Type Targeted Testing

Eric L. Seidel, Niki Vazou, and Ranjit Jhala

UC San Diego

Abstract. We present a new technique called *type targeted testing*, which translates precise *refinement types* into comprehensive test-suites. The key insight behind our approach is that through the lens of SMT solvers, refinement types can also be viewed as a high-level, declarative, test generation technique, wherein types are converted to SMT queries whose models can be decoded into concrete program inputs. Our approach enables the systematic and exhaustive testing of implementations from high-level declarative specifications, and furthermore, provides a gradual path from testing to full verification. We have implemented our approach as a Haskell testing tool called TARGET, and present an evaluation that shows how TARGET can be used to test a wide variety of properties and how it compares against state-of-the-art testing approaches.

1 Introduction

Should the programmer spend her time writing *better types* or *thorough tests*? Types have long been the most pervasive means of describing the intended behavior of code. However, a type signature is often a very coarse description; the actual inputs and outputs may be a subset of the actual values described by the types. For example, the set of ordered integer lists is a very sparse subset of the set of all integer lists. Thus, to validate functions that produce or consume such values, the programmer must painstakingly enumerate these values by hand or via ad-hoc generators for unit tests.

We present a new technique called *type targeted testing*, abbreviated to TARGET, that enables the generation of unit tests from precise *refinement types*. Over the last decade, various groups have shown how refinement types – which compose the usual types with logical refinement predicates that characterize the subset of actual type inhabitants – can be used to specify and formally verify a wide variety of correctness properties of programs [29,7,23,27]. Our insight is that through the lens of SMT solvers, refinement types can be viewed as a high-level, declarative, test generation technique.

TARGET tests an implementation function against a refinement type specification using a *query-decode-check* loop. First, TARGET translates the argument types into a logical *query* for which we obtain a satisfying assignment (or model) from the SMT solver. Next, TARGET *decodes* the SMT solver's model to obtain concrete input values for the function. Finally, TARGET executes the function on the inputs to get the corresponding output, which we *check* belongs to the specified result type. If the check fails, the inputs are returned as a counterexample, otherwise TARGET refutes the given model to force the SMT solver to return a different set of inputs. This process is repeated for a given number of iterations, or until *all* inputs up to a certain size have been tested.

TARGET offers several benefits over other testing techniques. Refinement types provide a succinct description of the input and output requirements, eliminating the need to enumerate individual test cases by hand or to write custom generators. Furthermore, TARGET generates *all* values (up to a given size) that inhabit a type, and thus does not skip any corner cases that a hand-written generator might miss. Finally, while the above advantages can be recovered by a brute-force generate-and-filter approach that discards inputs that do not meet some predicate, we show that our SMT-based method can be significantly more efficient for enumerating valid inputs in a highly-constrained space.

TARGET paves a *gradual path* from testing to verification, that affords several advantages over verification. First, the programmer has an *incentive* to write formal specifications using refinement types. TARGET provides the immediate gratification of an automatically generated, exhaustive suite of unit tests that can expose errors. Thus, the programmer is rewarded without paying, up front, the extra price of annotations, hints, strengthened inductive invariants, or tactics needed for formally verifying the specification. Second, our approach makes it possible to use refinement types to formally verify *some* parts of the program, while using tests to validate other parts that may not be cost effective for verification. TARGET integrates the two modes by using refinement types as the uniform specification mechanism. Functions in the verified half can be formally checked *assuming* the functions in the tested half adhere to their specifications. We could even use refinements to generate dynamic contracts [9] around the tested half if so desired. Third, even when formally verifying the type specifications, the generated tests can act as valuable *counterexamples* to help *debug* the specification or implementation in the event that the program is rejected by the verifier.

Finally, TARGET offers several concrete advantages over previous property-based testing techniques, which also have the potential for gradual verification. First, instead of specifying properties with arbitrary code [4,21] which complicates the task of subsequent formal verification, with TARGET the properties are specified via refinement types, for which there are already several existing formal verification algorithms [27]. Second, while symbolic execution tools [12,22,28] can generate tests from arbitrary code contracts (*e.g.* assertions) we find that highly constrained inputs trigger path explosion which precludes the use of such tools for gradual verification.

In the rest of this paper, we start with an overview of how TARGET can be used and how its query-decode-check loop is implemented (§ 2). Next, we formalize a general framework for type-targeted testing (§ 3) and show how it can be instantiated to generating tests for lists (§ 4), and then automatically generalized to other types (§ 4.6). All the benefits of TARGET come at a price; we are limited to properties that can be specified with refinement types. We present an empirical evaluation that shows TARGET is efficient and expressive enough to capture a variety of sophisticated properties, demonstrating that type-targeted testing is a sweet spot between automatic testing and verification (§ 5).

2 Overview

We start with a series of examples pertaining to a small grading library called `Scores`. The examples provide a bird's eye view of how a user interacts with TARGET, how TARGET is implemented, and the advantages of type-based testing.

Refinement Types A refinement type is one where the basic types are decorated with logical predicates drawn from an efficiently decidable theory. For example,

```
type Nat   = {v:Int | 0 <= v}
type Pos   = {v:Int | 0 <  v}
type Rng N = {v:Int | 0 <= v && v <  N}
```

are refinement types describing the set of integers that are non-negative, strictly positive, and in the interval `[0, N)` respectively. We will also build up function and collection types over base refinement types like the above. In this paper, we will not address the issue of *checking* refinement type signatures [27]. We assume the code is typechecked, *e.g.* by GHC, against the standard type signatures obtained by erasing the refinements. Instead, we focus on using the refinements to synthesize tests to *execute* the function, and to find *counterexamples* that violate the given specification.

2.1 Testing with Types

Base Types Let us write a function `rescale` that takes a source range `[0,r1)`, a target range `[0,r2)`, and a score `n` from the source range, and returns the linearly scaled score in the target range. For example, `rescale 5 100 2` should return `40`. Here is a first attempt at `rescale`

```
rescale :: r1:Nat -> r2:Nat -> s:Rng r1 -> Rng r2
rescale r1 r2 s = s * (r2 `div` r1)
```

When we run TARGET, it immediately reports

```
Found counter-example: (1, 0, 0)
```

Indeed, `rescale 1 0 0` results in `0` which is not in the target `Rng 0`, as the latter is empty! We could fix this in various ways, *e.g.* by requiring the ranges are non-empty:

```
rescale :: r1:Pos -> r2:Pos -> s:Rng r1 -> Rng r2
```

Now, TARGET accepts the function and reports

```
OK. Passed all tests.
```

Thus, using the refinement type *specification* for `rescale`, TARGET systematically tests the *implementation* by generating all valid inputs (up to a given size bound) that respect the pre-conditions, running the function, and checking that the output satisfies the post-condition. Testing against random, unconstrained inputs would be of limited value as the function is not designed to work on all `Int` values. While in this case we could filter invalid inputs, we shall show that TARGET can be more effective.

Containers Let us suppose we have normalized all scores to be out of `100`

```
type Score = Rng 100
```

Next, let us write a function to compute a *weighted* average of a list of scores.

```
average     :: [(Int, Score)] -> Score
average []   = 0
average wxs = total `div` n
  where
    total    = sum [w * x | (w, x) <- wxs ]
    n        = sum [w     | (w, _) <- wxs ]
```

It can be tricky to *verify* this function as it requires non-linear reasoning about an unbounded collection. However, we can gain a great degree of confidence by systematically testing it using the type specification; indeed, TARGET responds:

```
Found counter-example: [(0,0)]
```

Clearly, an unfortunate choice of weights can trigger a divide-by-zero; we can fix this by requiring the weights be non-zero:

```
average :: [({v:Int | v /= 0}, Score)] -> Score
```

but now TARGET responds with

```
Found counter-example: [(-3,3),(3,0)]
```

which also triggers the divide-by-zero! We will play it safe and require positive weights,

```
average :: [(Pos, Score)] -> Score
```

at which point TARGET reports that all tests pass.

