPE SYSTEM

Despite superficial similarities to other contemporary type systems (e.g., [17, 34]), the typing rules of $\boldsymbol{\lambda}^{TG}$ differ in significant ways from those of its peers, due to the fundamental semantic distinction

Well-Formedness $\boxed{\Gamma \vdash^{\textbf{WF}} \tau}$

$$\frac{\Gamma \equiv \overline{x_i{:}\{v{:}b_{x_i} \mid \phi_{x_i}\}, y_j{:}[v{:}b_{y_j} \mid \phi_{y_j}], z{:}(a{:}\tau_a \to \tau_b)} \\ (\overline{\forall x_i{:}b_{x_i}, \exists y_j{:}b_{y_i}}, \forall v{:}b, \phi) \text{ is a Boolean predicate} \quad \forall j, \texttt{err} \not\in [\![[v{:}b_{y_j} \mid \phi_{y_j}]]\!]_\Gamma}{\Gamma \vdash^{\textbf{WF}} [v{:}b \mid \phi]} \textsc{WfBase}$$

$$\frac{\Gamma, x{:}\{v{:}b \mid \phi\} \vdash^{\textbf{WF}} \tau}{\Gamma \vdash^{\textbf{WF}} x{:}\{v{:}b \mid \phi\} \to \tau} \textsc{WfArg} \qquad \frac{\Gamma \vdash^{\textbf{WF}} (a{:}\tau_a \to \tau_b) \quad \Gamma \vdash^{\textbf{WF}} \tau}{\Gamma \vdash^{\textbf{WF}} (a{:}\tau_a \to \tau_b) \to \tau} \textsc{WfRes}$$

Subtyping $\boxed{\Gamma \vdash \tau_1 <: \tau_2}$

$$\frac{[\![[v{:}b \mid \phi_1]]\!]_\Gamma \subseteq [\![[v{:}b \mid \phi_2]]\!]_\Gamma}{\Gamma \vdash [v{:}b \mid \phi_1] <: [v{:}b \mid \phi_2]} \textsc{SubUBase} \qquad \frac{[\![\{v{:}b \mid \phi_1\}]\!]_\Gamma \subseteq [\![\{v{:}b \mid \phi_2\}]\!]_\Gamma}{\Gamma \vdash \{v{:}b \mid \phi_1\} <: \{v{:}b \mid \phi_2\}} \textsc{SubOBase}$$

$$\frac{\Gamma \vdash \tau_{21} <: \tau_{11} \quad \Gamma, x{:}\tau_{21} \vdash \tau_{12} <: \tau_{22}}{\Gamma \vdash x{:}\tau_{11} \to \tau_{12} <: x{:}\tau_{21} \to \tau_{22}} \textsc{SubArr}$$

Disjunction $\boxed{\Gamma \vdash \tau_1 \vee \tau_2 = \tau_3}$

$$\frac{[\![\tau_1]\!]_\Gamma \cap [\![\tau_2]\!]_\Gamma = [\![\tau_3]\!]_\Gamma}{\Gamma \vdash \tau_1 \vee \tau_2 = \tau_3} \textsc{Disjunction}$$

Fig. 4. Auxillary typing relations

that arises when viewing types as an underapproximation and not overapproximation of program behavior.

Our type system depends on three auxiliary relations shown in Figure 4. The first group defines well-formedness conditions on a type under a particular type context, i.e., a sequence of variable-type bindings consisting of overapproximate refinement types, underapproximate coverage types, and arrow (function) types. A type τ that is well-formed under a type context Γ needs to meet three criteria: (1) the qualifier in τ need to be closed in the current typing context, and the denotation[4] of all the coverage types ($[v{:}b_{y_j} \mid \phi_{y_j}]$) found in Γ should not include `err` (WfBase); (2) overapproximate types may only appear in the domain of a function type (WfArg); and, (3) underapproximate coverage types may only appear in the range of a function type (WfRes). To understand the motivation for the first criterion, observe that a type context in our setting provides a witness to feasible execution paths in the form of bindings to local variables. Accordingly, *no* type is well formed under the type context $x{:}[v{:}nat \mid \bot]$ or under $x{:}\{v{:}nat \mid v > 0\}, y{:}[v{:}nat \mid x = 0 \wedge v = 2]$, as neither context corresponds to a valid manifest execution path. On the other hand, a well-formed type is allowed to include an error term in its denotation, e.g., type $[v{:}nat \mid \bot]$ is well-formed under type context $x{:}\{v{:}nat \mid v > 0\}$ as it always corresponds to a valid underapproximation.

Our second set of judgments defines a largely standard subtyping relation based on the underlying denotation of the types being related. Note also that over- and under-approximate types are incomparable– our typing rules tightly control when one can be treated as another.

The disjunction rule (Disjunction), which was informally introduced in Section 2, merges the coverage types found along distinct control paths. Intuitively, the type $[v{:}nat \mid v = 1 \vee v = 2]$ is the disjunction of the types $[v{:}nat \mid v = 1]$ and $[v{:}nat \mid v = 2]$. Notice that only an inhabitant of both $[v{:}nat \mid v = 1]$ and $[v{:}nat \mid v = 2]$ should be included in their disjunction: e.g., the term $1 \oplus 2$

[4] The definition of a type's denotation is given in subsection 4.1.

is one such inhabitant. Thus, we formally define this relation as the *intersection* of the denotations of two types.

Typing

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{\Gamma \vdash^{\mathbf{WF}} \mathsf{Ty}(c)}{\Gamma \vdash c : \mathsf{Ty}(c)}\ \textsc{TConst} \qquad \frac{\Gamma, x{:}\tau_x \vdash e : \tau \quad \Gamma \vdash^{\mathbf{WF}} x{:}\tau_x {\to} \tau}{\Gamma \vdash \lambda x{:}\lfloor \tau_x \rfloor . e : (x{:}\tau_x {\to} \tau)}\ \textsc{TFun} \qquad \frac{\Gamma \vdash^{\mathbf{WF}} [v{:}b \mid \bot]}{\Gamma \vdash \mathsf{err} : [v{:}b \mid \bot]}\ \textsc{TErr}$$

$$\frac{\Gamma \vdash^{\mathbf{WF}} [v{:}b \mid v = x]}{\Gamma \vdash x : [v{:}b \mid v = x]}\ \textsc{TVarBase} \qquad \frac{\Gamma(x) = (a{:}\tau_a {\to} \tau_b) \quad \Gamma \vdash^{\mathbf{WF}} a{:}\tau_a {\to} \tau_b}{\Gamma \vdash x : (a{:}\tau_a {\to} \tau_b)}\ \textsc{TVarFun}$$

$$\frac{\begin{array}{c} \Gamma \vdash v_1 : a{:}\{v{:}b \mid \phi\} \to \tau_x \\ \Gamma \vdash v_2 : [v{:}b \mid \phi] \\ \Gamma, x{:}\tau_x[a \mapsto v_2] \vdash e : \tau \\ \Gamma \vdash^{\mathbf{WF}} \tau \end{array}}{\Gamma \vdash \mathtt{let}\ x = v_1\ v_2\ \mathtt{in}\ e : \tau}\ \textsc{TApp} \qquad \frac{\begin{array}{c} \emptyset \vdash \tau <: \tau' \quad \emptyset \vdash e : \tau \\ \Gamma \vdash^{\mathbf{WF}} \tau' \end{array}}{\Gamma \vdash e : \tau'}\ \textsc{TSub} \qquad \frac{\begin{array}{c} \Gamma \vdash \tau' <: \tau \quad \Gamma \vdash \tau <: \tau' \\ \Gamma \vdash e : \tau \quad \Gamma \vdash^{\mathbf{WF}} \tau' \end{array}}{\Gamma \vdash e : \tau'}\ \textsc{TEq}$$

