Refinement Type Refutations

Refinement types combine SMT decidable constraints with a compositional, syntax-directed type system to provide a convenient way to statically and automatically check properties of programs. However, when type checking fails, programmers must use cryptic error messages that, at best, point out the code location where a subtyping constraint failed to determine the root cause of the failure. In this paper, we introduce *refinement type refutations*, a new approach to explaining why refinement type checking fails, which mirrors the compositional way in which refinement type checking is carried out. First, we show how to systematically transform standard bidirectional type *checking* rules to obtain refutations. Second, we extend the approach to account for global constraint-based refinement *inference* via the notion of a *must-instantiation*: a set of concrete inhabitants of the types of subterms that suffice to demonstrate why typing fails. Third, we implement our method in HayStack—an extension to LiquidHaskell which automatically finds type-refutations when refinement type checking fails, and helps users understand refutations via an interactive user-interface. Finally, we present an empirical evaluation of HayStack using the regression benchmark-set of LiquidHaskell, and the benchmark set of G2, a previous method that searches for (non-compositional) counterexample traces by symbolically executing Haskell source. We show that HayStack can find refutations for 99.7% of benchmarks, including those with complex typing constructs (*e.g.*, abstract and bounded refinements, and reflection), and does so, an order of magnitude faster than G2.

1 INTRODUCTION

Refinement types have proven to be an especially effective way to specify and statically verify properties of programs. The programmer annotates types with logical predicates which define the legal subsets of values for that type's inhabitants. For example, we might specify a type for non-negative integers as `type Nat = Int{v:0 <= v}`. The type-checker then uses an SMT solver to make sure that the restrictions enforced by the types are upheld throughout the program. Consequently, refinement types have been used to specify a variety of correctness properties including data structure invariants [Dunfield 2007; Kawaguchi et al. 2009], security properties [Bengtson et al. 2011; Fournet et al. 2011; Lehmann et al. 2021], hardware ISA specifications [Armstrong et al. 2019], and resource constraints [Knoth et al. 2020]; they were used to verify code developed in Haskell [Vazou et al. 2014], Java [Gamboa et al. 2023] Racket [Kent et al. 2016], Ruby [Kazerounian et al. 2017], Rust [Lehmann et al. 2023], Scala [Hamza et al. 2019], and TypeScript [Vekris et al. 2016].

The unreasonable effectiveness of refinement types stems from their compositional nature. Verification proceeds by exploiting a syntactic discipline to glue together invariants (refinements) of sub-components in a type-directed fashion that scales up to higher-order functions, datatypes and polymorphism. Consequently, type-based compositionality allows for considerable automation: only core library functions need to be annotated and all other refinements can be automatically synthesized by a solver. For example, consider the following Haskell code, that uses a polymorphic `foldr1` function to compute the maximum of a non-empty list of integers.

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 = ...
```

```
max :: Ord a => a -> a -> a
max x y = if x > y then x else y

test1 :: Nat
test1 = foldr1 max [4, 3, 2, 1]
```

A refinement type-checker like LiquidHaskell can automatically verify the above by inferring that `foldr1` is being invoked with **`Nat`** and then exploiting parametricity to conclude that the output must also be at **`Nat`**.

Refinement type-based verification is delightful when it works. When the code is *accepted* by the checker, the programmer can rejoice in the fact that all their specified correctness properties are indeed guaranteed to hold. However, when type checking *fails*, verification turns unpleasant, as the type-checker will often provide cryptic clues as to what went wrong — at best a particular line number where the assertion, precondition or postcondition check failed. This leaves the programmer scratching their heads to work out the root-cause of the typing failure, and what to change to make verification succeed. For example, the following variant of the code above would be *rejected* by the type-checker with the meager feedback that it cannot prove that `test2` is in fact **`Even`**

```
type Even = Int{v:v `mod` 2 = 0}
test2 :: Even
test2 = foldr1 max [4, 3, 2, 1]
```

The delphic feedback from failed typing is not just an implementation issue. Paradoxically, it is a consequence of the compositionality which makes refinement types so effective in the first place! Unlike other (non-compositional) approaches, most notably model checking [Clarke et al. 1999] and symbolic execution [Hallahan et al. 2019; King 1976] we cannot simply produce a *counterexample trace* — a concrete execution that shows why the property fails. Indeed, in `test2` no such trace exists because `test2` evaluates to `4` which is very much **`Even`**. Instead, the key challenge is to devise a way to explain why typing failed that *mirrors* the compositional way in which refinement type checking is carried out in the first place. This challenge is heightened by automation. If the type-checker synthesizes type annotations, our explanation needs to show why *no possible solution* for the yet-to-be-synthesized refinements can make the type checking work.

In this paper, we introduce *refinement type refutations*, a new approach to explaining why refinement type checking failed. We develop our method via four concrete contributions.

1. Type Checking Refutations (§ 3) Our first contribution is the notion of a *refinement type refutation*: a precise formalization of what a counterexample to a typing derivation should look like. Our calculus starts with a standard bidirectional refinement checker [Jhala and Vazou 2021] which has rules for establishing judgments $\Gamma \vdash \mathtt{e} \Leftarrow \mathtt{t}$ — which say that the term `e` can be *checked* to have type `t` in environment Γ — and $\Gamma \vdash \mathtt{e} \Rightarrow \mathtt{t}$ — which say that the type `t` can be *synthesized* for the term `e` in environment Γ. Our first insight is that we can represent refutations as two dual judgments of the form $\Gamma \vdash \mathtt{e} \not\Leftarrow \mathtt{t}$ — which say that the term `e` *cannot* be checked to have type `t` in environment Γ — and $\Gamma \vdash \mathtt{e} \not\Rightarrow !$ — which say that *no type* can be *synthesized* for the term `e` in environment Γ. We then show how to systematically transform the rules that establish checking and synthesis judgments to derive their refutation counterparts, to get a *sound* and *complete* notion of typing refutations.

2. Type Inference Refutations (§ 4) Our second contribution is to extend our notion of refutations to the setting where unknown refinements must be *inferred* via a global solver, *e.g.*, due to parametric

polymorphism, as in the `test1` and `test2` examples above. The key challenge in this setting is to show that *no solution* for the unknown refinements allows the type checking to succeed. Our second insight, is that we can solve the above problem via the notion of a *must-instantiation*: sets of concrete values that *must* inhabit the unknown refinements for the code to type check, and yet, whose presence makes it impossible to prove some verification goal. For example, in `test2` above, the type variable `a` for `foldr1` must be instantiated with the set of integers **`Int`** *containing* `3` — as that value is in the list passed into `foldr1` — and yet, since the output type of `foldr1` says the result can be any of those integers (including `3`) we cannot prove the output is **`Even`**. We show how to extend the refutation rules to formalize the above intuition, by describing how must-instantiations can be *used* to refute verification, and how the instantiations can themselves be *justified* by the constraints imposed by typing.

3. Implementation (§ 5) Our third contribution is an implementation of type refutations in HayStack: an extension to LiquidHaskell which automatically finds refutations when refinement typing fails. HayStack systematically *searches* for must-instantiations by reducing the (Horn) constraints produced by refinement typing into an imperative first-order program with assertions following the recipe of [Jhala et al. 2011]. HayStack then carries out a bounded symbolic execution [King 1976] on those imperative programs to find paths that violate an assertion. Concrete values, derived from SMT models for those paths, can then be mapped back to obtain must-instantiations that yield a refinement type refutation. To help users understand refinement type refutations, we built an interactive front-end called Explorer, which allows the user to interactively expand refinement type refutations to understand how must-instantiations are derived.

4. Evaluation (§ 6) Our final contribution is an evaluation that determines whether HayStack can efficiently find refutations for real-world typing failures. To this end, we run HayStack on two sets of benchmarks. First, we show that on the regression benchmarks of LiquidHaskell comprising about 200 files HayStack successfully finds refutations for 99.2% of the files in on average 3.2 seconds. Second, we compare HayStack against G2 [Hallahan et al. 2019] which produces (non-compositional) counterexamples by performing symbolic execution directly on the Haskell source code to find traces that violate an input type (precondition) or output type (postcondition). By contrast, HayStack directly refutes the typing derivation, and therefore exploits the compositionality of type checking in its search for refutations. Consequently, we show that on the G2 benchmarks, comprising a data set of refinement type errors in a student programming assignment of about 1500 files, HayStack finds refutations for 99.7% of the files an order of magnitude faster than G2 (13.8 vs 145.9 seconds). To assess the quality of counterexamples produced by HayStack, we present a case study (§ 6.1) – an interpreter for a simple arithmetic language, and show how Explorer helps the user understand the counterexample.

2 OVERVIEW

We start by discussing how HayStack and our interactive tool Explorer help users debug verification failures.

2.1 Using HayStack and Explorer

Consider again example `test2` from the introduction. Figure 1 shows LiquidHaskell's output *without* HayStack. The output highlights the line of code that produced the verification failure, but beyond that, the user gains little insight into what went wrong.