Ordered Containers The very nature of our business requires that at the end of the day, we order students by their scores. We can represent ordered lists by requiring the elements of the tail `t` to be greater than the head `h`:

```
data OrdList a = [] | (:) {h :: a, t :: OrdList {v:a | h <= v}}
```

Note that erasing the refinement predicates gives us plain old Haskell lists. We can now write a function to insert a score into an ordered list:

```
insert :: (Ord a) => a -> OrdList a -> OrdList a
```

TARGET automatically generates all ordered lists (up to a given size) and executes `insert` to check for any errors. Unlike randomized testers, TARGET is not thwarted by the ordering constraint, and does not require a custom generator from the user.

Structured Containers Everyone has a few bad days. Let us write a function that takes the `best k` scores for a particular student. That is, the output must satisfy a *structural* constraint – that its size equals `k`. We can encode the size of a list with a logical measure function [27]:

```
measure len :: [a] -> Nat
len []       = 0
len (x:xs)   = 1 + len xs
```

Now, we can stipulate that the output indeed has `k` scores:

```
best       :: k:Nat -> [Score] -> {v:[Score] | k = len v}
best k xs = take k $ reverse $ sort xs
```

Now, TARGET quickly finds a counterexample:

```
Found counter-example: (2,[])
```

Of course – we need to have at least `k` scores to start with!

```
best :: k:Nat -> {v:[Score]|k <= len v} -> {v:[Score]|k = len v}
```

and now, TARGET is assuaged and reports no counterexamples. While randomized testing would suffice for `best`, we will see more sophisticated structural properties such as height balancedness, which stymie random testers, but are easily handled by TARGET.

Higher-order Functions Perhaps instead of taking the k best grades, we would like to pad each individual grade, and, furthermore, we want to be able to experiment with different padding functions. Let us rewrite `average` to take a functional argument, and stipulate that it can only increase a `Score`.

```
padAverage       :: (s:Score -> {v:Score | s <= v})
                 -> [(Pos, Score)] -> Score
padAverage f []  = f 0
padAverage f wxs = total `div` n
  where
    total    = sum [w * f x | (w, x) <- wxs ]
    n        = sum [w       | (w, _) <- wxs ]
```

TARGET automatically checks that `padAverage` is a safe generalization of `average`. Randomized testing tools can also generate functions, but those functions are unlikely to satisfy non-trivial constraints, thereby burdening the user with custom generators.

2.2 Synthesizing Tests

Next, let us look under the hood to get an idea of how TARGET synthesizes tests from types. At a high-level, our strategy is to: (1) *query* an SMT solver for satisfying assigments to a set of logical constraints derived from the refinement type, (2) *decode* the model into Haskell values that are suitable inputs, (3) *execute* the function on the decoded values to obtain the output, (4) *check* that the output satisfies the output type, (5) *refute* the model to generate a different test, and repeat the above steps until all tests up to a certain size are executed. We focus here on steps 1, 2, and 4 – query, decode, and check – the others are standard and require little explanation.

Base Types Recall the initial (buggy) specification

```
rescale :: r1:Nat -> r2:Nat -> s:Rng r1 -> Rng r2
```

TARGET *encodes* input requirements for base types directly from their corresponding refinements. The constraints for multiple, related inputs are just the *conjunction* of the constraints for each input. Hence, the constraint for `rescale` is:

$$\mathsf{C}_0 \doteq 0 \leq \mathit{r1} \wedge 0 \leq \mathit{r2} \wedge 0 \leq s < \mathit{r1}$$

In practice, C_0 will also contain conjuncts of the form $-N \leq x \leq N$ that restrict `Int`-valued variables x to be within the size bound N supplied by the user, but we will omit these throughout the paper for clarity.

Note how easy it is to capture dependencies between inputs, *e.g.* that the score `s` be in the range defined by `r1`. On querying the SMT solver with the above, we get a model $[r1 \mapsto 1, r2 \mapsto 1, s \mapsto 0]$. TARGET decodes this model and executes `rescale 1 1 0` to obtain the value `v = 0`. Then, TARGET validates `v` against the post-condition by checking the validity of the output type's constraint:

$$r2 = 1 \wedge v = 0 \wedge 0 \leq v \wedge v < r2$$

As the above is valid, TARGET moves on to generate another test by conjoining C_0 with a constraint that refutes the previous model:

$$\mathsf{C}_1 \doteq \mathsf{C}_0 \wedge (r1 \neq 1 \vee r2 \neq 1 \vee s \neq 0)$$

This time, the SMT solver returns a model: $[r1 \mapsto 1, r2 \mapsto 0, s \mapsto 0]$ which, when decoded and executed, yields the result 0 that does *not* inhabit the output type, and so is reported as a counterexample. When we fix the specification to only allow `Pos` ranges, each test produces a valid output, so TARGET reports that all tests pass.

Containers Next, we use TARGET to test the implementation of `average`. To do so, TARGET needs to generate Haskell lists with the appropriate constraints. Since each list is recursively either "nil" or "cons", TARGET generates constraints that symbolically represent *all* possible lists up to a given depth, using propositional *choice variables* to symbolically pick between these two alternatives. Every (satisfying) assignment of choices returned by the SMT solver gives TARGET the concrete data and constructors used at each level, allowing it to decode the assignment into a Haskell value.

For example, TARGET represents valid `[(Pos, Score)]` inputs (of depth up to 3), required to test `average`, as the conjunction of $\mathsf{C}_{\mathsf{list}}$ and $\mathsf{C}_{\mathsf{data}}$:

$$\begin{aligned}
\mathsf{C}_{\mathsf{list}} \doteq\ & (c_{00} \Rightarrow xs_0 = []) \wedge (c_{01} \Rightarrow xs_0 = x_1 : xs_1) \wedge (c_{00} \oplus c_{01}) \\
\wedge\ & (c_{10} \Rightarrow xs_1 = []) \wedge (c_{11} \Rightarrow xs_1 = x_2 : xs_2) \wedge (c_{01} \Rightarrow c_{10} \oplus c_{11}) \\
\wedge\ & (c_{20} \Rightarrow xs_2 = []) \wedge (c_{21} \Rightarrow xs_2 = x_3 : xs_3) \wedge (c_{11} \Rightarrow c_{20} \oplus c_{21}) \\
\wedge\ & (c_{30} \Rightarrow xs_3 = []) \wedge (c_{21} \Rightarrow c_{30})
\end{aligned}$$

$$\begin{aligned}
\mathsf{C}_{\mathsf{data}} \doteq\ & (c_{01} \Rightarrow x_1 = (w_1, s_1)\ \wedge\ 0 < w_1\ \wedge\ 0 \leq s_1 < 100) \\
\wedge\ & (c_{11} \Rightarrow x_2 = (w_2, s_2)\ \wedge\ 0 < w_2\ \wedge\ 0 \leq s_2 < 100) \\
\wedge\ & (c_{21} \Rightarrow x_3 = (w_3, s_3)\ \wedge\ 0 < w_3\ \wedge\ 0 \leq s_3 < 100)
\end{aligned}$$

The first set of constraints $\mathsf{C}_{\mathsf{list}}$ describes all lists up to size 3. At each level i, the *choice* variables c_{i0} and c_{i1} determine whether at that level the constructed list xs_i is a "nil" or a "cons". In the constraints $[]$ and $(:)$ are *uninterpreted* functions that represent "nil" and "cons" respectively. These functions only obey the congruence axiom and hence, can be efficiently analyzed by SMT solvers [19]. The data at each level x_i is constrained to be a pair of a positive weight w_i and a valid score s_i.

The choice variables at each level are used to *guard* the constraints on the next levels. First, if we are generating a "cons" at a given level, then exactly one of the choice variables for the next level must be selected; *e.g.* $c_{11} \Rightarrow c_{20} \oplus c_{21}$. Second, the constraints on the data at a given level only hold if we are generating values for that level; *e.g.* c_{21} is used to guard the constraints on x_3, w_3 and s_3. This is essential to avoid over-constraining the system which would cause TARGET to miss certain tests.

To *decode* a model of the above into a Haskell value of type `[(Int, Int)]`, we traverse constraints and use the valuations of the choice variables to build up the list appropriately. At each level, if $c_{i0} \mapsto \mathit{true}$, then the list at that level is `[]`, otherwise $c_{i1} \mapsto \mathit{true}$ and we decode x_{i+1} and xs_{i+1} and "cons" the results.

We can iteratively generate *multiple* inputs by adding a constraint that refutes each prior model. As an important optimization, we only refute the relevant parts of the model, *i.e.* those needed to construct the list (§ 4.5).