$$\frac{\begin{array}{c} \Gamma \vdash v : \tau_v \quad \Gamma \vdash^{\mathbf{WF}} \tau \quad \Gamma, \overline{y{:}\tau_y} \vdash d_i(\overline{y}) : \tau_v \\ \Gamma, \overline{y{:}\tau_y} \vdash e_i : \tau \end{array}}{\Gamma \vdash (\mathtt{match}\ v\ \mathtt{with}\ \overline{d_i\ \overline{y} \longrightarrow e_i}) : \tau}\ \textsc{TMatch} \qquad \frac{\begin{array}{c} \Gamma \vdash e : \tau_1 \quad \Gamma \vdash e : \tau_2 \\ \Gamma \vdash \tau_1 \vee \tau_2 = \tau \quad \Gamma \vdash^{\mathbf{WF}} \tau \end{array}}{\Gamma \vdash e : \tau}\ \textsc{TMerge}$$

$$\frac{\Gamma \vdash \lambda x{:}b.\lambda f{:}(b {\to} \lfloor \tau \rfloor).e : (x{:}\{v{:}b \mid \phi\} {\to} f{:}(x{:}\{v{:}b \mid v{<}x \wedge \phi\} \to \tau) {\to} \tau) \quad \Gamma \vdash^{\mathbf{WF}} x{:}\{v{:}b \mid \phi\} \to \tau}{\Gamma \vdash \mathsf{fix} f{:}(b {\to} \lfloor \tau \rfloor).\lambda x{:}b.e : (x{:}\{v{:}b \mid \phi\} \to \tau)}\ \textsc{TFix}$$

Fig. 5. Selected typing rules

The salient rules of our type system are defined in Figure 5[5]. The rules collectively maintain the invariant that terms can only be assigned a well-formed type. The rule for constants (TConst) is straightforward. It relies on an auxiliary function, Ty, to assign types to the primitives of $\boldsymbol{\lambda}^{TG}$. Figure 6 presents some examples of the typings provided by Ty. We use method predicates in the types of constructors: the types for list constructors, for example, use *emp*, *hd* and *tl*, to precisely capture that `[]` constructs an empty list, and that (`Cons` x y) builds a list containing x as its head and y as its tail.[6]

The typing rules for function abstraction (TFun) and error (TErr) are similarly straightforward. The type of the function's argument τ should be consistent with the type of the argument's erasure ($\lfloor \tau_x \rfloor$) specified by the λ-abstraction. The error term can have arbitrary bottom coverage base type. The variable rule (TVarBase) establishes that the variable x in the type context with a base type can also be typed with the tautological qualifier $v = x$ (the well-formedness guarantee ensures that x is not free here). This judgment allows us to, for example, type the function $\lambda x : nat.x$ with the type $x{:}\{v{:}nat \mid \top_{\mathsf{nat}}\} \to [v{:}nat \mid v = x]$, indicating that the return value is guaranteed to be exactly equal to the input x. Observe that the type of x under the type context $x{:}\{v{:}nat \mid \top_{\mathsf{nat}}\}$ (generated by the function rule TFun) is *not* $[v{:}nat \mid \top_{\mathsf{nat}}]$. We cannot simply duplicate the qualifier for x from the type context here, as this is only sound when types characterize an overapproximation of program behavior. As an example, $\{v{:}nat \mid \top_{\mathsf{nat}}\}$ is a subtype of $\{v{:}nat \mid v = x\}$ under the type

[5] The full set of rules can be found in the supplementary material.

[6] The auxiliary function Ty also provides a type for operators, thus the rule for operators is the same as TConst.

Constants	$\mathsf{Ty}(\mathsf{true}) = [v{:}bool \mid v]$ $\mathsf{Ty}(8) = [v{:}nat \mid v = 8]$...
Data Constructors	$\mathsf{Ty}([]) = [v{:}b\ list \mid emp(v)]$ $\mathsf{Ty}(\mathsf{Cons}) = x{:}\{v{:}b \mid \top_{\mathsf{b}}\} \to y{:}\{v{:}b\ list \mid \top_{\mathsf{t\ list}}\} \to [v{:}b\ list \mid hd(v,x) \land tl(v,y)]$...
Operators	$\mathsf{Ty}(\mathsf{nat\_gen}) = \{v{:}unit \mid \top_{\mathsf{unit}}\} \to [v{:}nat \mid \top_{\mathsf{nat}}]$ $\mathsf{Ty}(+) = x{:}\{v{:}nat \mid \top_{\mathsf{unit}}\} \to y{:}\{v{:}nat \mid \top_{\mathsf{nat}}\} \to [v{:}nat \mid v = x + y]$...

Fig. 6. Example typings for $\boldsymbol{\lambda}^{TG}$ primitives.

context $x{:}\{v{:}nat \mid \top_{\mathsf{nat}}\}$). In contrast, in our underapproximate coverage type system, $[v{:}nat \mid \top_{\mathsf{nat}}]$ is not a subtype of $[v{:}nat \mid v = x]$ under the type context $x{:}\{v{:}nat \mid \top_{\mathsf{nat}}\}$.

The typing rule for application TApp requires both its underapproximate argument type and the overapproximate parameter type to have the same qualifier, and furthermore requires that the type of the body (τ) is well-formed under the original type context Γ, enforcing x (the result of the application) to not appear free in τ. When argument and parameter qualifiers are not identical, a subsumption rule is typically used to bring the two types into alignment. Recall the following example from Section 2, suitably modified to conform to $\boldsymbol{\lambda}^{TG}$'s syntax:

$$\mathsf{bst\_gen} : \mathsf{lo}{:}\{v{:}int \mid \top_{\mathsf{int}}\} \to \mathsf{hi}{:}\{v{:}int \mid \mathsf{lo} \le v\} \to [v{:}int\ tree \mid ...], \mathsf{low} : \{v{:}int \mid \top_{\mathsf{int}}\} \vdash$$

```
      let (g: unit -> int) = int_gen in let (x: unit) = () in
      let (high: int) = g x in let (y: int tree) = bst_gen low high in y
```

Here, the type of high, $[v{:}int \mid \top_{\mathsf{int}}]$ is stronger than the type expected for the second parameter of bst_gen, $[v{:}int \mid \mathsf{lo} \le v]$. The subsumption rule (TSub), that would normally allow us to strengthen the type of high to align with the required parameter type, is applicable to only closed terms, which high is not. For the same reason, we cannot use TSub to strengthen the type of high when it is bound to g x. Thankfully, we can strengthen g when it is bound to int_gen: According to Figure 6, the operator int_gen has type $\{v{:}unit \mid \top_{\mathsf{unit}}\} \to [v{:}int \mid \top_{\mathsf{int}}]$ and is also closed, and can thus be strengthened via TSub, allowing us to type the call to bst_gen under the following, stronger type context:

$$\begin{array}{l}\mathsf{bst\_gen} : \mathsf{lo}{:}\{v{:}int \mid \top_{\mathsf{int}}\} \to \mathsf{hi}{:}\{v{:}int \mid \mathsf{lo} \le v\} \to [v{:}int\ tree \mid ...], \mathsf{low} : \{v{:}int \mid \top_{\mathsf{int}}\},\\ \mathsf{g} : \{v{:}unit \mid \top_{\mathsf{unit}}\} \to [v{:}int \mid \mathsf{low} \le v], \mathsf{x} : [v{:}unit \mid \top_{\mathsf{unit}}], \mathsf{high} : [v{:}int \mid \mathsf{low} \le v] \vdash\end{array}$$

```
     let (y: int tree) = bst_gen low high in y
```

The subsumption rule allows us to use int_gen in a context that requires *fewer* guarantees than int_gen () actually provides, namely those values of high required by the signature of bst_gen. Intuitively, since our notion of coverage types records feasible executions in the type context in the form of existentials that serve as witnesses to an underapproximation, the strengthening provided by the subsumption rule establishes an invariant that all bindings introduced into a type context only characterize valid behaviors in a program execution. When coupled with TMerge, this allows us to *split* a typing derivation into multiple plausible strengthenings when a variable is introduced into the typing context and then *combine* the resulting types to reason about multiple feasible paths.