```
Liquid Type Mismatch
.
The inferred type
  VV : {v : GHC.Types.Int | $VV##510##k_ /= 0
                          && $VV##510##k_ > 0
                          && $VV##510##k_ ≥ 0
                          && $VV##510##k_ ≥ $lq_anf$##7205759403792803308##d2wA##k_
                          && $VV##510##k_ ≥ $lq_anf$##7205759403792803309##d2wB##k_
                          && $VV##510##k_ ≤ $lq_anf$##7205759403792803302##d2wu##k_
                          && $VV##510##k_ ≤ $lq_anf$##7205759403792803303##d2wv##k_}
.
is not a subtype of the required type
  VV : {VV##564 : GHC.Types.Int | VV##564 mod 2 == 0}
.
Constraint id 2
   |
21 | test2 = foldr1 max $ [4, 3, 2, 1]
   | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Fig. 1. Output of LiquidHaskell *without* HayStack for `test2`.

(a) Root violation with violating instance. (b) Must-instantiation that justifies the instance.

Fig. 2. Output of Explorer using the type-refutation produced by HayStack for `test2`.

By contrast, Figure 2 shows the output of Explorer on the type-refutation produced by HayStack. Figure 2a shows the first screen displayed by Explorer. The red text in the topmost box of the screen tells us that we can derive violating instance `3` for type **Even**. The lower part of the screen gives us an environment binding variables to concrete values that produce the violation.

Clicking on the highlighted variable `self` (the bound variable of the refinement) in the environment produces the screen in Figure 2b, which tells us how the must-instantiation was derived. In particular, the green text in the topmost part of the screen highlights that must-instantiation `3` comes from list `[4, 3, 2, 1]`.

2.2 Refinement Type Checking

Next, we give a high-level overview of our approach to type refutations, using a series of small examples to illustrate refinement types, type checking, and our notion of type refutations.

Refinement Types Say we want to implement a division operation. As division by zero is undefined and will result in an error that we cannot recover from, we want to make sure that we eliminate the possibility of division by zero at compile time. Refinement types allow us to enforce this property by annotating the basic integer type with a predicate that constrains the integer divisor to be not equal to zero. We can then use this definition in the type signature of the division function:

```
type NonZero = Int{v:v /= 0}
div :: x:Int -> y:NonZero -> Int{v:v == x / y}
```

The signature says that the second input to `div` must be non-zero, and that the output is the result of dividing `x` by `y` (*i.e.*, the logical division operation implemented by the SMT solver's decision procedures.) Refinement type checking will ensure that every caller of `div` passes in divisors that can be shown to be non-zero at compile time.

When Typing Goes Wrong Now suppose that we want to use `div` to define a division over natural, *i.e.*, non-negative, integers, defined as:

```
type Nat = Int{v:0 <= v}
```

The denominator argument is converted into a natural by taking its absolute value, via a function `abs :: Int -> Nat`. The division of two natural numbers will give us a new natural number, which is what we state in the return type of `ex0`.

```
ex0 :: Nat -> Int -> Nat
ex0 num den = div num (abs den)
```

Type checking will fail for this code as it is incorrect. However, as the type check will return a non-descript error message, we're left guessing about the cause. In this case, the reader may notice that this is because the absolute value of an integer will give us a natural number. As `0` is a natural, this violates the contract of `div` which constrains its denominator to be non-zero. We might try to fix the program by strengthening the denominator argument to be `NonZero`.

```
ex1 :: Nat -> NonZero -> Nat
ex1 num den = div num (abs den)
```

Perhaps surprisingly, verification of the modified example *still* fails. This time, our specification for `ex1` is correct, yet the verifier cannot verify the implementation. This happens because the type-checker does not have enough information about `abs` to rule out an error. As type checking is *modular*, all the type-checker knows about `abs` is what it says on the tin: *i.e.*, its signature `Int -> Nat`, which says the output could be any non-negative integer. Thus, as far as the type checker is concerned, `abs` *could* be implemented as shown below, which would be consistent with its signature, yet violate the specification of `ex1`.

```
abs :: Int -> Nat
abs _ = 0
```

We want to help the user debug such typing failures. But to do so, we first need to take a closer look at how types are checked in the first place.

Subtyping To validate refinement types, the type-checker will generate *verification conditions* (VCs) corresponding to subtyping constraints. In short, subtyping constraints ensure that the actual expressions used in the program obey the refinements prescribed by the type.

For example, for `ex0`, the refinement checker will try to prove that the actual type `Nat` of the second argument to `div`, that is `(abs den)`, is a subtype of the expected type `NonZero`, producing the following constraint, where $<:$ is the subtyping relation, and Γ denotes an environment mapping variable names to types.

$$\Gamma \vdash \texttt{Nat} <: \texttt{NonZero}$$

Verification Conditions This subtyping constraint requires us to prove that we can use an expression of type `Nat`, whenever we expect an expression of type `NonZero`. Internally, this constraint gets translated into the following *verification condition* (VC) to be proved by an SMT solver.

$$\forall v.\ 0 \leq v \Rightarrow v \neq 0$$

As this implication is *not valid*, the subtyping constraint, and thereby the type-derivation fail. Note that *other* subtyping constraints that need to be checked to establish a positive typing judgement — *e.g.*, the subtyping check $\Gamma \vdash$ `Nat` $<:$ `Int` for the first argument of `div` — are not needed to explain why the function cannot be typed, and hence can be omitted when explaining why typing failed. We formalize this idea via the notion of a *type refutation*.

2.3 Type Checking Refutations

Our formalization of refinement type refutation builds on the standard bidirectional refinement type system proposed in [Jhala and Vazou 2021].

Typing Judgments In this system, type checking is represented by a judgement $\Gamma \vdash \mathtt{e} \Leftarrow \mathtt{t}$ which says that under environment Γ, expression `e` can be *checked* to have type `t`. The type system is bidirectional [Dunfield and Krishnaswami 2021; Pierce and Turner 2000], and therefore makes use of a second judgement $\Gamma \vdash \mathtt{e} \Rightarrow \mathtt{t}$ saying that under environment Γ, we can *synthesize* type `t` for the expression `e`.

Refutation Judgments We represent type refutations via two judgements which intuitively correspond to the refutations of the classical *checking* and *synthesis* judgements. The first *check-refutation* judgment $\Gamma \vdash \mathtt{e} \not\Leftarrow \mathtt{t}$, states that expression `e` *cannot be typed* as `t` under environment Γ. The second *synthesis-refutation* judgement $\Gamma \vdash \mathtt{e} \not\Rightarrow !$, says that under Γ, we cannot synthesize *any* type for expression `e`. The refutation derives non-typing of an expression `e` via the following two steps:

- First, it traverses `e` towards the *faulty expression*: the sub-expression of `e` that is responsible for the typing failure.
- Second, the refutation judgement shows that the faulty expression *violates* a subtyping constraint, which we represent via a non-subtyping judgement $\Gamma \vdash \mathtt{s} \not<: \mathtt{t}$, which says that under environment Γ, type `s` is *not* a subtype of `t`.

Thus, a type refutation captures the core of a violation that we want to show to the user: *what* is the faulty subexpression, and *why* is it faulty. HayStack and its interactive user interface Explorer allow the user to inspect this derivation to help them understand the typing failure.

Type Refutation for `ex1` Let us now look at the type refutation that shows that the definition of `ex1` does not check for its type signature. We specify our notion of type refutations via a set of inference rules, which we will describe in Section 3. For now, we will give an informal overview of how they allow us to derive refutations. We start with the following goal, where we desugar the arguments of `ex1` into lambda abstractions.

$$\Gamma \vdash \texttt{\textbackslash num den -> div num (abs den)} \not\Leftarrow \texttt{Nat -> Int -> Nat}$$

The refutation is made under an environment Γ, which contains declarations of functions that are in scope. In this example, the initial environment contains the declarations of `div` and `abs`.