Ordered Containers Next, let us see how TARGET enables automatic testing with highly constrained inputs, such as the *increasingly ordered* `OrdList` values required by `insert`. From the type definition, it is apparent that ordered lists are the same as the usual lists described by $\mathsf{C}_{\mathsf{list}}$, except that each unfolded *tail* must only contain values that are greater than the corresponding *head*. That is, as we unfold `x1:x2:xs :: OrdList`

- At level `0`, we have `OrdList {v:Score| true}`
- At level `1`, we have `OrdList {v:Score| x1 <= v}`
- At level `2`, we have `OrdList {v:Score| x2 <= v && x1 <= v}`

and so on. Thus, we encode `OrdList Score` (of depth up to 3) by conjoining $\mathsf{C}_{\mathsf{list}}$ with $\mathsf{C}_{\mathsf{score}}$ and $\mathsf{C}_{\mathsf{ord}}$, which capture the valid score and ordering requirements respectively:

$$\begin{aligned} \mathsf{C}_{\mathsf{ord}} &\doteq (c_{11} \Rightarrow x_1 \le x_2) \;\wedge\; (c_{21} \Rightarrow x_2 \le x_3 \;\wedge\; x_1 \le x_3) \\ \mathsf{C}_{\mathsf{score}} &\doteq (c_{01} \Rightarrow 0 \le x_1 < 100) \;\wedge\; (c_{11} \Rightarrow 0 \le x_2 < 100) \;\wedge\; (c_{21} \Rightarrow 0 \le x_3 < 100) \end{aligned}$$

Structured Containers Recall that `best k` requires inputs whose *structure* is constrained – the size of the list should be no less than `k`. We specify size using special measure functions [27], which let us relate the size of a list with that of its unfolding, and hence, let us encode the notion of size inside the constraints:

$$\begin{aligned} \mathsf{C}_{\mathsf{size}} \;\doteq\; & (c_{00} \Rightarrow \mathit{len}\ xs_0 = 0) \wedge (c_{01} \Rightarrow \mathit{len}\ xs_0 = 1 + \mathit{len}\ xs_1) \\ \wedge\; & (c_{10} \Rightarrow \mathit{len}\ xs_1 = 0) \wedge (c_{11} \Rightarrow \mathit{len}\ xs_1 = 1 + \mathit{len}\ xs_2) \\ \wedge\; & (c_{20} \Rightarrow \mathit{len}\ xs_2 = 0) \wedge (c_{21} \Rightarrow \mathit{len}\ xs_2 = 1 + \mathit{len}\ xs_3) \\ \wedge\; & (c_{30} \Rightarrow \mathit{len}\ xs_3 = 0) \end{aligned}$$

At each unfolding, we instantiate the definition of the measure for each alternative of the datatype. In the constraints, $\mathit{len} \cdot$ is an uninterpreted function derived from the measure definition. All of the relevant properties of the function are spelled out by the unfolded constraints in $\mathsf{C}_{\mathsf{size}}$ and hence, we can use SMT to search for models for the above constraint. Hence, TARGET constrains the input type for `best` as:

$$0 \le k \wedge \mathsf{C}_{\mathsf{list}} \wedge \mathsf{C}_{\mathsf{score}} \wedge \mathsf{C}_{\mathsf{size}} \wedge k \le \mathit{len}\ xs_0$$

where the final conjunct comes from the top-level refinement that stipulates the input have at least `k` scores. Thus, TARGET only generates lists that are large enough. For example, in any model where $k = 2$, it will *not* generate the empty or singleton list, as in those cases, $\mathit{len}\ xs_0$ would be 0 (resp. 1), violating the final conjunct above.

```
-- Manipulating Refinements
refinement :: RefType -> Refinement
subst      :: RefType -> [(Var, Var)] -> RefType

-- Manipulating Types
unfold     :: Ctor  -> RefType -> [(Var, RefType)]
binder     :: RefType -> Var
proxy      :: RefType -> Proxy a
```

Fig. 1. Refinement Type API

Higher-order Functions Finally, TARGET's type-directed testing scales up to higher-order functions using the same insight as in QuickCheck [4], namely, to generate a function it suffices to be able to generate the *output* of the function. When tasked with the generation of a functional argument `f`, TARGET returns a Haskell function that when executed checks whether its inputs satisfy `f`'s pre-conditions. If they do, then `f` uses TARGET to dynamically query the SMT solver for an output that satisfies the constraints imposed by the concrete inputs. Otherwise, `f`'s specifications are violated and TARGET reports a counterexample.

This concludes our high-level tour of the benefits and implementation of TARGET. Notice that the property specification mechanism – refinement types – allowed us to get immediate feedback that helped debug not just the code, but also the specification itself. Additionally, the specifications gave us machine-readable documentation about the behavior of functions, and a large unit test suite with which to automatically validate the implementation. Finally, though we do not focus on it here, the specifications are amenable to formal verification should the programmer so desire.

3 A Framework for Type Targeted Testing

Next, we describe a framework for type targeted testing, by formalizing an abstract representation of refinement types (§ 3.1), describing the operations needed to generate tests from types (§ 3.2), and then using the above to implement TARGET via a query-decode-check loop (§ 3.3). Subsequently, we instantiate the framework to obtain tests for refined primitive types, lists, algebraic datatypes and higher-order functions (§ 4).

3.1 Refinement Types

A refinement type is a type, where each component is decorated with a predicate from a refinement logic. For clarity, we describe refinement types and refinements abstractly as `RefType` and `Refinement` respectively. We write `Var` as an alias for `Refinement` that is typically used to represent logical variables appearing within the refinement. In the sequel, we will use backticks to represent the various entities in the meta-language used to describe TARGET. For example, `` `x0` ``, `` `k <= len v` ``, and `{v:[Score] | x0 <= len v}` are the `Var`, `Refinement`, and `RefType` representing the corresponding entities within the backticks.

Next, we describe the various operations over them needed to implement TARGET. These operations, summarized in Figure 1, fall into two categories: those which manipulate the *refinements* and those which manipulate the *types*.

Operating on Refinements To generate constraints and check inhabitation, we use the function `refinement` which returns the (top-level) refinement that decorates the given refinement type. We will generate fresh `Var`s to name values of components, and will use `subst` to replace the free occurrences of variables in a given `RefType`. Suppose that `t` is the `RefType` represented by `{v:[Score] | k <= len v}`. Then,

– `refinement t` evaluates to `k <= len v` and
– `subst t [(`k`, `x0`)]` evaluates to `{v:[Score] | x0 <= len v}`.

Operating on Types To build up compound values (*e.g.* lists) from components (*e.g.* an integer and a list), `unfold` breaks a `RefType` (*e.g.* a list of integers) into its constituents (*e.g.* an integer and a list of integers) at a given constructor (*e.g.* "cons"). `binder` simply extracts the `Var` representing the value being refined from the `RefType`. To write generic functions over `RefType`s and use Haskell's type class machinery to `query` and `decode` components of types, we associate with each refinement type a *proxy* representing the corresponding Haskell type (in practice this must be passed around as a separate argument). For example, if `t` is `{v:[Score] | k <= len v}`,

– `unfold `:` t` evaluates to `[(`x`, `Score`), (`xs`, `[Score]`)]`,
– `binder t` evaluates to `v`, and
– `proxy t` evaluates to a value of type `Proxy [Int]`.

3.2 The `Targetable` Type Class

Following QuickCheck, we encapsulate the key operations needed for type-targeted testing in a type class `Targetable` (Figure 2). This class characterizes the set of types

```
class Targetable a where
  query  :: Proxy a -> Int -> RefType -> SMT Var
  decode :: Var -> SMT a
  check  :: a -> RefType -> SMT (Bool, Var)
```

Fig. 2. The class of types that can be tested by TARGET

which can be tested by TARGET. All of the operations can interact with an external SMT solver, and so return values in an `SMT` monad.

– `query` takes a *proxy* for the Haskell type for which we are generating values, an integer *depth* bound, and a *refinement type* describing the desired constraints, and generates a set of logical constraints and a `Var` that represents the constrained value.

– decode takes a Var, generated via a previous query and queries the model returned by the SMT solver to construct a Haskell value of type a.
– check takes a value of type a, translates it back into logical form, and verifies that it inhabits the output type t.