Now, applying TApp to type the application, and TVarBase to type the body of the let gives us:

$$\begin{array}{l}\mathsf{bst\_gen} : \mathsf{lo}{:}\{v{:}int \mid \top_{\mathsf{int}}\} \to \mathsf{hi}{:}\{v{:}int \mid \mathsf{lo} \le v\} \to [v{:}int\ tree \mid ...], \mathsf{low} : \{v{:}int \mid \top_{\mathsf{int}}\}, \mathsf{high} : [v{:}int \mid \mathsf{low} \le v],\\ \qquad \mathsf{y} : [v{:}int\ tree \mid bst(v) \land \forall u, mem(v,u) \implies \mathsf{lo} < u < \mathsf{hi}][\mathsf{lo} \mapsto \mathsf{low}][\mathsf{hi} \mapsto \mathsf{high}] \vdash\\ \quad \mathsf{y} : [v{:}int\ tree \mid v = \mathsf{y}]\end{array}$$

Observe that TVarBase types the body as: $[v{:}int\ tree \mid v = \mathsf{y}]$, which is not closed. To construct a well-formed term, we need a formula equivalent to this type that accounts for the type of y in the current type context. The TEq rule allows us to interchange formulae that are equivalent under a given type context to ensure the well-formedness of the types constructed. Unlike TSub, it simply changes the form of a type's qualifiers, *without altering* the scope of feasible behaviors under the current context. In this example, such a closed equivalent type, given the binding for y in the type context under which the expression is being type-checked, would be:

```
let (y: int tree) = bst_gen low high in y :
```

$$[\nu{:}\mathit{int\ tree} \mid \exists \mathtt{y}, (\mathit{bst}(\mathtt{y}) \land \forall u, \mathit{mem}(\mathtt{y}, u) \implies \mathtt{low} < u < \mathtt{high}) \land \nu = \mathtt{y}]$$

With these pieces in hand, we can see that the typing rule for `match` is a straightforward adaptation of the components we have already seen, where the type of matched variable v is assumed to have been strengthened by the rule TSub to fit the type required to take the i^{th} branch $\Gamma, \overline{y{:}\tau_y} \vdash d_i(\overline{y}) : \tau_v$. We can also safely assume the type of the branch τ_i is closed under original type context Γ, relying on TEq to meet this requirement. While TMatch only allows for a single branch to be typechecked, applying TMerge allows us to reason about the coverage provided by multiple branches, which have all been typed according to this rule.

The typing rule for recursive functions is similarly standard,[7] with the caveat that it can only type terminating functions; since types in our language serve as witnesses to feasible executions, the result type of any recursive procedure must characterize the set of values the procedure can plausibly return. Thus, the TFix rule forces its first argument to always decrease according to some well-founded relation $\prec$. To see why we impose this restriction, consider the function loop:

```
let rec loop (n: nat) = loop n
```

Without our termination check, this function can be assigned the type $\{\nu{:}\mathit{nat} \mid \top_{\mathtt{nat}}\} \to [\nu{:}\mathit{nat} \mid \nu = 3]$, despite the fact that this function never returns 3– or any value at all! The body of this expression can be type-checked under the following type context (via TFix and TFun):

$$\mathtt{n}{:}\{\nu{:}\mathit{nat} \mid \top_{\mathtt{nat}}\}, \mathtt{loop}{:}(\mathtt{n}{:}\{\nu{:}\mathit{nat} \mid \top_{\mathtt{nat}}\} \to [\nu{:}\mathit{nat} \mid \nu = 3]) \vdash \mathtt{loop\ n} \;:\; [\nu{:}\mathit{nat} \mid \nu = 3]$$

This judgment reflects an infinitely looping execution, however. Indeed, the same reasoning allows us to type this function with any result type. Constraining `loop`'s argument type to be decreasing according to $\prec$ yields the following typing obligation:

$$\mathtt{n}{:}\{\nu{:}\mathit{nat} \mid \top_{\mathtt{nat}}\}, \mathtt{loop}{:}(\mathtt{n}{:}\{\nu{:}\mathit{nat} \mid \nu \prec \mathtt{n}\} \to [\nu{:}\mathit{nat} \mid \nu = 3]) \vdash \mathtt{n} \;:\; [\nu{:}\mathit{nat} \mid \nu = \mathtt{n}]$$

where the qualifiers $\nu \prec \mathtt{n}$ and $\nu = \mathtt{n}$ conflict, raising a type error, and preventing `loop` from being recursively applied to n.

4.1 Soundness

Type Denotations. Assuming a standard typing judgement for basic types, $\emptyset \vdash_t e : \mathtt{t}$,[8] a type denotation for a type τ, $[\![\tau]\!]$, is a set of closed expressions:

$$\begin{aligned}
[\![\{\nu{:}b \mid \phi\}]\!] &\doteq \{v \mid \emptyset \vdash_t v : b \land \phi[\nu \mapsto v]\} \\
[\![[\nu{:}b \mid \phi]]\!] &\doteq \{e \mid \emptyset \vdash_t e : b \land \forall v{:}b, \phi[\nu \mapsto v] \implies e \hookrightarrow^* v\} \\
[\![x{:}\tau_x \to \tau]\!] &\doteq \{f \mid \emptyset \vdash_t f : \lfloor \tau_x \to \tau \rfloor \land \forall v_x \in [\![\tau_x]\!] \implies f\ v_x \in [\![\tau[x \mapsto v_x]]\!]\}
\end{aligned}$$

In the case of an overapproximate refinement type, $\{\nu{:}b \mid \phi\}$, the denotation is simply the set of all values of type b whose elements satisfy the type's refinement predicate (ϕ), when substituted for all free occurrences of ν in ϕ.[9] Dually, the denotation of an underapproximate coverage type is the set of expressions that evaluate to v whenever $\phi[\nu \mapsto v]$ holds, where ϕ is the type's refinement predicate. Thus, every expression in such a denotation serves as a witness to a feasible, type-correct,

[7] As in TFun, the self-reference to f and the parameter of the lambda abstraction x in the recursive function body must have type annotations consistent with the basic type of the *fix* expression.

[8] The typing rules for basic types are provided in the supplemental material.

[9] The denotation of an overapproximate refinement type is more generally $\{e{:}b \mid \emptyset \vdash e : b \land \forall v{:}b, e \hookrightarrow^* v \implies \phi[x \mapsto v]\}$. However, because such types are only used for function parameters, and our language syntax only admits values as arguments, our denotation uses the simpler form.

execution. The denotation for a function type is defined in terms of the denotations of the function's argument and result in the usual way, ensuring that our type denotation is a logical predicate.

Type Denotation under Type Context. The denotation of a refinement types τ under a type context Γ (written $\llbracket\tau\rrbracket_\Gamma$) is[10,11]:

$$\begin{aligned}
\llbracket\tau\rrbracket_\emptyset &\doteq \llbracket\tau\rrbracket \\
\llbracket\tau\rrbracket_{x:\tau_x,\Gamma} &\doteq \{e \mid \forall v_x \in \llbracket\tau_x\rrbracket, \texttt{let } x = v_x \texttt{ in } e \in \llbracket\tau[x \mapsto v_x]\rrbracket_{\Gamma[x\mapsto v_x]}\} && \text{if } \tau \equiv \{v{:}b \mid \phi\} \\
\llbracket\tau\rrbracket_{x:\tau_x,\Gamma} &\doteq \{e \mid \exists \hat{e}_x \in \llbracket\tau_x\rrbracket, \forall e_x \in \llbracket\tau_x\rrbracket, \texttt{let } x = e_x \texttt{ in } e \in \bigcap_{\hat{e}_x \hookrightarrow^* v_x} \llbracket\tau[x \mapsto v_x]\rrbracket_{\Gamma[x\mapsto v_x]}\} && \text{otherwise}
\end{aligned}$$

The denotation of an overapproximate refinement type under a type context is mostly unsurprising, other than our presentation choice to use a let-binding, rather than substitution, to construct the expressions included in the denotations. For a coverage type, however, the definition precisely captures our notion of a reachability witness by explicitly constructing an execution path as a sequence of `let`-bindings that justifies the inhabitant of the target type τ. Using `let`-bindings forces expressions in the denotation to make consistent choices when evaluated. The existential introduced in the definition captures the notion of an underapproximation, while the use of set intersection allows us to reason about non-determinism introduced by primitive generators like `nat_gen()`.