We start by traversing towards the faulting expression. For this, we apply our rule for lambda expressions ([Ref-Lam]) — the dual of the usual rule to *check* the type of a λ-abstraction — which adds the arguments and their respective types to the environment $\Gamma' \doteq \Gamma$ `; num :: Nat; den :: Int`. With this extended environment, our goal is to prove the inner expression does not type.

$$\Gamma' \vdash \texttt{div num (abs den)} \Leftarrow \texttt{Nat}$$

We notice that the expression cannot be given *any* type (let alone `Nat`), as it violates a subtyping constraint. We therefore use a rule ([Ref-Fail]) that lets us deduce a synthesis-refutation, that is, we show $\Gamma \vdash \texttt{e} \Leftarrow \texttt{t}$ by showing $\Gamma \vdash \texttt{e} \Rightarrow !$.

$$\Gamma' \vdash \texttt{div num (abs den)} \Rightarrow !$$

Next, we locate the cause of error in the function argument, rather than the function body (via rule [Fail-App-Arg]). This rule uses the type synthesis judgement to synthesize a type for the partial application `(div num)` from the environment which yields `NonZero -> Int`, that is, `(div num)` expects a `NonZero` integer. This leads to a new goal which requires us to show that argument `(abs den)` does not satisfy this input constraint.

$$\Gamma' \vdash \texttt{abs den} \Leftarrow \texttt{NonZero}$$

We now have found the faulty subterm and can use a rule ([Ref-Syn]) to show that its inferred type does not match its expected type, that is, $\Gamma \vdash \texttt{e} \Rightarrow \texttt{s}$ but $\Gamma \vdash \texttt{s} \not<: \texttt{t}$. Using the synthesis judgement from the original type-system, we find that `(abs den)` has type `Nat` (the output type of `abs`). We therefore need to show that the inferred type for `(abs den)` — namely `Nat` — *is not a subtype of* `NonZero`.

$$\Gamma' \vdash \texttt{Nat} \not<: \texttt{NonZero}$$

We can prove this statement via a call to the SMT solver which concludes the type refutation. The solver can additionally create a concrete value that refutes the subtyping constraint. In our case, this instance says that `(abs den)` could evaluate to `0`, which indeed describes the root cause of the failure.

Fixing the Violation With this refutation, it becomes clear that verification failed since the type-checker doesn't have enough information about `abs` to rule out the violation. We can fix this problem by helping the verifier with a refinement for `abs` that stipulates the output is non-zero iff the input is non-zero:

```
abs :: x:Int -> Nat{v:x /= 0 <=> v /= 0}
```

With this new type, the program `ex0` is accepted.

2.4 Type Inference Refutations

So far, our verification conditions can be easily checked by an SMT solver, even if the user does not annotate types for subexpressions like partial function applications. However, with the addition of simple conveniences like local variables, branches, recursion, polymorphism and collections, this is no longer possible, as we must first *infer* suitable refinements for various sub-expressions.

For example, consider the polymorphic function `max` shown below, and suppose that we wish to verify that taking the `max` of `2` and an `Even` number will result in an even number.

```
max :: Ord a => a -> a -> a

ex2 :: Even -> Even
ex2 y = max 2 y
```

The type-checker can easily synthesize types for the expression `2` (via the rule that types primitive constants), and `y` (via the rule that looks up the type of the variable bound in the given context)

```
2 :: Int{v:v == 2}
y :: Even
```

However, as `max` is polymorphic, we need a way to appropriately *instantiate* the type variable in its signature with a refinement type that can be ascribed to both the arguments `2` and `y`. One option would be to require a user annotation, but this would be severely hamper usability (owing to the ubiquity of polymorphic instantiation). Instead, refinement type-checkers like LiquidHaskell automatically *infer* suitable refinements for polymorphic instantiation sites by: (1) representing the missing annotations with *template variables* representing the *unknown refinements*, and then (2) computing an assignment from the template variables to concrete refinement predicates that suffice to establish the desired typing judgment. For our example, the type-checker therefore creates a fresh refinement variable κ, to represent the fact that the type parameter of `max` is being instantiated with the (unknown) refinement type `Int{v:κ(v)}`. Thus, at the call-site inside `ex2` `max` is instantiated to the monomorphic type:

```
max @Int{v:κ(v)} :: Int{v:κ(v)} -> Int{v:κ(v)} -> Int{v:κ(v)}
```

Next, the type-checker produces the following subtyping constraints over the unknown refinement κ. The first two constraints come from the arguments `2` and `y` which need to satisfy the constraints placed on the inputs of `max`. The last constraint encodes that the output needs to satisfy the postcondition on `ex2`, that is, `ex2` returns an even number.

```
∅ ⊢ Int{v:v == 2} <: Int{v:κ(v)}
∅ ⊢ Even          <: Int{v:κ(v)}
∅ ⊢ Int{v:κ(v)}   <: Even
```

We cannot query an SMT solver with these constraints, as it does not know how to reason about these template variables. Instead, happily, the constraints correspond to Constrained Horn clauses (CHC), which can be solved by existing solvers [Cosman and Jhala 2017; Grebenshchikov et al. 2012]. For our example, the solver can produce the solution `κ(v) ↦ v `mod` 2 == 0` which satisfies the subtyping constraints and therefore verifies `ex2`.

Type Refutations for Unknown Refinements While refinement inference makes refinement typing more convenient for the programmer, it makes it harder to refute a typing judgement! If our subtyping constraints contain an unknown refinement κ, we need to show *no possible solution* for κ satisfies all subtyping constraints. Consider the following modification of our example above where we try to show that the maximum value of 1 and an integer is even.

```
max :: Ord a => a -> a -> a
ex3 :: Even -> Even
ex3 y = max 1 y
```

Clearly, this program should not type-check as it is incorrect. The type-checker produces the following three subtyping constraints.

$\Gamma \vdash$ **Int**{v:v == 1} <: **Int**{v:κ(v)} (ℓ_1)
$\Gamma \vdash$ **Even** <: **Int**{v:κ(v)} (ℓ_2)
$\Gamma \vdash$ **Int**{v:κ(v)} <: **Even** (ℓ_3)

Refutation via Must-Instantiation Our type-refutation derivation needs to show that the subtyping constraints will fail for any possible choice for unknown refinement κ. Intuitively, each refinement (unknown or not) corresponds to a set of possible values that can inhabit the corresponding refinement type [Constable and Smith 1987; Rushby et al. 1998]. Thus, we generalize the notion of refutations to the setting of unknown refinements, by synthesizing *must-instantiations* — concrete sets of values that some template variables *must* contain — and that suffice to refute some subtyping constraints. For our example, we can see that $\kappa(v)$ must at least contain value `v == 1`, due to the first constraint (ℓ_1). This instantiation is enough to show that the third subtyping constraint (ℓ_3) cannot hold, as the following subtyping constraint is invalid.

$\Gamma \vdash$ **Int**{v:v == 1} <: **Even**

We formalize must-instantiations in two parts: First, we show how they can be *used* to refute subtyping; Second, we show how they can themselves be *justified*, again via subtyping. Let us see how must-instantiations are used and justified in our example, where we begin with the goal of having to refute the type `Even -> Even` for `ex3`.

Using Must-Instantiations To use must-instantiations, we extend our refutation judgements with an environment Σ that contains all must-instantiations needed to refute subtyping constraints. In our example, Σ contains an instantiation of the unknown refinement κ to value 1, that is, we have $\Sigma \doteq [\kappa(v) \mapsto_{\ell_1} v == 1]$. Here label ℓ_1 says that the instance was derived from subtyping constraint ℓ_1. We use labels to ensure that all instances are indeed justified by some subtyping constraint, and none are made up out of thin air. We collect labels via an environment σ. For our example, $\sigma \doteq \ell_1$, that is, our set of labels contains the single location ℓ_1. Now, our goal is to show the following type refutation where the `[Int{⋆}]` denotes that `max` is instantiated to an **Int** with an unknown refinement ⋆:

$\Sigma, \sigma, \Gamma \vdash$ `\y -> max [Int{⋆}] 1 y` $\Leftarrow$ **Even -> Even**

Environment Γ contains the signature of polymorphic function `max`, saying that `max` can be applied for all base-types `a`. As before, we apply our rule for λ-abstractions ([Ref-Lam]), which yields a new environment Γ' containing the input type: $\Gamma' \doteq \Gamma$; `y :: Even`. Our new goal now is to refute that `(max [Int{⋆}] 1 y)` has type **Even**.