3.3 The Query-Decode-Check Loop

Figure 3 summarizes the overall implementation of TARGET, which takes as input a function f and its refinement type specification t and proceeds to test the function against the specification via a *query-decode-check* loop: (1) First, we translate the refined inputTypes into a logical *query*. (2) Next, we *decode* the model (*i.e.* satisfying assignment) for the query returned by the SMT solver to obtain concrete inputs. (3) Finally, we execute the function f on the inputs to get the corresponding output, which we check belongs to the specified outputType. If the check fails, we return the inputs as a counterexample. After each test, TARGET, refutes the given test to force the SMT solver to return a different set of inputs, and this process is repeated until a user specified number of iterations. The checkSMT call may fail to find a model meaning that we have exhaustively tested all inputs upto a given testDepth bound. If all iterations succeed, *i.e.* no counterexamples are found, then TARGET returns Ok, indicating that f satisfies t up to the given depth bound.

```
target f t = do
  vars  <- forM (inputTypes t) $ \t ->
             query (proxy t) testDepth t           -- Query
  forM [1 .. testNum] $ \_ -> do
    hasModel <- checkSMT
    when hasModel $ do
      inputs <- forM vars decode                   -- Decode
      output <- execute f inputs
      (ok,_) <- check output (outputType t)        -- Check
      if ok then
        refuteSMT
      else
        throw (CounterExample inputs)
  return Ok
```

Fig. 3. Implementing TARGET via a *query-decode-check* loop

4 Instantiating the TARGET Framework

Next, we describe a concrete instantiation of TARGET for lists. We start with a constraint generation API (§ 4.1). Then we use the API to implement the key operations query (§ 4.2), decode (§ 4.3), check (§ 4.4), and refuteSMT (§ 4.5), thereby enabling TARGET to automatically test functions over lists. Finally, we show how the list instance can be generalized to algebraic datatypes and higher-order functions (§ 4.6).

4.1 SMT Solver Interface

Figure 4 describes the interface to the SMT solvers that TARGET uses for constraint generation and model decoding. The interface has functions to (a) generate logical variables of type `Var`, (b) constrain their values using `Refinement` predicates, and (c) determine the values assigned to the variables in satisfying models.

```
fresh     :: SMT Var
guard     :: Var -> SMT a        -> SMT a
constrain :: Var -> Refinement  -> SMT ()

apply     :: Ctor -> [Var] -> SMT Var
unapply   :: Var  -> SMT (Ctor, [Var])

oneOf     :: Var -> [(Var, Var)] -> SMT ()
whichOf   :: Var -> SMT Var

eval      :: Refinement -> SMT Bool
```

Fig. 4. SMT Solver API

- `fresh` allocates a new logical variable.
- `guard b act` ensures that all the constraints generated by `act` are *guarded by* the choice variable `b`. That is, if `act` generates the constraint p then `guard b act` generates the (implication) constraint $b \Rightarrow p$.
- `constrain x r` generates a constraint that `x` satisfies the refinement predicate `r`.
- `apply c xs` generates a new `Var` for the folded up value obtained by applying the constructor `c` to the fields `xs`, while also generating constraints from the measures. For example, `apply ‘:‘ [‘x1‘, ‘xs1‘]` returns $x_1 : xs_1$ and generates the constraint $len\ (x_1 : xs_1) = 1 + len\ xs_1$.
- `unapply x` returns the `Ctor` and `Var`s from which the input `x` was constructed.
- `oneOf x cxs` generates a constraint that `x` equals exactly one of the elements of `cxs`. For example, `oneOf ‘xs0‘ [(‘c00‘,‘[]‘),(‘c01‘,‘x1 : xs1‘)]` yields:
$$c_{00} \Rightarrow xs_0 = [] \wedge c_{01} \Rightarrow xs_0 = x_1 : xs_1 \wedge c_{00} \oplus x_{01}$$
- `whichOf x` returns the particular alternative that was assigned to `x` in the current model returned by the SMT solver. Continuing the previous example, if the model sets `c00` (resp. `c01`) to *true*, `whichOf ‘xs0‘` returns `‘[]‘` (resp `‘x1 : xs1‘`).
- `eval r` checks the validity of a refinement with no free variables. For example, `eval ‘len (1 : [])> 0‘` would return `True`.

```
query p d t = do
  let cs = ctors d
  bs <- forM cs $ \_ -> fresh
  xs <- zipWithM (queryCtor (d-1) t) bs cs
  x  <- fresh
  oneOf x     $ zip bs xs
  constrain x $ refinement t
  return x

queryCtor d t b c = guard b $ do
  let fts = unfold c t
  fs'     <- scanM (queryField d) [] fts
  x       <- apply c fs'
  return x

queryField d su (f, t) = do
  f' <- query (proxy t) d $ t `subst` su
  return ((f, f') : su, f')

ctors d
  | d > 0      = [ `:`, `[]` ]
  | otherwise = [       `[]` ]
```

Fig. 5. Generating a Query

4.2 Query

Figure 5 shows the procedure for constructing a `query` from a refined list type, *e.g.* the one required as an input to the `best` or `insert` functions from § 2.

Lists `query` returns a `Var` that represent *all* lists up to depth `d` that satisfy the logical constraints associated with the refined list type `t`. To this end, it invokes `ctors` to obtain all of the suitable constructors for depth `d`. For lists, when the depth is `0` we should only use the `` `[]` `` constructor, otherwise we can use either `` `:` `` or `` `[]` ``. This ensures that `query` terminates after encoding all possible lists up to a given depth `d`. Next, it uses `fresh` to generate a distinct *choice* variable for each constructor, and calls `queryCtor` to generate constraints and a corresponding symbolic `Var` for each constructor. The choice variable for each constructor is supplied to `queryCtor` to ensure that the constraints are *guarded*, *i.e.* only required to hold *if* the corresponding choice variable is selected in the model and not otherwise. Finally, a fresh `x` represents the value at depth `d` and is constrained to be `oneOf` the alternatives represented by the constructors, and to satisfy the top-level refinement of `t`.

Constructors `queryCtor` takes as input the refined list type `t`, a depth `d`, a particular constructor `c` for the list type, and generates a query describing the *unfolding* of `t` at the constructor `c`, guarded by the choice variable `b` that determines whether this alternative is indeed part of the value. These constraints are the conjunction of those describing the

```
                                 decodeCtor `[]` []     = return []
decode x = do                    decodeCtor `:` [x,xs]  = do
  x'       <- whichOf x            v  <- decode x
  (c,fs') <- unapply x'            vs <- decode xs
  decodeCtor c fs'                 return (v:vs)
```

Fig. 6. Decoding Models into Haskell Values

values of the individual fields which can be combined via `c` to obtain a `t` value. To do so, `queryCtor` first `unfold`s the type `t` at `c`, obtaining a list of constituent fields and their respective refinement types `fts`. Next, it uses

```
scanM :: Monad m => (a -> b -> m (a, c)) -> a -> [b] -> m [c]
```

to traverse the fields from left to right, building up representations of values for the fields from their unfolded refinement types. Finally, we invoke `apply` on `c` and the fields `fs'` to return a symbolic representation of the constructed value that is constrained to satisfy the measure properties of `c`.

Fields `queryField` generates the actual constraints for a single field `f` with refinement type `t`, by invoking `query` on `t`. The `proxy` enables us to resolve the appropriate type-class instance for generating the query for the field's value. Each field is described by a new symbolic name `f'` which is `subst`ituted for the formal name of the field `f` in the refinements of subsequent fields, thereby tracking dependencies between the fields. For example, these substitutions ensure the values in the tail are greater than the head as needed by `OrdList` from § 2.

4.3 Decode

Once we have generated the constraints we query the SMT solver for a model, and if one is found we must *decode* it into a concrete Haskell value with which to test the given function. Figure 6 shows how to decode an SMT model for lists.

Lists `decode` takes as input the top-level symbolic representation `x` and queries the model to determine which alternative was assigned by the solver to `x`, *i.e.* a nil or a cons. Once the alternative is determined, we use `unapply` to destruct it into its constructor `c` and fields `fs'`, which are recursively decoded by `decodeCtor`.