Example 4.1. The term x+1 is included in the denotation of the type $[v{:}nat \mid v = x + 1 \lor v = x + x]$ under the type context $x : [v{:}nat \mid v = 1]$. This is justified by picking 1 for $\hat{e}_x$, which yields a set intersection that is equivalent to $\llbracket[v{:}nat \mid v = 2]\rrbracket$. Observe that *any* expression in $\llbracket[v{:}nat \mid v = 1]\rrbracket$, e.g. $0 \oplus 1$ and $1 \oplus 2$, yields an expression, $\texttt{let } x = 0 \oplus 1 \texttt{ in } x + 1$ or $\texttt{let } x = 1 \oplus 2 \texttt{ in } x + 1$, included in this intersection.

On the other hand, the term x is not a member of this denotation. To see why, let us pick `nat_gen()` for $\hat{e}_x$. This yields a set intersection that is equivalent to $\llbracket[v{:}nat \mid \top_{\mathsf{nat}}]\rrbracket$. While specific choices for e_x, e.g., `nat_gen()`, are included in this denotation, it does not work for all terms $e_x \in \llbracket[v{:}nat \mid v = 1]\rrbracket$. As one example, $0 \oplus 1 \oplus 2$ is an element of this set, but $\texttt{let } x = 0 \oplus 1 \oplus 2 \texttt{ in } x$ is clearly not a member of $\llbracket[v{:}nat \mid \top_{\mathsf{nat}}]\rrbracket$. Suppose instead that we picked a more restrictive expression for $\hat{e}_x$, like the literal 1 from the previous example. Here, it is easy to choose $e_x \in \llbracket[v{:}nat \mid v = 1]\rrbracket$ (e.g., the literal 1) such that $\texttt{let } x = e_x \texttt{ in } x \notin \llbracket[v{:}nat \mid v = 2]\rrbracket$.

Our main soundness result establishes the correctness of type-checking in the presence of coverage types with respect to a type's denotation[12]:

Theorem 4.2 (Soundness of typing rules). *For all type contexts Γ, terms e and coverage types τ, $\Gamma \vdash e : \tau \implies e \in \llbracket\tau\rrbracket_\Gamma$.*

It immediately follows that a closed input generator e with coverage type $[v{:}b \mid \phi]$ must produce every value satisfying ϕ, as desired.

5 TYPING ALGORITHM

The declarative typing rules are highly nondeterministic, relying on a combination of the TMerge and TSub rules to both explore and combine the executions needed to establish the desired coverage

[10] In the last case, since $\hat{e}_x$ may non-deterministically reduce to multiple values, we employ intersection (not union), similar to the Disjunction rule.

[11] In reasoning about the subset relation of the denotations of two types under a type context $\llbracket[v{:}b \mid \phi_1]\rrbracket_\Gamma \subseteq \llbracket[v{:}b \mid \phi_2]\rrbracket_\Gamma$ we require that the denotations be computed using the same Γ; details are provided in the supplemental material.

[12] A mechanized proof of this theorem in Coq is provided in the supplemental material.

properties. In addition, each of the auxillary typing relations depend on logical properties of the semantic interpretation of types. Any effective type checking algorithm based on these rules must address both of these issues. Our solution to the first problem is to implement a bidirectional type checker [7] whose type synthesis phase characterizes a set of feasible paths and whose type checking phase ensures those paths produce the desired results. Our solution to the second is to encode the logical properties into a *decidable* fragment of first order logic that can be effectively discharged by an SMT solver.

5.1 Bidirectional Typing Algorithm

Type Synthesis $\boxed{\Gamma \vdash e \Rightarrow \tau}$ **Type Check** $\boxed{\Gamma \vdash e \Leftarrow \tau}$

$$\frac{\begin{array}{c}\forall i, \mathsf{Ty}(d_i) = \overline{y{:}\{v{:}b_y \mid \theta_y\}}{\rightarrow}[v{:}b \mid \psi_i] \quad \Gamma'_i = \overline{y{:}[v{:}b_y \mid \theta_y]}, a{:}[v{:}b \mid v = v_a \land \psi_i] \\ \Gamma, \Gamma'_i \vdash e_i \Rightarrow \tau_i \quad \tau'_i = \mathsf{Ex}(\Gamma'_i, \tau_i) \quad \Gamma \vdash \mathsf{Disj}(\overline{\tau'_i}) <: \tau' \quad \Gamma \vdash^{\mathbf{WF}} \tau'\end{array}}{\Gamma \vdash \mathtt{match}\ v_a\ \mathtt{with}\ \overline{d_i\ \overline{y} \longrightarrow e_i} \Leftarrow \tau'} \textsc{ChkMatch}$$

$$\frac{\begin{array}{c}\Gamma \vdash v_1 \Rightarrow (a{:}\tau_a{\rightarrow}\tau_b){\rightarrow}\tau_x \\ \Gamma \vdash v_2 \Leftarrow a{:}\tau_a{\rightarrow}\tau_b \quad \Gamma' = x{:}\tau_x \\ \Gamma, \Gamma' \vdash e \Rightarrow \tau \quad \tau' = \mathsf{Ex}(\Gamma', \tau) \\ \Gamma \vdash^{\mathbf{WF}} \tau'\end{array}}{\Gamma \vdash \mathtt{let}\ x = v_1\ v_2\ \mathtt{in}\ e \Rightarrow \tau'} \textsc{SynAppFun} \qquad \frac{\begin{array}{c}\Gamma \vdash v_1 \Rightarrow a{:}\{v{:}b \mid \phi\} \rightarrow \tau_x \\ \Gamma' = a{:}[v{:}b \mid v = v_2 \land \phi], x{:}\tau_x \\ \Gamma, \Gamma' \vdash e \Rightarrow \tau \quad \tau' = \mathsf{Ex}(\Gamma', \tau) \\ \Gamma \vdash^{\mathbf{WF}} \tau'\end{array}}{\Gamma \vdash \mathtt{let}\ x = v_1\ v_2\ \mathtt{in}\ e \Rightarrow \tau'} \textsc{SynAppBase}$$

Fig. 7. Selected Bidirectional Typing Rules

As is standard in bidirectional type systems, our typing algorithm consists of a type synthesis judgement ($\Gamma \vdash e \Rightarrow \tau$) and a type checking judgment ($\Gamma \vdash e \Leftarrow \tau$). Figure 7 presents the key rules for both.

Typing `match`. As we saw in Section 4, applying the declarative typing rule for `match` expressions typically requires first using several other rules to get things into the right form: TMerge is required to analyze and combine the types of each branch, TSub is used to equip each branch with the right typing context, and TEq is used to remove any local or pattern variables from the type of a branch. Our bidirectional type system combines all of these into the single ChkMatch rule shown in Figure 7. At a high level, this rule synthesizes a type for all the branches and then ensures that, in combination, they cover the desired type.

Similarly to other refinement type systems, when synthesizing the type for the branch for constructor d_i , we use a ghost variable $a{:}[v{:}b \mid v = v_a \land \psi_i]$ to ensure that the types of any pattern variables $\overline{y}$ are consistent with the parameters of d_i. This strategy allows us to avoid having to apply TSub to focus on a particular branch: instead, we simply infer a type for *each* branch, and then combine them using our disjunction operation. In order for the inferred type of a branch to make sense, we need to remove any occurrences of pattern variables or the ghost variable a. To do, we use the Ex function, which intuitively allows us to embed information from the typing context into a type.[13] This function takes as input a typing context Γ and type τ and produces an equivalent type $\tau <: \tau' <: \tau$ in which pattern and ghost variables do not appear free. Finally, ChkMatch uses Disj to ensure that the combination of the types of all the branches cover the required type $\Gamma \vdash \mathsf{Disj}(\overline{\tau'_i}) <: \tau$.

[13] The definition of Ex can be found in the supplementary materials.