$\Sigma, \sigma, \Gamma' \vdash$ `max [Int{⋆}] 1 y` $\Leftarrow$ **Even**

We start by applying our rule ([Ref-Syn]) to indicate we have reached the faulty expression. We now have to show that unknown refinement **Int**$\{v : \kappa(v)\}$ does not subtype **Even**, that is, we get the following non-subtyping obligation.

$\Sigma, \sigma, \Gamma' \vdash$ **Int**{v:κ(v)} $\not<:$ **Even**

We can discharge this obligation through the must-instantiation $\Sigma = \kappa(v) \mapsto_{\ell_1} v == 1$, which says that $\kappa(v)$ must at least contain instance $v == 1$, which violates the restriction from type **Even**. Importantly, the rule checks that $\ell_1 \in \sigma$. This ensures that the instance is indeed justified by one of the subtyping constraints used in the derivation.

Justifying Must-Instantiations The rest of the derivation will now have to justify the presence of the must-instantiation $v == 1$ in $\kappa(v)$, that is, to justify ℓ_1's presence in σ. Our goal is to synthesize type $\kappa(v)$ for `(max [Int{★}] 1 y)` and build up σ as we go. Applying the synthesis judgement of the original type system gives us the following goal, which we can discharge via a rule that allows us to instantiate the unknown refinement of polymorphic function `max` to a fresh unknown refinement $\kappa(v)$.

$$\Sigma, \sigma, \Gamma' \vdash \texttt{max [Int\{★\}] 1} \Rightarrow \texttt{Int\{v:}\kappa(v)\texttt{\} -> Int\{v:}\kappa(v)\texttt{\} -> Int\{v:}\kappa(v)\texttt{\}}$$

This leaves us with the goal to show that we can check argument `1` against $\kappa(v)$, from which we want to derive our instance.

$$\Sigma, \sigma, \Gamma' \vdash \texttt{Int\{v:v == 1\}} <: \texttt{Int\{v:}\kappa(v)\texttt{\}}$$

We can achieve this via our rule ([Sub-KVar]) for deriving must-instantiation. The rule allows us to derive any v satisfying the left hand-side of a subtyping constraint. In our case, this only leaves instance $v == 1$. Importantly, as the corresponding subtyping constraint is labeled with ℓ_1, we can justify ℓ_1's presence in σ. Finally, σ and $\emptyset$ get combined into σ by taking the union of all labels. This completes the refutation.

3 TYPE CHECKING REFUTATIONS

We now formalize the ideas presented so far. We start with the rules for type refutations without unknown refinements and introduce them into the language in Section 4. For now, we mark everything which is not required until Section 4 with a box.

Language Figure 3 shows our target language, which consists of the simply typed lambda calculus. The language is standard. We use integers and Booleans as base types. Base types can be refined with predicates `p` from suitable first-order theories. Importantly, refinements may refer to all function arguments that are in scope. Types are kinded as base for base types and star for everything else. Environments are lists binding variables to types. We will later use type polymorphism, which will require refinement templates. To keep our presentation focused, we omit other language constructs like let-bindings, conditionals, or pattern matching, however our implementation can handle the full Haskell language.

Type Refutation Judgements We base our notion of type refutations on the standard refinement type-system presented in [Jhala and Vazou 2021]. This system consists of two typing judgements: $\Gamma \vdash \texttt{e} \Leftarrow \texttt{t}$ which says that under environment Γ, expression `e` can be *checked* to have type `t`, and judgement $\Gamma \vdash \texttt{e} \Rightarrow \texttt{t}$ which says that under environment Γ, we can *synthesize* type `t` for expression `e`. The main difference between checking and synthesis is that in checking, type `t` is an *input* to the type-system, and in synthesis, `t` is an *output*. Our two type refutation judgments mirror these judgements as follows.

- ***Checking Failures*** are represented by judgements of the shape $\Sigma, \sigma, \Gamma \vdash \texttt{e} \not\Leftarrow \texttt{t}$, and state that we can show that type `t` is not a valid type for expression `e` in context Γ.
- ***Synthesis Failures*** are represented by judgements of shape $\Sigma, \sigma, \Gamma \vdash \texttt{e} \not\Rightarrow !$, and state that we can show that there is no type we could synthesize for `e` under environment Γ.

To refute a typing derivation for the simply typed calculus, we need to identify a faulting expression and traverse towards it. The failure in the faulting expression can either occur due to a type check that isn't met—a checking failure—or due to a synthesized constraint being violated—a synthesis

Terms	e	::=	c	*constants*
		|	x	*variables*
		|	λx.e	*functions*
		|	e x	*application*
		|	e :: t	*type annotation*
		|	$\Lambda\alpha$:k.e	*type abstraction*
		|	e[t]	*type application*
Basic Types	b	::=	Int	*integers*
		|	Bool	*booleans*
		|	α	*type variable*
Refinements	r	::=	{v:p}	*known*
		|	{$\star$}	*hole*
		|	{v:$\kappa(\bar{x})$}	*template variable*
Types	t, s	::=	b{r}	*refined base*
		|	x:s $\rightarrow$ t	*dependent function*
Kinds	k	::=	B	*base kind*
		|	$\star$	*star kind*
Environments	Γ	::=	$\emptyset$	*empty*
		|	Γ; x :: t	*variable binding*
		|	Γ; α :: k	*type variable binding*
	Σ	::=	$\emptyset$	*empty*
		|	$\Sigma; \kappa(\bar{x}) \mapsto_{\ell} \hat{v}$	*template instance*
	σ	::=	$\emptyset$	*empty*
		|	$\sigma; \ell$	*label-set*

Fig. 3. Syntax of Types, Terms, and Environments.

failure. In the process of traversing towards the faulting expression, we build up the necessary type environment for the refutation. Once we have arrived at the faulting expression, we can refute it by showing it does not subtype the goal type. As we will see, both our checking and synthesis failure judgements use the synthesis judgement of the original type-system in [Jhala and Vazou 2021] to find the type of sub-expressions. We will omit its definition as it is standard, however, we provide the full rules in the Appendix.

3.1 Non-Subtyping and Non-Entailment

We start by defining what to do once we have reached the faulty expression. Figure 4 shows our definitions. In this step of the refutation, we want to show that the subtyping constraint belonging to a faulty expression fails. We represent this failure via the judgement $\Gamma \vdash s \not<: t$ which says that s is not a subtype of t under environment Γ. As we have seen in the overview, subtyping constraints are discharged by an SMT solver. We therefore need to define when the SMT constraint corresponding to a subtyping constraint is not valid. We do this via a counterexample judgement $\Gamma \not\vdash c$ which says that a constraint c is not entailed by environment Γ. Rule [Non-Sub-Base] defines non-subtyping for base types. The rule says that for some non-subtyping relation $b\{v_1 : p_1\} \not<: b\{v_2 : p_2\}$ to hold, it must not be the case that any value that satisfies p_1 also satisfies p_2 under environment Γ. Rules

$$\frac{(\kappa(\bar{v}) \mapsto_\ell \hat{v}) \in \Sigma \Rightarrow \ell \in \sigma \qquad \Sigma, \Gamma \nvdash \forall v_1{:}b.p_1 \Rightarrow p_2[v_2 := v_1]}{\Sigma, \sigma, \Gamma \vdash b\{v_1{:}p_1\} \not<: b\{v_2{:}p_2\}} \text{Non-Sub-Base}$$

$$\frac{\Sigma, \sigma, \Gamma \vdash s_2 \not<: s_1}{\Sigma, \sigma, \Gamma \vdash x_1{:}s_1 \to t_1 \not<: x_2{:}s_2 \to t_2} \text{Non-Sub-Contra}$$

$$\frac{\Sigma, \sigma_2, \Gamma \vdash s_2 \prec: s_1 \qquad \Sigma, \sigma_1; \sigma_2, \Gamma; x_2 :: s_2 \vdash t_1[x_1 := x_2] \not<: t_2}{\Sigma, \sigma_1, \Gamma \vdash x_1{:}s_1 \to t_1 \not<: x_2{:}s_2 \to t_2} \text{Non-Sub-Co}$$

$$\frac{\Sigma, \Gamma \nvdash \forall x{:}b.p \Rightarrow c}{\Sigma, \Gamma; x :: b\{x{:}p\} \nvdash c} \text{Cex-Ext} \qquad \frac{\texttt{SmtCex(c)}}{\Sigma, \emptyset \nvdash c} \text{Cex-Emp}$$

$$\frac{\Sigma; \kappa(\bar{x}) \mapsto \hat{y}, \emptyset \nvdash c[\kappa(\bar{x}) := \hat{y}]}{\Sigma; \kappa(\bar{x}) \mapsto \hat{y}, \emptyset \nvdash c} \text{Cex-KVar-Ins}$$