Constructors `decodeCtor` takes the constructor `c` and a list of symbolic representations for fields, and decodes each field into a value and applies the constructor to obtain the Haskell value. For example, in the case of the `` `[]` `` constructor, there are no fields, so we return the empty list. In the case of the `` `:` `` constructor, we decode the head and the tail, and cons them to return the decoded value. `decodeCtor` has the type

```
(Targetable a) => Ctor -> [Var] -> SMT [a]
```

i.e. if `a` is a decodable type, then `decodeCtor` suffices to decode lists of `a`. Primitives like integers that are directly encoded in the refinement logic are the base case – *i.e.* the value in the model is directly translated into the corresponding Haskell value.

```
check v t = do
  let (c,vs) = splitCtor v
  let fts    = unfold c t
  (bs, vs') <- unzip <$> scanM checkField [] (zip vs fts)
  v'        <- apply c vs'
  let t'     = t `subst` [(binder t, v')]
  b'        <- eval $ refinement t'
  return (and (b:bs), v')

checkField su (v, (f, t)) = do
  (b, v') <- check v $ t `subst` su
  return ((f, v') : su, (b, v'))

splitCtor []     = ('[]', [])
splitCtor (x:xs) = (':', [x,xs])
```

Fig. 7. Checking Outputs

4.4 Check

The third step of the query-decode-check loop is to verify that the output produced by the function under test indeed satisfies the output refinement type of the function. We accomplish this by *encoding* the output value as a logical expression, and evaluating the output refinement applied to the logical representation of the output value.

`check`, shown in Figure 7, takes a Haskell (output) value `v` and the (output) refinement type `t`, and recursively verifies each component of the output type. It converts each component into a logical representation, `subst`itutes the logical expression for the symbolic value, and `eval`uates the resulting `Refinement`.

4.5 Refuting Models

Finally, TARGET invokes `refuteSMT` to *refute* a given model in order to force the SMT solver to produce a different model that will yield a different test input. A naïve implementation of refutation is as follows. Let X be the set of all variables appearing in the constraints. Suppose that in the current model, each variable x is assigned the value $\sigma(x)$. Then, to refute the model, we add a *refutation constraint* $\vee_{x \in X} x \neq \sigma(x)$. That is, we stipulate that *some* variable be assigned a different value.

The naïve implementation is extremely inefficient. The SMT solver is free to pick a different value for some *irrelevant* variable which was not even used for decoding. As a result, the next model can, after decoding, yield the *same* Haskell value, thereby blowing up the number of iterations needed to generate all tests of a given size.

TARGET solves this problem by forcing the SMT solver to return models that yield *different decoded tests* in each iteration. To this end TARGET restricts the refutation constraint to the set of variables that were actually used to `decode` the Haskell value.

We track this set by instrumenting the `SMT` monad to log the set of variables and choice-variables that are transitively queried via the recursive calls to `decode`. That is, each call to `decode` logs its argument, and each call to `whichOf` logs the choice variable corresponding to the alternative that was returned. Let R be the resulting set of *decode-relevant* variables. TARGET refutes the model by using a *relevant refutation constraint* $\vee_{x \in R} x \neq \sigma(x)$ which ensures that the next model decodes to a different value.

4.6 Generalizing TARGET To Other Types

The implementation in § 4 is for List types, but `ctors`, `decodeCtor`, and `splitCtor` are the only functions that are List-specific. Thus, we can easily generalize the implementation to:

- *primitive datatypes*, *e.g.* integers, by returning an empty list of constructors,
- *algebraic datatypes*, by implementing `ctors`, `decodeCtor`, and `splitCtor` for that type.
- *higher-order functions*, by lifting instances of `a` to functions returning `a`.

Algebraic Datatypes Our List implementation has three pieces of type-specific logic:

- `ctors`, which returns a list of constructors to unfold;
- `decodeCtor`, which decodes a specific `Ctor`; and
- `splitCtor`, which splits a Haskell value into a pair of its `Ctor` and fields.

Thus, to instantiate TARGET on a new data type, all we need is to implement these three operations for the type. This implementation essentially follows the concrete template for Lists. In fact, we observe that the recipe is entirely mechanical boilerplate, and can be fully automated for *all* algebraic data types by using a *generics* library.

Any algebraic datatype (ADT) can be represented as a *sum-of-products* of component types. A generics library, such as GHC.Generics [15], provides a *univeral* sum-of-products type and functions to automatically convert any ADT to and from the universal representation. Thus, to obtain `Targetable` instances for *any* ADT it suffices to define a `Targetable` instance for the *universal* type.

Once the universal type is `Targetable` we can automatically get an instance for any new user-defined ADT (that is an instance of `Generic`) as follows: (1) to generate a *query* we simply create a query for GHC.Generics' universal representation of the refined type, (2) to *decode* the results from the SMT solver, we decode them into the universal representation and then use GHC.Generics to map them back into the user-defined type, (3) to *check* that a given value inhabits a user-defined refinement type, we check that the universal representation of the value inhabits the type's universal counterpart.

The `Targetable` instance for the universal representation is a generalized version of the List instance from § 4, that relies on various technical details of GHC.Generics.

Higher Order Functions Our type-directed approach to specification makes it easy to extend TARGET to higher-order functions. Concretely, it suffices to implement a type-class instance:

```
instance (Targetable input, Targetable output)
  => Targetable (input -> output)
```

In essence, this instance uses the `Targetable` instances for `input` and `output` to create an instance for functions from `input -> output`, after which Haskell's type class machinery suffices to generate concrete function values.

To create such instances, we use the insight from QuickCheck, that to generate (constrained) functions, we need only to generate *output* values for the function. Following this route, we generate functions by creating new lambdas that take in the inputs from the calling context, and use their values to create queries for the output, after which we can call the SMT solver and decode the results to get concrete outputs that are returned by the lambda, completing the function definition. Note that we require `input` to also be `Targetable` so that we can encode the Haskell value in the refinement logic, in order to constrain the output values suitably. We additionally memoize the generated function to preserve the illusion of purity. It is also possible to, in the future, extend our implementation to refute functions by asserting that the output value for a given input be distinct from any previous outputs for that input.

5 Evaluation

We have built a prototype implementation of TARGET and next, describe an evaluation on a series of benchmarks ranging from textbook examples of algorithms and data structures to widely used Haskell libraries like CONTAINERS and XMONAD. Our goal in this evaluation is two-fold. First, we describe micro-benchmarks (*i.e.* functions) that *quantitatively compare* TARGET with the existing state-of-the-art, property-based testing tools for Haskell – namely SmallCheck and QuickCheck– to determine whether TARGET is indeed able to generate highly constrained inputs more effectively. Second, we describe macro-benchmarks (*i.e.* modules) that evaluate the amount of *code coverage* that we get from type-targeted testing.

5.1 Comparison with QuickCheck and SmallCheck

We compare TARGET with QuickCheck and SmallCheck by using a set of benchmarks with highly constrained inputs. For each benchmark we compared TARGET with SmallCheck and QuickCheck, with the latter two using the generate-and-filter approach, wherein a value is generated and subsequently discarded if it does not meet the desired constraint. While one could possibly write custom "operational" generators for each property, the point of this evaluation is compare the different approaches ability to enable "declarative" specification driven testing. Next, we describe the benchmarks and then summarize the results of the comparison (Figure 8).