Example 5.1. Consider how we might check that the body of the generator for natural numbers introduced in Section 2 has the expected type $[v{:}\mathit{int} \mid v \bmod 2 = 0]$:[14]

$$\begin{array}{l} \texttt{int\_gen}{:}\{v{:}\mathit{unit} \mid \top_{\mathsf{unit}}\} \to [v{:}\mathit{int} \mid \top_{\mathsf{int}}] \vdash \\ \texttt{let (n: int) = int\_gen() in let (b: bool) = n mod 2 == 0 in} \\ \texttt{match b with true -> err | false -> n} \;\Leftarrow [v{:}\mathit{int} \mid v \bmod 2 = 0] \end{array}$$

Our typing algorithm first adds the local variable n and b to the type context, and then checks the pattern-matching expression against the given type:

$$\begin{array}{l} \texttt{int\_gen}{:}\{v{:}\mathit{unit} \mid \top_{\mathsf{unit}}\} \to [v{:}\mathit{int} \mid \top_{\mathsf{int}}], \mathsf{n}{:}[v{:}\mathit{int} \mid \top_{\mathsf{int}}], \mathsf{b}{:}[v{:}\mathit{bool} \mid v \iff \mathsf{n} \bmod 2 = 0] \vdash \\ \texttt{match b with true -> err | false -> n} \;\Leftarrow [v{:}\mathit{int} \mid v \bmod 2 = 0] \end{array}$$

The ChkMatch rule first synthesizes types for the two branches separately. Inferring a type of the first branch using the existing type context:

$$..., \mathsf{b}{:}[v{:}\mathit{bool} \mid v \iff \mathsf{n} \bmod 2 = 0], \mathsf{b}'{:}[v{:}\mathit{bool} \mid v = \mathsf{b} \land v] \vdash \texttt{err} \;\Rightarrow [v{:}\mathit{int} \mid \bot]$$

adds a ghost variable b′ to reflect the fact that n must be less than 0 in this branch. By next applying the TErr rule, our algorithm infers the type $[v{:}\mathit{int} \mid \bot]$ for this branch. The rule next uses Ex to manifest b′ in the inferred type, encoding the path constraints under which this type holds (i.e. b is true).

$$..., \mathsf{b}{:}[v{:}\mathit{bool} \mid v \iff \mathsf{n} \bmod 2 = 0], \mathsf{b}'{:}[v{:}\mathit{bool} \mid v = \mathsf{b} \land v] \vdash \texttt{err} \;\Rightarrow [v{:}\mathit{int} \mid \exists \mathsf{b}', \mathsf{b}' = \mathsf{b} \land \mathsf{b}' \land \bot]$$

Thus, the synthesized type for the first branch is $[v{:}\mathit{int} \mid \mathsf{b} \land \bot]$ after trivial simplification. The type of the second branch provides a better demonstration of why Ex is needed:

$$..., \mathsf{b}{:}[v{:}\mathit{bool} \mid v \iff \mathsf{n} \bmod 2 = 0], \mathsf{b}'{:}[v{:}\mathit{bool} \mid v = \mathsf{b} \land \neg v] \vdash \texttt{n} \;\Rightarrow [v{:}\mathit{int} \mid v = \mathsf{n}]$$

After applying this operator, the inferred type is $[v{:}\mathit{int} \mid \exists \mathsf{b}', \mathsf{b}' = \mathsf{b} \land \neg \mathsf{b}' \land v = \mathsf{n}]$; after simplification, this becomes $[v{:}\mathit{int} \mid \neg \mathsf{b} \land v = \mathsf{n}]$. The disjunction of these two types:

$$\mathsf{Disj}([v{:}\mathit{int} \mid \mathsf{b} \land \bot], [v{:}\mathit{int} \mid \neg \mathsf{b} \land v = \mathsf{n}]) = [v{:}\mathit{int} \mid (\mathsf{b} \land \bot) \lor (\neg \mathsf{b} \land v = \mathsf{n})]$$

results in exactly the type shown in the Section 2 that can be then successfully checked against the target type $[v{:}\mathit{nat} \mid v \bmod 2 = 0]$.

Application. Our type synthesis rules for function application adopt a strategy similar to ChkMatch's, trying to infer the strongest type possible for an expression that uses the result of a function application. The rule for a function whose parameter is an overapproximate refinement type (SynAppBase) is most interesting, since it has to bridge the gap with an argument that has an underappproximate coverage type. When typing e, the expression that uses the result of the function call, the rule augments the typing context with a ghost variable a. This variable records that the coverage type of the argument must overlap with the type expected by the function (both must satisfy the refinement predicate ϕ): if this intersection is empty, i.e., the type of a is equivalent to $\bot$, we will fail to infer a type for e, as no type will be well-formed in this context. As with ChkMatch, SynAppBase uses Ex to ensure that it does not infer a type that depends on a.

5.2 Auxiliary Typing Functions

The disjunction operation is a straightforward syntactic transformation; its full definition can be found in the supplementary materials. More interesting are our implementation of the two other relations. Our type checking algorithm translates well-formedness and subtyping obligations into logical formulae that can be discharged by a SMT solver. Both obligations are encoded by the Query subroutine shown in Algorithm 1. $\mathsf{Query}(\Gamma, [v{:}b \mid \phi_1], [v{:}b \mid \phi_2])$ encodes the bindings in Γ in the typing context from right to left, before checking whether ϕ_1 implies ϕ_2. Variables with

[14] We have replaced the `if` expression from the original example with a `match` expression to be consistent with the language syntax (Figure 3).

Auxillary Typing Functions

$$\frac{\not\models \mathsf{Query}(\Gamma, [v{:}b \mid \bot], [v{:}b \mid \phi])}{\texttt{err} \notin [\![[v{:}b \mid \phi]]\!]_\Gamma} \qquad \frac{\models \mathsf{Query}(\Gamma, [v{:}b \mid \phi_1], [v{:}b \mid \phi_2])}{\Gamma \vdash [v{:}b \mid \phi_1] <: [v{:}b \mid \phi_2]} \qquad \frac{\models \mathsf{Query}(\Gamma, [v{:}b \mid \phi_2], [v{:}b \mid \phi_1])}{\Gamma \vdash \{v{:}b \mid \phi_1\} <: \{v{:}b \mid \phi_2\}}$$

Fig. 8. Auxillary Typing Algorithm

function types, on the other hand, are omitted entirely, as qualifiers cannot have function variables in FOL. Variables with an overapproximate (underapproximate) type are translated as a universally (existential) quantified variable, and are encoded into the refinement of both coverage types.

Example 5.2. Consider the subtyping obligation generated by Example 5.1 above:

$$\texttt{int\_gen}{:}\{v{:}unit \mid \top_{\mathsf{unit}}\} \rightarrow [v{:}int \mid \top_{\mathsf{int}}], \mathsf{n}{:}[v{:}int \mid \top_{\mathsf{int}}], \mathsf{b}{:}[v{:}bool \mid v \iff \mathsf{n} \bmod 2 = 0] \vdash$$
$$[v{:}int \mid (b \land \bot) \lor (\neg b \land v = n)] <: [v{:}int \mid v \geq 0]$$