Fig. 4. Rules for Non-subtyping and Non-entailment.

$$\frac{\Sigma, \sigma_2, \Gamma \vdash e \Rightarrow s \qquad \Sigma, \sigma_1; \sigma_2, \Gamma \vdash s \not<: t}{\Sigma, \sigma_1, \Gamma \vdash e \Leftarrow t} \text{Ref-Syn}$$

$$\frac{\Sigma, \sigma, \Gamma \vdash e \Rightarrow !}{\Sigma, \sigma, \Gamma \vdash e \Leftarrow t} \text{Ref-Fail} \qquad \frac{\Sigma, \sigma, \Gamma; x :: s \vdash e \Leftarrow t}{\Sigma, \sigma, \Gamma \vdash \lambda x.e \Leftarrow s \to t} \text{Ref-Lam}$$

$$\frac{\Gamma \vdash s \rhd t \qquad \Gamma \vdash t : k \qquad \Sigma, \sigma, \Gamma \vdash e \Leftarrow t}{\Sigma, \sigma, \Gamma \vdash e :: t \Rightarrow !} \text{Fail-Ann}$$

$$\frac{\Sigma, \sigma_2, \Gamma \vdash e \Rightarrow s \to t \qquad \Sigma, \sigma_1; \sigma_2, \Gamma \vdash y \Leftarrow s}{\Sigma, \sigma_1, \Gamma \vdash e\ y \Rightarrow !} \text{Fail-App-Arg}$$

$$\frac{\Sigma, \sigma, \Gamma \vdash e \Rightarrow !}{\Sigma, \sigma, \Gamma \vdash e\ y \Rightarrow !} \text{Fail-App-Fun}$$

Fig. 5. Rules for Checking and Synthesis Failures.

[Non-Sub-Contra] and [Non-Sub-Co] define non-subtyping over function-types. Non-subtyping over functions is contra-variant over inputs and co-variant over outputs. Both constraints need to hold when we check a function's type. To refute it, it is enough to show that *one* of them is not met. Rule [Non-Sub-Contra] allows us to show that the subtyping relation did not hold because the

input was not contra-variant. Similarly, the rule [Non-Sub-Co] allows us to show that the subtyping relation did not hold because the output was not co-variant.

Next, we define judgement $\Gamma \not\vdash c$ which says that constraint c is not valid under environment Γ. As the SMT solver cannot reason about Γ directly, we need to move assumptions from the environment into the constraint via rule [Cex-Ext]. That is, if for constraint c, the environment contains variable x with refinement p, we extend the constraint to say it must hold for any x satisfying p.

Once the environment is empty, we can apply rule [Cex-Emp] such that an SMT solver may disprove the final formula c by producing a counterexample. Such a counterexample proves that c was not valid, and therefore that the corresponding subtyping constraint failed.

3.2 Checking and Synthesis Failures

To derive a checking failure, we have to traverse to a faulty expression, and then show that the expression violates a subtyping constraint. We formalize this in rule [Ref-Syn], shown in Figure 5. The rule derives a checking failure for type t and expression e by first synthesizing a type s for e and then showing that s doesn't subtype t.

Synthesis Failure If the current expression e is not the faulty expression, and we want to traverse its sub-expressions, we can readily ignore the subtyping constraint between e's synthesized type and t. Instead, our goal becomes to show that we cannot synthesize any type for e as a subtyping constraint is violated in one of its subexpressions. We formalize this reasoning in Rule [Ref-Fail].

Functions Rule [Ref-Lam] shows how to refute typing for functions. We extend the environment with the argument and its respective type. With this, our new goal becomes to produce a checking failure for the function body.

Annotations A type annotation shifts the goal from a synthesis failure to a refutation. Rule [Fail-Ann] in Figure 5 says that to derive a synthesis failure for an expression e that is annotated with a type t, it is enough to show that e fails to check for t. The expression $\Gamma \vdash t : k$ is a well-formedness check on the type annotation, which checks if the type is closed under the environment. Note that our implementation only requires annotations for top level-functions and uses refinement inference to synthesize missing annotations for intermediate terms.

Function Applications Synthesis failure for function applications can occur in one of two positions: The function argument or the function body. The case where the faulty expression resides in the argument of a function is captured by the rule [Fail-App-Arg]. The rule synthesizes a type for function e and shows that argument y violates the input constraint of the function. Conversely, the faulty expression can also reside in the function itself. This case is captured by rule [Fail-App-Fun]. Here, we can discard the argument and show a synthesis failure for the body.

3.3 Example Derivation

We can now revisit example ex1 from Section 2. We want to derive a checking failure for ex1 for type **Nat -> Int -> Nat**. Figure 6 shows the type refutation-tree. We start by applying [Ref-Lam] twice to add the arguments to the environment. The resulting environments are shown below.

```
Γ  ≐ div :: x:Int -> y:NonZero -> Int{v:v == x / y}; abs :: Int -> Nat
Γ′ ≐ Γ; num :: Nat
Γ″ ≐ Γ′; den :: Int
```

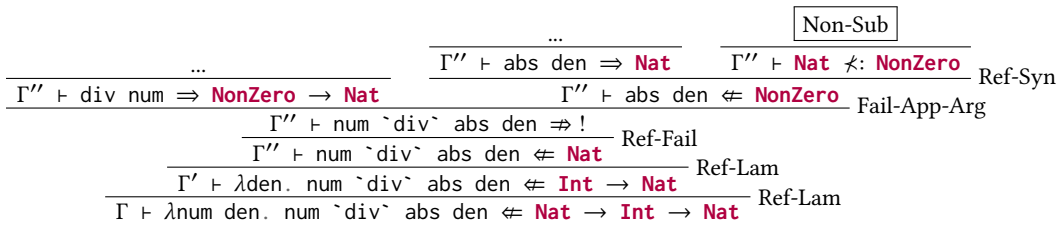
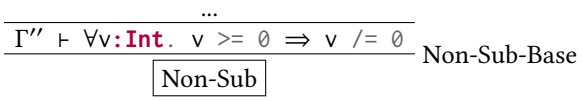


Fig. 6. Type Refutation derivation for example ex1 from the overview.

We end up with the goal of having to show that `(num `div` abs den)` fails to check against type **Nat**. Since the failing expression resides in one of the sub-expressions, we apply rule [Ref-Fail] to instead derive a synthesis failure. Since the faulting expression appears in the argument, we next apply rule [Fail-App-Arg]. We first synthesize type **NonZero -> Nat** for function `(div num)`, and then get a new goal that requires us to show that argument `(abs den)` fails to check under its expected type **NonZero**. We have now reached our faulting expression, and therefore apply rule [Ref-Syn]. We first synthesize type **Nat** for `(abs den)` from the environment and get a new goal that requires us to show that synthesized type **Nat** doesn't subtype checked type **NonZero**. We show the derivation for this goal below.



We apply rule [Non-Sub-Base] which requires us to show that the constraint $\forall v.\ v \geq 0 \Rightarrow v \neq 0$ does not hold under environment Γ''. This is derived by repeated applications of [Cex-Ext] to extend the environment, and finally, rule [Cex-Emp] which discharges the constraint to the solver.

3.4 Soundness and Completeness

We prove soundness and completeness of our refutation judgements with respect to the original judgements from [Jhala and Vazou 2021]. In short, our proof shows that for each original judgement from [Jhala and Vazou 2021], the refutation judgement holds if and only if the original judgment does not hold. For this, we extend the original rules to produce a constraint c, such that the judgement holds if and only if the constraint is valid. We omit the definitions in the interest of space, but include them in the Appendix. Our main theorem is shown below.