Inserting into a sorted List Our first benchmark is the insert function from the homonymous sorting routine. We use the specification that given an element and a sorted list, `insert x xs` should evaluate to a sorted list. We express this with the type

```
type Sorted a = List <{\hd v -> hd < v}> a
insert :: a -> Sorted a -> Sorted a
```

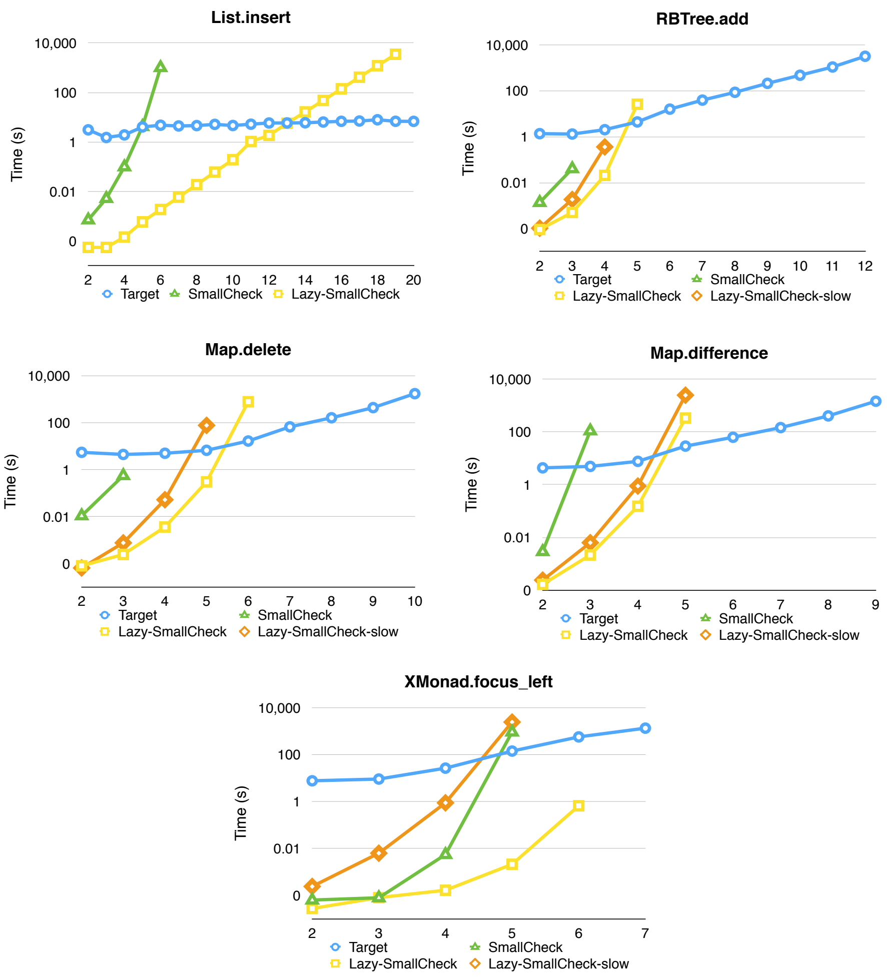

Fig. 8. Results of comparing TARGET with QuickCheck, SmallCheck, and Lazy SmallCheck on a series of functions. TARGET, SmallCheck, and Lazy SmallCheck were both configured to check the first 1000 inputs that satisfied the precondition at increasing depth parameters, with a 60 minute timeout per depth; QuickCheck was run with the default settings, *i.e.* it had to produce 100 test cases. TARGET, SmallCheck, and Lazy SmallCheck were configured to use the same notion of depth, in order to ensure they would generate the same number of valid inputs at each depth level. QuickCheck was unable to successfully complete any run due to the low probability of generating valid inputs at random.

where the ordering constraint is captured by an abstract refinement [25] which states that *each* list head `hd` is less than every element `v` in its tail.

Inserting into a Red-Black Tree Next, we consider insertion into a Red-Black tree.

```
data RBT a = Leaf  | Node Col a (RBT a) (RBT a)
data Col    = Black | Red
```

Red-black trees must satisfy three invariants: (1) red nodes always have black children, (2) the black height of all paths from the root to a leaf is the same, and (3) the elements in the tree should be ordered. We capture (1) via a measure that recursively checks each `Red` node has `Black` children.

```
measure isRB :: RBT a -> Prop
isRB Leaf             = true
isRB (Node c x l r) = isRB l && isRB r &&
                       (c == Red => isBlack l && isBlack r)
```

We specify (2) by defining the `Black` height as:

```
measure bh :: RBT a -> Int
bh Leaf             = 0
bh (Node c x l r) = bh l + (if c == Red then 0 else 1)
```

and then checking that the `Black` height of both subtrees is the same:

```
measure isBH :: RBT a -> Prop
isBH Leaf             = true
isBH (Node c x l r) = isBH l && isBH r && bh l == bh r
```

Finally, we specify the (3), the ordering invariant as:

```
type OrdRBT a = RBT <{\r v -> v < r}, {\r v -> r < v}> a
```

i.e. with two abstract refinements for the left and right subtrees respectively, which state that the root `r` is greater than (resp. less than) each element `v` in the subtrees. Finally, a valid Red-Black tree is:

```
type OkRBT a = {v:OrdRBT a | isRB v && isBH v}
```

Note that while the specification for the *internal* invariants for Red-Black trees is tricky, the specification for the public API – *e.g.* the `add` function – is straightforward:

```
add :: a -> OkRBT a -> OkRBT a
```

Deleting from a Data.Map Our third benchmark is the `delete` function from the `Data.Map` module in the Haskell standard libraries. The `Map` structure is a balanced binary search tree that implements purely functional key-value dictionaries:

```
data Map k a = Tip | Bin Int k a (Map k a) (Map k a)
```

A valid `Data.Map` must satisfy two properties: (1) the size of the left and right subtrees must be within a factor of three of each other, and (2) the keys must obey a binary search ordering. We specify the balancedness invariant (1) with a measure

```
measure isBal :: Map k a -> Prop
isBal (Tip)           = true
isBal (Bin s k v l r) = isBal l && isBal r &&
                         (sz l + sz r <= 1 ||
                          sz l <= 3 * sz r <= 3 * sz l)
```

and combine it with an ordering invariant (like `OrdRBT`) to specify valid trees.

```
type OkMap k a = {v : OrdMap k a | isBal v}
```

We can check that `delete` preserves the invariants by checking that its output is an `OkMap k a`. However, we can also go one step further and check the functional correctness property that `delete` removes the given key, with a type:

```
delete :: Ord k => k:k -> m:OkMap k a
       -> {v:OkMap k a | MinusKey v m k}
```

where the predicate `MinusKey` is defined as:

```
predicate MinusKey M1 M2 K
  = keys M1 = difference (keys M2) (singleton K)
```

using the measure `keys` describing the contents of the `Map`:

```
measure keys :: Map k a -> Set k
keys (Tip)             = empty ()
keys (Bin s k v l r)   = union (singleton k)
                               (union (keys l) (keys r))
```

Refocusing XMonad StackSets Our last benchmark comes from the tiling window manager XMonad. The key invariant of XMonad's internal `StackSet` data structure is that the elements (windows) must all be *unique*, *i.e.* contain no duplicates. XMonad comes with a test-suite of over 100 QuickCheck properties; we select one which states that moving the focus between windows in a `StackSet` should not affect the *order* of the windows.

```
prop_focus_left_master n s =
  index (foldr (const focusUp) s [1..n]) == index s
```

With QuickCheck, the user writes a custom generator for valid `StackSet`s and then runs the above function on test inputs created by the generator, to check if in each case, the result of the above is `True`.

With TARGET, it is possible to test such properties *without* requiring custom generators. Instead the user writes a declarative specification:

```
type OkStackSet = {v:StackSet | NoDuplicates v}
```

(We refer the reader to [26] for a full discussion of how to specify `NoDuplicates`). Next, we define a refinement type:

```
type TTrue = {v:Bool | Prop v}
```

that is only inhabited by `True`, and use it to type the QuickCheck property as:

```
prop_focus_left_master :: Nat -> OkStackSet -> TTrue
```

This property is particularly difficult to *verify*; however, TARGET is able to automatically generate valid inputs to *test* that `prop_focus_left_master` always returns `True`.

Summary of Results Figure 8 summarizes the results of the comparison. QuickCheck was unable to successfully complete *any* benchmark to the low probability of generating properly constrained values at random.

List Insert TARGET is able to test `insert` all the way to depth 20, whereas Lazy SmallCheck times out at depth 19.

Red-Black Tree Insert TARGET is able to test `add` up to depth 12, while Lazy SmallCheck times out at depth 5.

Map Delete TARGET is able to check `delete` up to depth 10, whereas Lazy SmallCheck times out at depth 6 if it checks ordering first, or depth 5 if it checks balancedness first.

StackSet Refocus TARGET and is able to check this property up to depth 7, while Lazy SmallCheck times out at the same depth.

TARGET sees a performance hit with properties that require reasoning with the theory of Sets *e.g.* the no-duplicates invariant of `StackSet`. While Lazy SmallCheck times out at a higher depths, when it completes *e.g.* at depth 6, it does so in 0.7s versus TARGET's 9 minutes. We suspect this is because the theory of sets are a relatively recent addition to SMT solvers [18], and with further improvements in SMT technology, these numbers will get significantly better.