This obligation is encoded by the following call to Query

$$\begin{aligned}
&\mathsf{Query}(\texttt{int\_gen}{:}\{v{:}unit \mid \top_{\mathsf{unit}}\} \rightarrow [v{:}int \mid \top_{\mathsf{int}}], \mathsf{n}{:}[v{:}int \mid \top_{\mathsf{int}}], \mathsf{b}{:}[v{:}bool \mid v \iff \mathsf{n} \bmod 2 = 0])\\
&\qquad [v{:}int \mid (\mathsf{b} \land \bot) \lor (\neg \mathsf{b} \land v = \mathsf{n})], [v{:}int \mid v \geq 0]) \equiv\\
&\mathsf{Query}(\texttt{int\_gen}{:}\{v{:}unit \mid \top_{\mathsf{unit}}\} \rightarrow [v{:}int \mid \top_{\mathsf{int}}], \mathsf{n}{:}[v{:}int \mid \top_{\mathsf{int}}],\\
&\qquad [v{:}int \mid \exists \mathsf{b}, \mathsf{b} \iff \mathsf{n} \bmod 2 = 0 \land (\mathsf{b} \land \bot) \lor (\neg \mathsf{b} \land v = \mathsf{n})], [v{:}int \mid \exists \mathsf{b}, \mathsf{b} \iff \mathsf{n} \bmod 2 = 0 \land v \geq 0]) \equiv\\
&\mathsf{Query}(\texttt{int\_gen}{:}\{v{:}unit \mid \top\} \rightarrow [v{:}int \mid \top_{\mathsf{int}}],\\
&\qquad [v{:}int \mid \exists \mathsf{n}, \top_{\mathsf{int}} \land \exists \mathsf{b}, \mathsf{b} \iff \mathsf{n} \bmod 2 = 0 \land (\mathsf{b} \land \bot) \lor (\neg \mathsf{b} \land v = \mathsf{n})],\\
&\qquad [v{:}int \mid \exists \mathsf{n}, \top_{\mathsf{int}} \land \exists \mathsf{b}, \mathsf{b} \iff \mathsf{n} \bmod 2 = 0 \land v \geq 0]) \equiv\\
&\mathsf{Query}(\emptyset, [v{:}int \mid \exists \mathsf{n}, \top_{\mathsf{int}} \land \exists \mathsf{b}, \mathsf{b} \iff \mathsf{n} \bmod 2 = 0 \land (\mathsf{b} \land \bot) \lor (\neg \mathsf{b} \land v = \mathsf{n})],\\
&\qquad\quad [v{:}int \mid \exists \mathsf{n}, \top_{\mathsf{int}} \land \exists \mathsf{b}, \mathsf{b} \iff \mathsf{n} \bmod 2 = 0 \land v \geq 0]) \equiv\\
&\qquad \forall v, \exists \mathsf{n}, \top_{\mathsf{int}} \land \exists \mathsf{b}, \mathsf{b} \iff \mathsf{n} \bmod 2 = 0 \land v \geq 0) \implies\\
&\qquad\quad \exists \mathsf{n}, \top_{\mathsf{int}} \land \exists \mathsf{b}, \mathsf{b} \iff \mathsf{n} \bmod 2 = 0 \land (\mathsf{b} \land \bot) \lor (\neg \mathsf{b} \land v = \mathsf{n})
\end{aligned}$$

This is equivalent to the formula (2) shown in Section 2:

$$\forall v, (v \geq 0) \implies (\exists \mathsf{n}, \exists \mathsf{b}, \mathsf{b} \iff \mathsf{n} \bmod 2 = 0 \land (\mathsf{b} \land \bot) \lor (\neg \mathsf{b} \land v = \mathsf{n}))$$

Using Query, it is straightforward to discharge well-formedness and subtyping obligations using the rules shown in Figure 8. In the case of WfBase, observe that the error term `err` is always an inhabitant of the type $[v{:}b \mid \bot]$ for arbitrary base type b. Thus, to check the last assumption of WfBase, it suffices to iteratively check if any coverage types in the type context are a supertype of their associated bottom type.

Discharging subtyping obligations is slightly more involved, as we need to ensure that the formulas sent to the SMT solver are decidable. Observe that in order to produce effectively decidable formulas, the encoding strategy realized by Query always generates a formula of the form $\forall \overline{x}.\exists \overline{y}.\phi$, i.e. it does not allow for arbitrary quantifier alternations. To ensure that this is sound strategy, we restrict all overapproximate refinement types in a type contexts to not have any free variables that have a coverage type. This constraint allows us to safely lift all universal quantifiers to the top level, thus avoiding arbitrary quantifier alternations.

As an example of a scenario disallowed by this restriction, consider the following type checking judgment:

$$x{:}[v{:}nat \mid v > 0]\ \ \texttt{fun (y: nat) -> x + y}\ \Leftarrow y{:}\{v{:}nat \mid v > x + 1\} \rightarrow [v{:}nat \mid \phi]$$

This judgment produces the following subtyping check:

$$x{:}[v{:}nat \mid v > 0], y{:}\{v{:}nat \mid v > x + 1\} \vdash [v{:}nat \mid v = x + y] <: [v{:}nat \mid \phi]$$

Algorithm 1: Subtyping Query

```
Procedure Query($(\Gamma, [v{:}b \mid \phi_1], [v{:}b \mid \phi_2])$ :=
    match $\Gamma$:
        case $\emptyset$ do
            return $\forall v{:}b, \phi_2 \implies \phi_1$;
        case $\Gamma, x{:}(a{:}\tau_a{\to}\tau)$ do
            $\phi \leftarrow$ Query($(\Gamma, [v{:}b \mid \phi_1], [v{:}b \mid \phi_2]$);
            return $\phi$;
        case $\Gamma, x{:}\{v{:}b_x \mid \phi_x\}$ do
            $\tau_1 \leftarrow [v{:}b \mid \forall x{:}b_x, \phi_x[v \mapsto x] \implies \phi_1]$;
            $\tau_2 \leftarrow [v{:}b \mid \forall x{:}b_x, \phi_x[v \mapsto x] \implies \phi_2]$;
            return Query($\Gamma, [v{:}b \mid \tau_1], [v{:}b \mid \tau_2]$);
        case $\Gamma, x{:}[v{:}b_x \mid \phi_x]$ do
            $\tau_1 \leftarrow [v{:}b \mid \exists x{:}b_x, \phi_x[v \mapsto x] \land \phi_1]$;
            $\tau_2 \leftarrow [v{:}b \mid \exists x{:}b_x, \phi_x[v \mapsto x] \land \phi_2]$;
            return Query($\Gamma, [v{:}b \mid \tau_1], [v{:}b \mid \tau_2]$);
```

where the normal refinement type $\{v{:}nat \mid v > x + 1\}$ in the type context has free variable x that has coverage type. Evaluating this judgment entails solving the formula:

$$\forall v, (\exists x, x > 0 \land (\forall y, y > x + 1 \implies \phi)) \implies (\exists x, x > 0 \land (\forall y, y > x + 1 \implies v = x + y))$$

which is not decidable due to the quantifier alternation $\forall v \exists x \forall y$.

Theorem 5.3 (Soundness of typing algorithm). *For all type context Γ, term e and coverage type τ, $\Gamma \vdash e \Leftarrow \tau \implies \Gamma \vdash e : \tau$*

6 EVALUATION

Implementation. We have implemented a coverage type checker, called Poirot, based on the above approach. Poirot targets OCaml programs that rely on libraries to manipulate algebraic data types; it consists of approximately 11K lines of OCaml code and uses Z3 [4] as its backend solver.

Poirot takes as input an Ocaml program representing a test input generator and a user-supplied coverage type for that generator. After basic type-checking and translation into MNF form, Poirot applies bi-directional type inference and checking to validate that the program satisfies the requirements specified by the type. Our implementation provides built-in coverage types for a number of OCaml primitives, including constants, various arithmetic operators, and data constructors for a range of datatypes. Refinements defined in coverage types can also use predefined (polymorphic) method predicates that capture non-trivial datatype shape properties. For example, the method predicate $mem(\mathtt{l}, \mathtt{u})$ indicates the element $\mathtt{u}{:}b$ is contained in the data type instance $\mathtt{l}{:}b\ T$; the method predicate $len(\mathtt{l}, 3)$ indicates the list $\mathtt{l}$ has length 3, or the tree $\mathtt{l}$ has depth 3. The semantics of these method predicates are defined as a set of FOL-encoded lemmas and axioms to facilitate automated verification; e.g., the lemma $len(\mathtt{l}, 0) \implies \forall \mathtt{u}, \neg mem(\mathtt{l}, \mathtt{u})$ indicates that the empty datatype instance contains no element.

Benchmarks. We have evaluated Poirot on a corpus[15] of hand-written non-trivial test input generators drawn from a variety of sources (see Table 1), including the Coq PBT framework QuickChick [22] (annotated with *), the Haskell PBT framework QuickCheck [3] (annotated with °), and bespoke test input generators collected from selected papers [20, 38] (annotated with $^\diamond$).