Theorem $(\Gamma \vdash e \not\Leftarrow t \iff \Gamma \vdash e \Leftarrow t \cdots c \wedge \not\vDash c) \wedge (\Gamma \vdash e \not\Rightarrow ! \iff \Gamma \vdash e \Rightarrow t \cdots c \wedge \not\vDash c)$

The proof relies on the following main lemma, which states that non-derivation and non-subtyping faithfully encode their positive counterparts.

Lemma $(\Gamma \not\vDash c_1 \iff \Gamma \vdash c_1 \cdots c \wedge \not\vDash c) \wedge (\Gamma \vdash s \not<: t \iff \Gamma \vdash s <: t \cdots c \wedge \not\vDash c)$

4 TYPE INFERENCE REFUTATIONS

We now show how to extend type refutations to include polymorphism and unknown refinements.

Language Consider again Figure 3. We now extend our simply typed lambda calculus with polymorphism. This introduces new terms for type abstraction and type application, as shown in the grayed-out parts of the figure. At the refinement type level, polymorphism requires us to

$$\frac{}{\Gamma \vdash \mathsf{b}\{\star\} \rhd \mathsf{b}\{\mathsf{v}{:}\kappa(\bar{\mathsf{x}})\}}\ \text{Ins-Hole} \qquad \frac{}{\Gamma \vdash \mathsf{b}\{\mathsf{v}{:}\mathsf{p}\} \rhd \mathsf{b}\{\mathsf{v}{:}\mathsf{p}\}}\ \text{Ins-Conc}$$

$$\frac{\Gamma \vdash \mathsf{s}_1 \rhd \mathsf{s}_2 \qquad \Gamma; \mathsf{x} :: \mathsf{s}_2 \vdash \mathsf{t}_1 \rhd \mathsf{t}_2}{\Gamma \vdash \mathsf{s}_1 \to \mathsf{t}_1 \rhd \mathsf{s}_2 \to \mathsf{t}_2}\ \text{Ins-Fun}$$

$$\frac{\Sigma, \sigma, \Gamma; \alpha :: \mathsf{k} \vdash \mathsf{e} \Leftarrow \mathsf{t} \qquad \Gamma \vdash \forall \alpha{:}\mathsf{k}.\mathsf{t} : \star}{\Sigma, \sigma, \Gamma \vdash \Lambda \alpha{:}\mathsf{k}.\mathsf{e} \Leftarrow \forall \alpha{:}\mathsf{k}.\mathsf{t}}\ \text{Ref-TLam}$$

$$\frac{\Sigma, \sigma, \Gamma \vdash \mathsf{e} \Rrightarrow !}{\Sigma, \sigma, \Gamma \vdash \mathsf{e}[\mathsf{t}] \Rrightarrow !}\ \text{Fail-TApp}$$

Fig. 7. Rules for Holes and Templates, and Type Abstractions.

infer solutions for unknown refinements. We extend our language with holes $\star$ which represent unknown refinements in the surface language. The type-checker internally replaces holes with template variables of the form $\{\mathsf{v}{:}\kappa(\bar{\mathsf{x}})\}$, where κ represents the unknown refinement. We extend the environment syntax with bindings for type-variables. Finally, our typing judgements now make use of a template instantiation environment Σ, and label-set σ. The template instantiation environment tracks must-instantiations. Labels track the origin of instantiations to ensure that instantiations are not created out of thin air. We now describe how to modify our previous rules to account for these new features.

4.1 Holes, Templates, and Polymorphism

Template variables are not explicitly written by the programmer. Instead, the type system creates one for every hole $\star$ that is present in the expression. Following [Jhala and Vazou 2021], we define a judgement of the form $\Gamma \vdash \mathsf{s} \rhd \mathsf{t}$ which states that we can instantiate the holes in s with templates, such that t will be the resulting type. The rules are shown in Figure 7. Rule [Ins-Hole] allows us to create a fresh template variable κ in place of a hole. The arguments $\bar{\mathsf{x}}$ of the template variable range over both variable v bound to the current value, as well as all variables in environment Γ. Concrete refinements do not have to be instantiated and are thus left unchanged via rule [Ins-Conc]. Lastly, functions are instantiated by rule [Ins-Fun]. Since refinement types permit us to write dependent function types, we extend the environment of the output type with the type of its argument.

Annotations Since annotations can now contain holes, we have to adjust the annotation rule to instantiate templates for them. This is done via the grayed-out instantiation term in [Fail-Ann] shown in Figure 5. The term creates templates for any holes that reside in the annotation.

Next, we need to define rules for polymorphism. This amounts to defining rules for type application and type abstraction. We show the rules in Figure 7. Rule [Ref-TLam] for type abstractions follows the rule for regular lambdas, however, it allows us to extend the environment with a type instead of a term. Additionally, it introduces a well-formedness constraint, which shows that the type is closed under the environment and all refinements are **Bool**-typed. Rule [Fail-TApp] allows us to discard the type argument, if a synthesis failure occurred in a type application. This is in line with application on terms, where we can discard the argument.

4.2 Must-Instantiations

In order to track must-instantiations, we make use of the template instantiation environment Σ.

$$\frac{\Gamma \vdash \mathsf{b\{v{:}p\}} \prec: \mathsf{b\{v : \kappa(\bar{v})\}} \cdots \mathsf{c} \qquad (\kappa(\bar{x}) \mapsto_\ell \hat{y}) \in \Sigma \Rightarrow (\Sigma, \emptyset \vdash \mathsf{c} \Downarrow \hat{y})}{\Sigma, \ell, \Gamma \vdash \mathsf{b\{v{:}p\}} \prec: \mathsf{b\{v : \kappa(\bar{v})\}}} \text{ Sub-KVar}$$

$$\frac{}{\emptyset \vdash \mathsf{c} \cdots \mathsf{c}} \text{ Ent-Emp} \qquad \frac{\Gamma \vdash \forall \mathsf{x{:}b.p} \Rightarrow \mathsf{c}_1 \cdots \mathsf{c}}{\Gamma; \mathsf{x{:}b\{x{:}p\}} \vdash \mathsf{c}_1 \cdots \mathsf{c}} \text{ Ent-Ext}$$

$$\frac{\Gamma \vdash \forall \mathsf{v}_1{:}\mathsf{b.p}_1 \Rightarrow \kappa(\bar{v})[\mathsf{v}_2 := \mathsf{v}_1] \cdots \mathsf{c}}{\Gamma \vdash \mathsf{b\{v}_1{:}\mathsf{p}_1\} \prec: \mathsf{b\{v}_2{:}\ \kappa(\bar{v})\} \cdots \mathsf{c}} \text{ Sub-Base}$$

Fig. 8. Rules for Subtyping.

Using Must-Instantiations Consider again rule [Non-Sub-Base] from Figure 4. To show that must-instantiations are sound, we attach a label environment σ to all rules. A label ℓ is added whenever the must-instantiations that correspond to this label are checked. These labels are propagated from every subtyping constraint towards the final refutation. In rule [Non-Sub-Base], we check whether every instantiation in Σ indeed has a sound origin ℓ. This ensures that no must-instantiations can be created out of thin air. The template instantiation environment Σ is global, and passed to every rule.

Justifying Must-Instantiations Figure 8 shows our rules for subtyping. We assume that each subtyping constraint is marked with a unique label. For each subtyping constraint where a template variable occurs on the right-hand side, rule [Sub-KVar] checks whether all instances in Σ marked with its label can indeed be derived. We start by using judgement $\Gamma \vdash \mathsf{s} \prec: \mathsf{t} \cdots \mathsf{c}$ to create a constrained Horn clause c that encodes the subtyping constraint. Next, [Sub-KVar] uses clause c to check that every instance $(\kappa(\bar{x}) \mapsto_\ell \hat{y})$ in the must-instantiation set Σ labeled with the current subtyping constraint can indeed be derived. This check is performed using the judgement $\Sigma, \emptyset \vdash \mathsf{c} \Downarrow \hat{y}$, which computes all valid assignments $\hat{y}$ for κ under c and Σ. We will describe this judgement in more detail later. Here, $\hat{y}$ is a concrete instance, where every variable in $\bar{x}$ is mapped to a value. The rule checks that for every must-instantiation $\kappa(\bar{x}) \mapsto_\ell \hat{y}$, instance $\hat{y}$ can indeed be derived. Note that due to recursion, several instantiations can be derived by the same subtyping constraint. Finally, the rule adds label ℓ to the σ to denote that instantiations with this label have a sound origin.