Overall, we found that for *small inputs* Lazy SmallCheck is substantially faster as exhaustive enumeration is tractable, and does not incur the overhead of communicating with an external general-purpose solver. Additionally, Lazy SmallCheck benefits from pruning predicates that exploit laziness and only force a small portion of the structure (*e.g.* ordering). However, we found that constraints that force the entire structure (*e.g.* balancedness), or composing predicates in the wrong *order*, can force Lazy SmallCheck to enumerate the entire exponentially growing search space.

TARGET, on the other hand, scales nicely to larger input sizes, allowing systematic and exhaustive testing of larger, more complex inputs. This is because TARGET eschews *explicit* enumeration-and-filtering (which results in searching for fewer needles in larger haystacks as the sizes increas), in favor of *symbolically* searching for valid models via SMT, making TARGET robust to the strictness or ordering of constraints.

5.2 Measuring Code Coverage

The second question we seek to answer is whether TARGET is suitable for testing entire libraries, *i.e.* how much of the program can be automatically exercised using our system? Keeping in mind the well-known issues with treating code coverage as an indication of test-suite quality [16], we ask the reader to consider this experiment as a negative filter.

To this end, we ran TARGET against the entire user-facing API of `Data.Map`, our `RBTree` library, and `XMonad.StackSet` – using the constrained refined types (*e.g.* `OkMap`, `OkRBT`, `OkStackSet`) as the specification for the exposed types – and measured the expression and branch coverage, as reported by `hpc` [11]. We used an increasing timeout ranging from one to thirty minutes per exported function.

Results The results of our experiments are shown in Figure 9. Across all three libraries, TARGET achieved at least 80% expression and alternative coverage at the shortest timeout of one minute per function. Interestingly, `Data.Map` and `RBTree` show no change in coverage metrics beyond a 5 minute timeout, while `XMonad` has another bump in coverage between 10 and 15 minutes.

There are three things to consider when examining these results. First is that some expressions are not evaluated due to Haskell's laziness (*e.g.* the values contained in a `Map`). Second is that some expressions *should not* be evaluated and some branches *should not* be taken, as these only happen when an unexpected error condition is triggered (*i.e.* these expressions should be dead code). TARGET considers any inputs that trigger an uncaught exception a valid counterexample; the pre-conditions should rule out these inputs, and so we expect not to cover those expressions with TARGET.

The last remark is not intrinsically related to TARGET, but rather our means of collecting the coverage data. `hpc` includes **`otherwise`** guards in the "always-true" category, even though they cannot evaluate to anything else. `Data.Map` contained 56 guards, of which 26 were marked "always-true". We manually counted 22 **`otherwise`** guards, the remaining 4 "always-true" guards compared the size of subtrees when rebalancing to determine whether a single or double rotation was needed; we were unable to trigger the double rotation in these cases. `XMonad` contained 9 guards, of which 3 were "always-true". 2 of these were **`otherwise`** guards; the remaining "always-true" guard dynamically checked a function's pre-condition. If the pre-condition check had failed an error would have been thrown by the next case, we consider it a success of TARGET that the error branch was not triggered.

5.3 Limitations of TARGET

Our approach is not without drawbacks. We highlight three classes of pitfalls the user may encounter.

Laziness in the function or in the output refinement can cause exceptions to go unthrown if the output value is not fully demanded. For example, TARGET would decide that the result `[1, undefined]` inhabits `[Int]` but not `[Score]`, as the latter would have to evaluate `0 <= undefined < 100`. This limitation is not specific to our system, rather it is fundamental to any tool that exercises lazy programs. Furthermore, TARGET only generates inductively-defined values, it cannot generate infinite or cyclic structures, nor will the generated values ever contain $\perp$.

Polymorphism Like any other tool that actually runs the function under scrutiny, TARGET can only test monomorphic instantiations of polymorphic functions. For example, when testing `XMonad` we instantiated the "window" parameter to `Char` and all other type parameters to `()`, as the properties we were testing only examined the window. This helped drastically reduce the search space, both for TARGET and SmallCheck.

Our monomorphism restriction simplifies TARGET's implementation as we do not have to consider type-class or equality constraints when generating test values, but it also reduces the generalizability of TARGET's result. Parametricity helps a great deal by telling us that the choice of concrete instantiation will not affect the behavior of the function, but type-classes negate this benefit.

Advanced type-system features such as GADTs and Existential types may prevent GHC from deriving a `Generic` instance, which would force the programmer to write her own `Targetable` instance. Though tedious, the single hand-written instance allows TARGET to automatically generate values satisfying disparate constraints, which is still an improvement over the generate-and-filter approach.

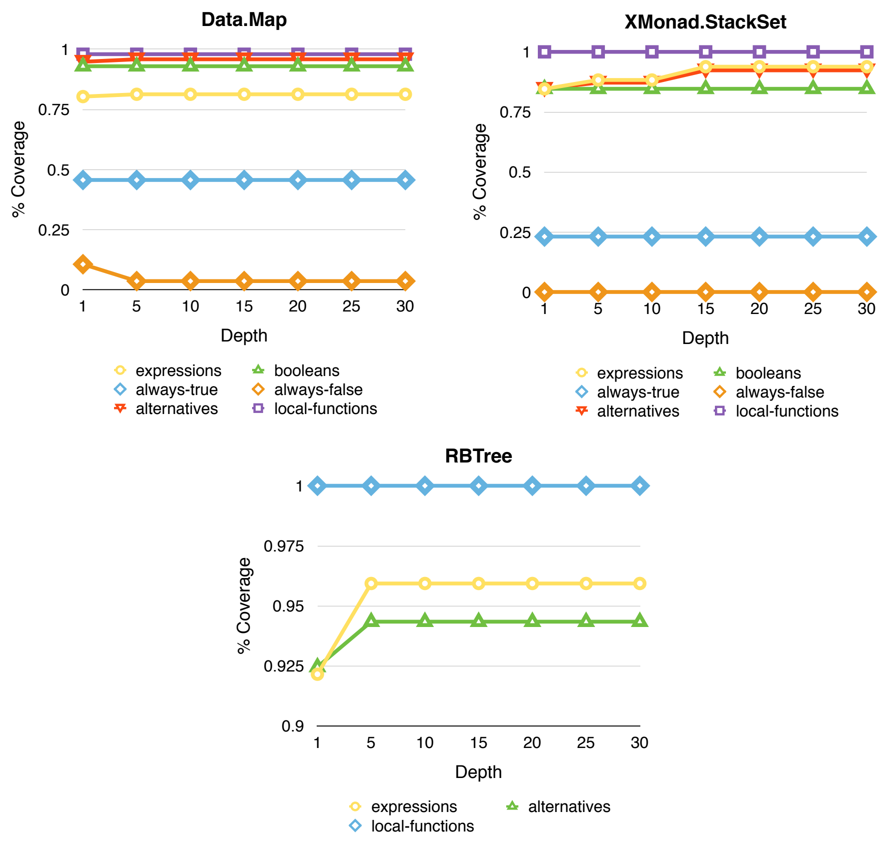

Fig. 9. Coverage-testing of `Data.Map.Base`, `RBTree`, and `XMonad.StackSet` using TARGET. Each exported function was tested with increasing depth limits until a single run hit a timeout ranging from one to thirty minutes. Lower is better for "always-true" and "always-false", higher is better for everything else.

6 Related Work

TARGET is closely related to a number of lines of work on connecting formal specifications, execution, and automated constraint-based testing. Next, we describe the closest lines of work on test-generation and situate them with respect to our approach.

6.1 Model-based Testing

Model-based testing encompasses a broad range of black-box testing tools that facilitate generating concrete test-cases from an abstract model of the system under test. These systems generally (though not necessarily) model the system at a holistic level using state machines to describe the desired behavior [6], and may or may not provide fully automatic test-case generation. In addition to generating test-cases, many model-based testing tools, *e.g.* Spec Explorer [28] will produce extra artifacts like visualizations to help the programmer understand the model. One could view property-based testing, including our system, as a subset of model-based testing focusing on lower-level properties of individual functions (unit-testing), while using the type-structure of the functions under scrutiny to provide fully automatic generation of test-cases.