These benchmarks provide test input generators over a diverse range of datatypes, including various kinds of lists, trees, queues, streams, heaps, and sets. For each datatype implementation, Poirot type checks the provided implementation against its supplied coverage type to verify that the generator is able to generate all possible datatype instances consistent with this type. The properties that we check are non-trivial. For example, to check soundness of a red-black tree

[15] All benchmarks and the coverage types used to validate them are provided in the supplemental material.

Table 1. Experimental results.

	#Branch	#LocalVar	#MP	#Query	(max. #∀,#∃)	total (avg. time)(s)
SizedList*	$4^{\dagger}$	12	2	11	$(7,9)$	$0.35(0.03)$
SortedList*	$4^{\dagger}$	11	4	13	$(9,9)$	$6.77(0.52)$
UniqueList$^{\diamond}$	$3^{\dagger}$	8	3	10	$(7,7)$	$0.64(0.06)$
SizedTree*	$4^{\dagger}$	13	2	14	$(9,12)$	$0.48(0.03)$
CompleteTree*	$3^{\dagger}$	10	2	13	$(8,10)$	$0.38(0.03)$
RedBlackTree*	$6^{\dagger}$	36	3	70	$(16,53)$	$6.69(0.10)$
SizedBST*	$5^{\dagger}$	20	4	29	$(17,18)$	$12.20(0.42)$
BatchedQueue$^{\diamond}$	2	6	1	9	$(8,8)$	$0.52(0.06)$
BankersQueue$^{\diamond}$	2	6	1	11	$(8,8)$	$0.46(0.04)$
Stream$^{\diamond}$	4	13	2	13	$(9,11)$	$0.44(0.03)$
SizedHeap$^{\circ}$	$5^{\dagger}$	16	4	18	$(12,15)$	$3.89(0.22)$
LeftistHeap$^{\diamond}$	$3^{\dagger}$	11	1	16	$(9,11)$	$0.54(0.03)$
SizedSet$^{\circ}$	$4^{\dagger}$	16	4	23	$(14,15)$	$4.66(0.20)$
UnbalanceSet$^{\diamond}$	$5^{\dagger}$	20	4	29	$(17,18)$	$9.32(0.32)$

generator, we use the predicate $black\_height(v, n)$ to indicate that all branches of the tree v have exactly n black nodes, the predicate $no\_red\_red(v)$ to indicates v contains no red node with red children, and the predicate $root\_color(v, b)$ to indicate the root of the tree v has the red (black) color when the boolean value b is true (false). [16]

Given this rich set of predicates, it is straightforward to express interesting coverage types. For example, given size s and lower bound lo, we can express the property that a sorted list generator sorted_list_gen *must* generate all possible sorted lists with the length s and in which all elements are greater than or equal to lo, as the following type:

$$\mathtt{s}{:}\{v{:}int \mid v \leq 0\} \rightarrow \mathtt{lo}{:}\{v{:}int \mid \top\} \rightarrow [v{:}int\ list \mid len(v, \mathtt{s}) \land sorted(v) \land \forall u, mem(v, u) \implies \mathtt{lo} \leq u]$$

Notice that this type is similar to a normal refinement type:

$$\mathtt{s}{:}\{v{:}int \mid v \leq 0\} \rightarrow \mathtt{lo}{:}\{v{:}int \mid \top\} \rightarrow \{v{:}int\ list \mid len(v, \mathtt{s}) \land sorted(v) \land \forall u, mem(v, u) \implies \mathtt{lo} \leq u\}$$

with the return type marked as a coverage type to capture our desired must-property.

The first group of columns in Table 1 describes the salient features of our benchmarks. Each benchmark exhibits non-trivial control-flow, containing anywhere from 2 to 6 nested branches; a number of the benchmarks are also recursive (annotated with a †). The number of local (i.e., let-bound) variables (column #LocalVars) is a proxy for path lengths that must be encoded within the types inferred by our type-checker; column #MP indicates the number of method predicates found in the benchmark's type specification.

The second group of columns presents type checking results. Column #Query indicates the number of SMT queries that are triggered during type checking. Column $\#(\forall, \exists)$ indicates the maximum number of universal and existential quantifiers in these queries, respectively. The $\exists$ column is a direct reflection of control-flow (path) complexity - complex generators with deeply nested match-expressions like RedBlackTree result in queries with over 50 existential quantifiers. These numbers broadly track with the values in columns #Branch and #LocalVar. Despite the complexity of some of these queries, as evidenced by the number of their quantifiers, overall verification time, shown in the last column, is quite reasonable, with times ranging from .35 to 12.20 seconds, with seven of the 13 benchmarks verified in less than a second.

[16] These method predicates can be found in the implementation of the red-black tree generator given in [22].

```
1  let rec sized_list_gen
2      (size : int) : (int list) =
3    if (size == 0) then []
4    else
5      if (bool_gen  ()) then
6        sized_list_gen (size - 1)
7      else
8        int_gen () ::
9        (sized_list_gen (size - 1))
```

(a) A complete generator validated by Poirot.

```
let rec sized_list_gen
    (size : int) : (int list) =
  if (size == 0) then []
  else
    int_gen () ::
    (sized_list_gen (size - 1))
```

(b) A safe but incomplete generator.

```
let rec sized_list_gen
    (size : int) : (int list) =
  if (size == 0) then []
     else
       if (bool_gen  ()) then
        sized_list_gen (size - 1)
      else
        size ::
        (sized_list_gen   (size - 1))
```

(c) Another safe but incomplete generator.

Fig. 10. Three example generators that generate size-bounded lists.

6.1 Coverage Validation on Synthesized Generators

An underlying hypothesis motivating our work is that writing sound and *complete* test input generators can be subtle and tricky, as demonstrated by our motivating example (Figure 1). To justify this hypothesis, we repurposed an existing deductive component-based program synthesizer [24] to automatically synthesize correct (albeit possibly incomplete) generators that satisfy a specification given as an overapproximate refinement type; these generators are then fed to Poirot to validate their completeness. We provided the synthesizer with a datatype definition and a set of specifications describing constraints on that datatype the synthesized generator should use, along with a library of functions, including primitive generators such as `nat_gen`, available to the synthesizer for construction. A refinement type-guided enumeration is performed to find all correct programs consistent with the specification. Since the space of these programs is potentially quite large (possibly infinite), we constrain the synthesizer to only generate programs with bounded function call depths; in our experiments, this bound was set to three. The generator outputs all programs that are safe with respect to the specification. Figure 9 shows results of this experiment for five of the benchmarks given in Table 1. We report the total number of synthesized generators (#Total) constructed and the number of those that are *correct and complete* as verified by Poirot (#Complete). The table confirms our hypothesis that the space of complete generators with respect to the supplied coverage type is significantly smaller than the space of safe generators, as defined by the corresponding overapproximate refinement type specification.

Benchmark	#Total	#Complete
UniqueList	284	10
SizedList	126	28
SortedList	30	8
SizedTree	103	2
SizedBST	229	54

Fig. 9. Quantifying the space of safe and complete test input generators constructed using an automated program synthesis tool.

More concretely, Figure 10 shows three synthesized generators that satisfy the following specification of a list generate that is meant to generate *all* lists no longer than some provided bound:

$$\texttt{size}{:}\{v{:}int \mid v \leq 0\} \rightarrow [v{:}int\ list \mid \forall u, len(v,u) \implies (0 \leq u \land u \leq \texttt{size})]$$

Figure 10b is incomplete because it never generates an empty list when the size parameter `size` is greater than 0. On the other hand, while Figure 10c does generate empty lists, the else branch of its second conditional has a fixed first element, and will therefore never generate lists with distinct elements. The complete generator shown in Figure 10a incorporates a control-flow path (line 5) that can non-deterministically choose to make a recursive call to sized_list_gen with a smaller size, thereby allowing it to generate lists of variable size upto the `size` bound, including the empty list; another conditional branch uses `int_gen()` to generate a new randomly selected list element, thereby allowing the implementation to generate lists containing distinct elements. We emphasize again that Poirot was able to verify the correct generator and discard the two incorrect generators *automatically*, without any user involvement.