The rules for judgement $\Gamma \vdash \mathsf{s} \prec: \mathsf{t} \cdots \mathsf{c}$ follow the normal judgements for checking entailment, as shown in Figure 4, but extend it to produce a constraint rather than check it. Rule [Sub-Base] turns a subtyping constraint whose head is a template variable into a Horn clause. Rule [Ent-Ext] pushes refinement constraints from the environment to the Horn clause, similar to rule [Cex-Ext] in Figure 4. Finally, rule [Ent-Emp] provides the base-case, which uses the current constraint as the final Horn clause, once the environment is empty, similar to rule [Cex-Emp] in Figure 4.

Propagating Labels We use the synthesis rules of the original type-system from [Jhala and Vazou 2021] like in Section 3. As our typing judgements contain the template instantiation environment Σ and labels set σ, we extend the original rules to propagate these environments. The extension of the original rules is straightforward, and requires conjoining different σ's whenever two goal derivations are required. We give the full rules for synthesis in the supplementary material.

Evaluation Figure 9 contains the set of evaluation rules with which we can check must-instantiations for a template variable. The rules keep track of a variable mapping δ, which contains mappings from variables to concrete values. This mapping is populated whenever we bring a new variable

into scope via rule [Evl-Forall]. Any concrete value w may be chosen here as long as it allows one to prove the constraints in subsequent rules. We can eliminate implications with and without templates via rules [Evl-Body] and [Evl-Assume] respectively. The former requires that the mapping δ is consistent with an instantiation in Σ. The latter that the mapping is a model for the predicate.

When all implications have been resolved, the final rule [Evl-Head] allows us to extract an instantiation for the template variable.

$$\frac{\Sigma, \delta[x := w] \vdash c \Downarrow \hat{v}}{\Sigma, \delta \vdash \forall x.c \Downarrow \hat{v}}\ \text{Evl-Forall} \qquad \frac{\Sigma, \delta \vdash c \Downarrow \hat{v} \qquad \delta \models p}{\Sigma, \delta \vdash p \Rightarrow c \Downarrow \hat{v}}\ \text{Evl-Assume}$$

$$\frac{\Sigma, \delta \vdash c \Downarrow \hat{v} \qquad (\kappa(\bar{x}) \mapsto_{\ell} \delta(\bar{y})) \in \Sigma}{\Sigma, \delta \vdash \kappa(\bar{y}) \Rightarrow c \Downarrow \hat{v}}\ \text{Evl-Body} \qquad \frac{}{\Sigma, \delta \vdash \kappa(\bar{x}) \Downarrow \delta(\bar{x})}\ \text{Evl-Head}$$

Fig. 9. Rules for Evaluation of Horn Clauses.

Instantiation Finally, we can use instances in Σ for our refutation, that is, we use must-instantiations to refute a subtyping constraint on a template variable. Rule [Cex-KVar-Ins] in Figure 4 allows us to substitute a template for its must-instantiation in Σ, and thereby use it in the refutation.

4.3 Example

We will now highlight parts of the derivation of ex3, specifically of the subtyping constraints that allow us to build a refutation. The global template instantiation environment is $\Sigma \doteq \kappa(v) \mapsto_{\ell_1} v = 1$. Figure 11 starts at the subtyping constraint which we want to use for an instantiation on κ. First, we create the Horn clause that is generated from the constraint. We can use the evaluation rules to find that the instantiation $\kappa(v) \mapsto_{\ell_1} v = 1$ is indeed valid under this Horn clause. Figure 10 contains the final refutation. From its label environment, we know that all template instantiations with label ℓ_1 are soundly derived. We check whether all rules in Σ are soundly derived, which is true as it only contains ℓ_1. From here, we continue by checking the constraint. We can expand the template variable inside of the constraint using the template in Σ. An SMT solver finds a counterexample to this constraint, which finishes the refutation.

$$\dfrac{\dfrac{}{\ell_1 \in \ell_1} \qquad \dfrac{\dfrac{\dfrac{}{\mathtt{SmtCex}(\forall v{:}\mathtt{Int}.\ v = 1 \Rightarrow \mathtt{mod}\ v\ 2 = 0)}\ \text{Cex-Emp}}{\Sigma, \emptyset \vdash \forall v{:}\mathtt{Int}.\ v = 1 \Rightarrow \mathtt{mod}\ v\ 2 = 0}\ \text{Cex-KVar-Ins}}{\Sigma, \emptyset \vdash \forall v{:}\mathtt{Int}.\ \kappa(v) \Rightarrow \mathtt{mod}\ v\ 2 = 0}\ \text{Non-Sub-Base}}{\Sigma, \ell_1, \emptyset \vdash \mathtt{Int}\{v{:}\kappa(v)\} \not<: \mathtt{Even}}$$

Fig. 10. Refutation using Σ.

$$\dfrac{\dfrac{}{\Gamma \vdash \mathtt{Int}\{v{:}v = 1\} <: \mathtt{Int}\{v{:}\kappa(v)\} \cdots \forall v{:}\mathtt{Int}.\ v = 1 \Rightarrow \kappa(v)}\ \ldots \qquad \dfrac{\dfrac{\dfrac{}{\Sigma, v \mapsto 1 \vdash \kappa(v) \Downarrow 1} \qquad \dfrac{}{v \mapsto 1 \models v = 1}}{\Sigma, v \mapsto 1 \vdash v = 1 \Rightarrow \kappa(v) \Downarrow 1}\ \text{Evl-Assume}}{\Sigma, \emptyset \vdash \forall v{:}\mathtt{Int}.\ v = 1 \Rightarrow \kappa(v) \Downarrow 1}\ \text{Evl-Forall}}{\Sigma, \ell_1, \Gamma \vdash \mathtt{Int}\{v{:}v = 1\} <: \mathtt{Int}\{v{:}\kappa(v)\}}\ \text{Sub-KVar}$$

Fig. 11. Check of all template instances with label ℓ_1.

```
fn main() {
    let self: int;
    call $k(self);
    assert self mod 2 = 0;
}
```

```
fn $k(self: int) {
    assume self = 1;
||
    assume self mod 2 = 0;
}
```

Fig. 12. Program transformation for constraints of ex3.

5 IMPLEMENTATION

To find refinement refutations, we have to find a suitable set of must-instantiations that satisfy the constraints in Section 4.2. We have implemented the search for must-instantiations in a tool called HayStack that builds on the LiquidHaskell theorem prover. HayStack is implemented in around 1500 lines of Haskell code. We keep the size of our implementation low by reusing LiquidHaskell's code wherever possible. For example, subtyping constraints are identical for both refinement types and refutations, which allows reuse of LiquidHaskell's code for transforming program expressions to constraints.

Search Strategy At high-level, our search uses the following strategy. First, HayStack translates the subtyping constraints created by LiquidHaskell into a non-deterministic program, following [Jhala et al. 2011], and then symbolically executes the program. Here, assertions in the program encode the restrictions imposed by the subtyping constraints. Whenever we find a path that violates an assertion, we can use the SMT solver to produce a concrete model. This yields a set of concrete instances for each unknown refinement, which refute the subtyping constraints, and which we can therefore use as must-instantiations to populate Σ.

Example Consider again example ex3 in Section 2.4. Type-checking produces the following subtyping constraints.

$$\Gamma \vdash \texttt{Int\{v:v == 1\}} \prec: \texttt{Int\{v:}\kappa\texttt{(v)\}} \quad (\ell_1)$$
$$\Gamma \vdash \texttt{Even} \prec: \texttt{Int\{v:}\kappa\texttt{(v)\}} \quad (\ell_2)$$
$$\Gamma \vdash \texttt{Int\{v:}\kappa\texttt{(v)\}} \prec: \texttt{Even} \quad (\ell_3)$$

HayStack transforms these constraints into the program shown in Figure 12. Constraint ℓ_3 gets translated into the function's main routine. The program creates a fresh variable self for the bound value of the refinement v, and transforms the unknown refinement κ into a function call to function k. This function call explores possible values of κ, as described by the other subtyping constraints. Upon return, the program checks that the value of self is indeed even. The definition of function k offers two possible execution paths, derived from constraint ℓ_1 and ℓ_2, respectively. We use || to denote non-deterministic choice. The first path sets self to 1, according to constraints ℓ_1, and the second path sets self to an even number, following constraint ℓ_2.