6.2 Property-based Testing

Many property-based testing tools have been developed to automatically generate test-suites. QuickCheck [4] randomly generates inputs based on the property under scrutiny, but requires custom generators to consistently generate constrained inputs. [3] extends QuickCheck to randomly generate constrained values from a uniform distribution. In contrast SmallCheck [21] enumerates all possible inputs up to some depth, which allows it to check existential properties in addition to universal properties; however, it too has difficulty generating inputs to properties with complex pre-conditions. Lazy SmallCheck [21] addresses the issue of generating constrained inputs by taking advantage of the inherent laziness of the property, generating *partially-defined* values (*i.e.* values containing $\perp$) and only filling in the holes if and when they are demanded. Korat [2] instruments a programmer-supplied `repOk` method, which checks class invariants and method pre-conditions, to monitor which object fields are accessed. The authors observe that unaccessed fields cannot have had an effect on the return value of `repOk` and are thereby able to exclude from the search space any objects that differ only in the values of the unaccessed fields. While Lazy SmallCheck and Korat's reliance on functions in the source language for specifying properties is convenient for the programmer (specification and implementation in the same language), it makes the method less amenable to formal verification, the properties would need to be re-specified in another language that is restricted enough to facilitate verification.

6.3 Symbolic Execution and Model-checking

Another popular technique for automatically generating test-cases is to analyze the source code and attempt to construct inputs that will trigger different paths through the program. DART [12], CUTE [22], and Pex [24] all use a combination of symbolic and dynamic execution to explore different paths through a program. While executing the program they collect *path predicates*, conditions that characterize a path through a program, and at the end of a run they negate the path predicates and query a constraint solver for another assignment of values to program variables. This enables such tools to efficiently explore many different paths through a program, but the technique relies on the path predicates being expressible symbolically. When the predicates are not expressible in the logic of the constraint solver, they fall back to the values produced by the concrete execution, at a severe loss of precision. Instead of trying to trigger all paths through a program, one might simply try to trigger erroneous behavior. Check 'n' Crash [5] uses the ESC/Java analyzer [10] to discover potential bugs and constructs concrete test-cases designed to trigger the bugs, if they exist. Similarly, [1] uses the BLAST model-checker to construct test-cases that bring the program to a state satisfying some user-provided predicate.

In contrast to these approaches, TARGET (and more generally, property-based testing) treats the program as a *black-box* and only requires that the pre- and post-conditions be expressible in the solver's logic. Of course, by expressing specifications in the source language, *e.g.* as contracts, as in PEX [24], one can use symbolic execution to generate tests directly from specifications. One concrete advantage of our approach over the symbolic execution based method of PEX is that the latter generates tests by *explicitly enumerating* paths through the contract code, which suffers from a similar combinatorial problem as SmallCheck and QuickCheck. In contrast, TARGET performs the same search *symbolically* within the SMT engine, which performs better for larger input sizes.

6.4 Integrating Constraint-solving and Execution

TARGET is one of many tools that makes specifications executable via constraint solving. An early example of this approach is TestEra [17] that uses specifications written in the Alloy modeling language [13] to generate all non-isomorphic Java objects that satisfy method pre-conditions and class invariants. As the specifications are written in Alloy, one can use Alloy's SAT-solver based model finding to symbolically enumerate candidate inputs. Check 'n' Crash uses a similar idea, and SMT solvers to generate inputs that satisfy a given JML specification [5]. Recent systems such as SBV [8] and Kaplan [14] offer a monadic API for writing SMT constraints within the program, and use them to synthesize program values at *run-time*. SBV provides a thin DSL over the logics understood by SMT solvers, whereas Kaplan integrates deeply with Scala, allowing the use of user-defined recursive types and functions. Test generation can be viewed as a special case of value-synthesis, and indeed Kaplan has been used to generate test-suites from preconditions in a similar manner to TARGET.

However, in all of the above (and also symbolic execution based methods like PEX or JCrasher), the specifications are *assertions* in the Floyd-Hoare sense. Consequently, the techniques are limited to testing first-order functions over monomorphic data types. In contrast, TARGET shows how to view *types* as executable specifications, which yields several advantages. First, we can use types to compositionally lift specifications about flat values (*e.g.* `Score`) over collections (*e.g.* `[Score]`), without requiring special recursive predicates to describe such collection invariants. Second, the compositional nature of types yields a compositional method for generating tests, allowing us to use type-class machinery to generate tests for richer structures from tests for sub-structures. Third, (refinement) types have proven to be effective for *verifying* correctness properties in modern modern languages that make ubiquitous use of parametric polymorphism and higher order functions [29,7,20,23,26] and thus, we believe TARGET's approach of making refinement types executable is a crucial step towards our goal of enabling *gradual verification* for modern languages.

References

1. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: ICSE 04: Software Engineering. pp. 326–335 (2004)
2. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on Java predicates. In: ISSTA 02: Software Testing and Analysis. pp. 123–133. ACM (2002)

3. Claessen, K., Duregård, J., Palka, M.H.: Generating constrained random data with uniform distribution. pp. 18–34. FLOPS '14 (2014)
4. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of haskell programs. In: ICFP. ACM (2000)
5. Csallner, C., Smaragdakis, Y.: Check 'n' crash: combining static checking and testing. In: ICSE. pp. 422–431 (2005)
6. Dias Neto, A.C., Subramanyan, R., Vieira, M., Travassos, G.H.: A survey on model-based testing approaches: A systematic review. In: WEASELTech '07. ACM (2007)
7. Dunfield, J.: Refined typechecking with Stardust. In: PLPV (2007)
8. Erkök, L.: SBV: SMT based verification in haskell, http://leventerkok.github.io/sbv/
9. Findler, R.B., Felleisen, M.: Contract soundness for object-oriented languages. In: OOPSLA. pp. 1–15 (2001)
10. Flanagan, C., Leino, K., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI (2002)
11. Gill, A., Runciman, C.: Haskell program coverage. pp. 1–12. Haskell '07, ACM (2007)
12. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: PLDI. pp. 213–223 (2005)
13. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology (TOSEM) 11(2), 256–290 (2002)
14. Köksal, A.S., Kuncak, V., Suter, P.: Constraints as control. pp. 151–164. POPL '12, ACM, New York, NY, USA (2012)
15. Magalhães, J.P., Dijkstra, A., Jeuring, J., Löh, A.: A generic deriving mechanism for haskell. In: Haskell Symposium. ACM (2010)
16. Marick, B.: How to misuse code coverage. In: Proceedings of the 16th Interational Conference on Testing Computer Software. pp. 16–18 (1999)
17. Marinov, D., Khurshid, S.: Testera: A novel framework for automated testing of java programs. ASE '01, IEEE Computer Society, Washington, DC, USA (2001)
18. de Moura, L., Bjørner, N.: Generalized, efficient array decision procedures. In: FMCAD (2009)
19. Nelson, G.: Techniques for program verification. Tech. Rep. CSL81-10, Xerox Palo Alto Research Center (1981)
20. Nystrom, N., Saraswat, V.A., Palsberg, J., Grothoff, C.: Constrained types for object-oriented languages. In: OOPSLA. pp. 457–474 (2008)
21. Runciman, C., Naylor, M., Lindblad, F.: Smallcheck and lazy smallcheck: Automatic exhaustive testing for small values. In: Haskell Symposium. ACM (2008)
22. Sen, K., Marinov, D., Agha, G.: Cute: A concolic unit testing engine for c. In: ESEC/FSE. ACM (2005)
23. Swamy, N., Chen, J., Fournet, C., Strub, P.Y., Bhargavan, K., Yang, J.: Secure distributed programming with value-dependent types. In: ICFP (2011)
24. Tillmann, N., Halleux, J.d.: Pex–White box test generation for .NET. In: Tests and Proofs, pp. 134–153 (2008)
25. Vazou, N., Rondon, P., Jhala, R.: Abstract refinement types. In: ESOP (2013)
26. Vazou, N., Seidel, E.L., Jhala, R.: Liquidhaskell: Experience with refinement types in the real world. In: Haskell Symposium (2014)
27. Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Peyton-Jones, S.: Refinement types for haskell. In: ICFP (2014)
28. Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., Nachmanson, L.: Model-based testing of object-oriented reactive systems with spec explorer. In: Formal Methods and Testing. Springer-Verlag (2008)
29. Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. In: PLDI (1998)