7 RELATED WORK

The effectiveness of PBT suffers when the property of interest has a strict precondition [18], because most of the inputs produced by a purely random test generation strategy will be simply discarded. Accordingly, there has been much recent interest on improving the coverage of test generators with respect to a particular precondition. Proposed solutions range from adopting ideas from fuzzing [6, 37] to intelligently mutate the outputs produced by the generator [20, 27], to focusing on generators for particular classes of inputs (e.g., well-typed programs) [9, 28, 36], to automatically building complete-by-construction generators [2, 19, 21]. While sharing broadly similar goals with these proposals, our approach differs significantly in its framing of coverage in purely type-theoretic terms. This fundamental change in perspective allows us to statically and compositionally verify coverage properties of a generator without the need for any form of instrumentation on, or runtime monitoring of, the program under test (as in [6, 20]). Expressing coverage as part of a type system also allows us to be agnostic to how generators are constructed, particulars of the application domain [9, 28, 36], and to the specific structure of the properties being tested [19, 21]. Poirot's ability to specify and type-check a complex coverage property depends only on whether we can express a desired specification using available method predicates.

A number of logics have been proposed for reasoning about underapproximations of program behavior, including the recently developed incorrectness logic (IL) [26, 30], reverse Hoare logic (RHL) [5], and dynamic logic (DL) [29]. Both IL and RHL are formalisms similar to Hoare logic, but support composable specifications that assert underapproximate postconditions, with IL adding special post-assertions for error states. IL was originally proposed as a way of formalizing the conditions under which a particular program point (say an error state) is guaranteed to be reachable, and has recently been used in program analyses that discover memory errors [23]. DL, in contrast, reinterprets Hoare logic as a multi-modal logic equipped with operators for reasoning about the existence of executions that end in a state satisfying some desired postcondition. In contrast to these logics, this paper provides the first development that interprets these notions in the context of a type system for a rich functional language. While our ideas are formulated in the context of verifying coverage properties for test input generators, we believe our framework can also help express type-based program analyses for bug finding or compiler optimizations.

Our approach shares similar goals with other refinement type systems for functional programs [34, 35], insofar as we encode verification conditions in a logic for which efficient solvers exist (e.g. SMT). Indeed, our setup follows exactly the same verification playbook as Liquid Types [31] - our underapproximate specifications are exactly the same as their overapproximate counterpart, except that our syntactically distinguished return types for functions reflects their expected underapproximate (rather than overapproximate) behavior. An important consequence of this design is that the burden of specifying and checking underapproximate (coverage) behavior of a program is no greater than in the overapproximate (safety) case.

8 CONCLUSIONS

This paper adapts principles of underapproximate reasoning found in recent work on Incorrectness Logic to the specification and automated verification of test input generators used in modern PBT systems. Specifications are expressed in the language of refinement types, augmented with coverage types, types that reflect underapproximate constraints on program behavior. A novel bi-directional type-checking algorithm enables an expressive form of inference over these types. Our experimental results demonstrate that our approach is capable of verifying both sophisticated hand-written generators, as well as being able to successfully identify type-correct (in an overapproximate sense) but coverage-incomplete generators produced from a deductive refinement type-aware synthesizer.

REFERENCES

[1] Koen Claessen. 2020. *QuickCheck.* https://hackage.haskell.org/package/QuickCheck

[2] Koen Claessen, Jonas Duregård, and Michał H. Pałka. 2014. Generating Constrained Random Data with Uniform Distribution. In *Functional and Logic Programming*. Springer International Publishing, 18–34.

[3] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

[4] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[5] Edsko de Vries and Vasileios Koutavas. 2011. Reverse Hoare Logic. In *Software Engineering and Formal Methods*, Gilles Barthe, Alberto Pardo, and Gerardo Schneider (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 155–171.

[6] Stephen Dolan. 2022. . https://github.com/stedolan/crowbar

[7] Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *ACM Comput. Surv.* 54, 5, Article 98 (may 2021), 38 pages. https://doi.org/10.1145/3450952

[8] FastCheck 2022. *fast-check: Property based testing for JavaScript and TypeScript.* https://dubzzz.github.io/fast-check.github.com/

[9] Burke Fetscher, Koen Claessen, Michal H. Palka, John Hughes, and Robert Bruce Findler. 2015. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 383–405. https://doi.org/10.1007/978-3-662-46669-8_16

[10] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 237–247. https://doi.org/10.1145/155090.155113

[11] Kimball Germane and Jay McCarthy. 2021. Newly-single and Loving It: Improving Higher-Order Must-Alias Analysis with Heap Fragments. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–28. https://doi.org/10.1145/3473601

[12] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. 2010. Compositional May-Must Program Analysis: Unleashing the Power of Alternation. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 43–56. https://doi.org/10.1145/1706299.1706307

[13] John Hatcliff and Olivier Danvy. 1994. A Generic Account of Continuation-Passing Styles. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 458–471. https://doi.org/10.1145/174675.178053

[14] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (oct 1969), 576–580. https://doi.org/10.1145/363235.363259

[15] Hypothesis 2022. *Hypothesis.* https://github.com/HypothesisWorks/hypothesis/tree/master/hypothesis-python

[16] Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew K. Wright. 1998. Single and Loving It: Must-Alias Analysis for Higher-Order Languages. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, David B. MacQueen and Luca Cardelli (Eds.). ACM, 329–341. https://doi.org/10.1145/268946.268973

[17] Ranjit Jhala and Niki Vazou. 2021. Refinement Types: A Tutorial. *Found. Trends Program. Lang.* 6, 3-4 (2021), 159–317. https://doi.org/10.1561/2500000032

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

[19] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner's Luck: A Language for Property-Based Generators. *SIGPLAN Not.* 52, 1 (jan 2017), 114–129. https://doi.org/10.1145/3093333.3009868

[20] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage Guided, Property Based Testing. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 181 (oct 2019), 29 pages. https://doi.org/10.1145/3360607

[21] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2018. Generating Good Generators for Inductive Relations. *Proc. ACM Program. Lang.* 2, POPL (2018), 45:1–45:30. https://doi.org/10.1145/3158133

[22] Leonidas Lampropoulos and Benjamin C. Pierce. 2022. *QuickChick: Property-Based Testing in Coq*. Software Foundations, Vol. 4. Electronic textbook. Version 1.3.1, https://softwarefoundations.cis.upenn.edu.

[23] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding Real Bugs in Big Programs with Incorrectness Logic. *Proc. ACM Program. Lang.* 6, OOPSLA (2022), 1–27. https://doi.org/10.1145/3527325

[24] Ashish Mishra and Suresh Jagannathan. 2022. Specification-Guided Component-Based Synthesis from Effectful Libraries. https://doi.org/10.48550/ARXIV.2209.02752 To appear Proc. ACM Program. Lang. (OOPSLA).

[25] Peter W. O'Hearn. 2019. Incorrectness Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 10 (dec 2019), 32 pages. https://doi.org/10.1145/3371078

[26] Peter W. O'Hearn. 2019. Incorrectness Logic. 4, POPL (2019). https://doi.org/10.1145/3371078

[27] 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*. Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576

[28] Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test* (Waikiki, Honolulu, HI, USA) *(AST '11)*. Association for Computing Machinery, New York, NY, USA, 91–97. https://doi.org/10.1145/1982595.1982615

[29] Vaughan R Pratt. 1976. Semantical consideration on Floyd-Hoare logic. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. IEEE, 109–121.

[30] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 225–252.

[31] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. *SIGPLAN Not.* 43, 6 (jun 2008), 159–169. https://doi.org/10.1145/1379022.1375602

[32] RustCheck 2021. *Crate for PBT in Rust*. https://github.com/BurntSushi/quickcheck

[33] ScalaCheck 2021. *ScalaCheck*. https://scalacheck.org/

[34] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. *SIGPLAN Not.* 49, 9 (aug 2014), 269–282. https://doi.org/10.1145/2692915.2628161

[35] Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 214–227. https://doi.org/10.1145/292540.292560

[36] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[37] Michal Zalewski. 2020. . https://github.com/google/afl

[38] Zhe Zhou, Robert Dickerson, Benjamin Delaware, and Suresh Jagannathan. 2021. Data-driven abductive inference of library specifications. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–29. https://doi.org/10.1145/3485493