Program Transformation More generally, any subtyping constraint whose right-hand side is a template variable is expanded as a possible body for the corresponding function. The remaining constraints — those without a template variable on the right-hand side — are added to the main function, which contains all bodies that could produce a violation. As such, every constraint produces a single function body. This makes it easy to track labels, *i.e.*, which constraint produced which must-instantiation. The translation of a constraint into a function body starts with a declaration of the entire environment. self is the variable whose subtyping relation is checked and

it is thus always contained by a function body. After having declared all variables, the function assumes the refinements placed on each variable. Template variables form a special case, as they are converted into a call instead. Any constraint in the `main` function will additionally assert its required refinement, *i.e.*, the final constraint placed on `self`.

Search Once the program is generated, HaySTack symbolically executes it. Execution starts with the bodies of the main function and recursively follows function calls, while enumerating different paths. HaySTack only executes the bodies of subtyping constraints which were violated in the original type-checking attempt by LiquidHaskell. Exploration stops once a satisfying assignment (a counterexample) has been found. We use depth-first search, which allows us to benefit from the SMT solver's incremental solving capabilities to maintain solver state for execution paths that share a common prefix. Since subtyping constraints may produce recursive programs, we bound the number of times each recursive function call may be unfolded via a parameter **d**. If the search succeeds within the bound, we can conclude that a refutation exists. If the search fails, it could either be the case that a refutation could be found with a larger bound, or that in fact no refutation exists, as a different solution for the unknown refinements would complete the proof. In this case, we substitute the recursive instance with the original solution for the unknown refinement, as computed by LiquidHaskell to receive a *partial counterexample*. We clearly mark those cases, so a user can distinguish them from a full refutation. While partial counterexamples do not yield a full refutation to the typing derivation, we found that they can often still be useful in determining the root cause of the verification failure. To speed up the search, we apply an optimization that lets us avoid exploring paths, if the (negation of the) top-level assertion we want to check is already unsatisfiable. In this case, further exploration would only add additional assumptions which would constrain the search even more, and therefore, we can soundly avoid exploring the path further.

Example Symbolically executing the program in Figure 12 produces two paths — one for each function body of `k`. The path corresponding to the upper function body violates the assertion in `main`, and we query the SMT solver for a counterexample model, which yields `self`= 1. This yields the must-instantiation needed to refute the typing judgement. In particular, we can use the must-instantiation to guide the instantiation choice in rule [Evl-Forall] in Figure 9.

Counterexample Object Once a satisfying assignment has been found, HaySTack extracts it into a JSON object. The object keeps track of must-instantiations, concrete instances for environment variables, as well as source locations and code-spans for all objects. This information is enough to reconstruct a full type refutation. Must-instantiations form a valid environment Σ that lets us discharge rules [Non-Sub-Base] from Figure 4 and [Sub-KVar] in Figure 8. Concrete instances for environment variables let us discharge rule [Cex-Emp] in Figure 4. Source spans tell us which subterms contributed to a derivation. Once the object is compiled, we pass it to our interactive exploration tool Explorer.

Explorer To make it easier to navigate the counterexample, we have implemented a front-end tool called Explorer. Explorer is implemented in about 600 lines of Rust and provides a visual interface into the object. It lets the user interactively explore the type refutation by unfolding instantiations for unknown template variables, as described in Figure 2.

Usage HaySTack forms an extension to LiquidHaskell which is enabled via its feature flag `--counter-examples`. With this, HaySTack automatically searches for a refutation whenever LiquidHaskell cannot verify a constraint, emitting a concrete counterexample JSON object, if successful. Explorer is a stand-alone program that can be used to view the emitted JSON file(s).

Benchmark	Avg. LOC	# Files	Avg. Runtime		Accuracy			
			HayStack	G2	HayStack			G2
					Full	Partial	Total	
k-means	204	295	51.34 s	386.97 s	63.52 %	35.48 %	99.00 %	98.5 %
list	228	582	1.85 s	80.7 s	98.50 %	1.47 %	99.97 %	97.7 %
map-reduce	83	172	1.28 s	26.12 s	100 %	0 %	100 %	96.6 %
lh-regression	31	191	3.25 s	N/A	96.60 %	2.64 %	99.25 %	N/A

Fig. 13. Average runtime and success rate of HayStack and G2.

6 EVALUATION

Benchmarks We evaluate our implementation against two datasets. First, the dataset of lazy counterfactual symbolic execution [Hallahan et al. 2019], against which their implementation G2 was tested. Second, the regression test suite of LiquidHaskell, which contains a wider variety of refinements that G2 is not compatible with. In their paper, G2 measured runtime of their system against a set of programs written by students as part of a homework assignment. Their assignment was to verify a number of functions using LiquidHaskell, which were spread over 3 programs. The final corpus contains a log of every failed verification attempt. In total, this is a set of 1240 incorrect programs. We additionally measure our implementation against the LiquidHaskell regression set. This test suite is a set of 191 incorrect programs, which range over a more diverse set of refinement type constructs compared to the G2 benchmark. It encompasses several simple checks, such as out-of-bounds accesses on vectors, uniqueness of elements in collections and refinements over elements inside of collections. The regression test-suite also contains refinements that are not handled by G2, for example, abstract refinements, bounded refinements, and refinement reflections. We use $\mathbf{d} = 2$ for all our experiments.

Results Figure 13 compares the runtime and accuracy of HayStack against those of G2. HayStack's runtime includes LiquidHaskell's type checking and HayStack's search for a type refutation. The results show that HayStack finds counterexamples for 99.7% of all benchmarks, and does so an order of magnitude faster than G2. We believe that this can be attributed to the difference in approach. G2's notion of counterexamples is based on traces, which requires enumerating program runs via symbolic execution. By contrast, HayStack directly refutes the typing derivation, and can therefore benefit from the abstractions and modularity the type-system imposes. In Figure 14, we present a detailed overview of the distribution of runtimes for each file. Here, we find a bimodal distribution between runs. Most runs are fast, however some runs — in particular for our most complicated benchmark k-means — are slow. As a fast feedback loop is important when fixing verification errors, HayStack users can often benefit from the improved speed: most counterexamples are found within timeframes similar to those required to do regular verification with refinement times. This radically improves practicality of verification

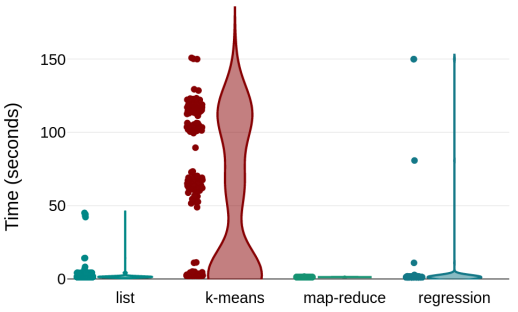


Fig. 14. Runtime per file of HayStack.

with user feedback. In the cases where search times are not quick, HayStack still outperforms G2. We believe that these numbers could be further improved with better search heuristics.

HayStack's accuracy is higher than that of G2; It produces counterexamples for 99.7% of the benchmarks in G2's suite, 10% of which are partial. HayStack can also find counterexamples for benchmarks with advanced features like abstract refinements, bounded refinements, and refinement reflections, which are outside of G2's scope.

Counterexample Quality Manual inspection shows that HayStack's counterexamples provide useful insights into the errors found in the benchmark. For example, the list benchmark contains an assertion checking whether concatenating a list of lists produces a list whose length is the sum of those of its inputs. Here, HayStack generates a counterexample with an emtpy output list and non-empty inputs. This gives us a valuable clue: the function is missing a specification that relates the lengths of its inputs to that of its output. Partial counterexamples also prove useful. A violation that took particularly long to find in the k-means benchmark is in fact a simple division by zero. The problem: division is called via a long chain of unannotated functions. Though large, the final counterexample is easy to understand. If we expand `self` until it reaches a concrete instance for the argument, we find the root cause to be an integer that was unconstrained by a refinement and that HayStack could thus instantiate with `0`.

6.1 Case Study

We present a case study that shows how HayStack and Explorer help debug verfication failures.

6.2 Expression Evaluator

Our case-study is a simple expression evaluator. An expression is recursively defined as a number, a variable, an addition, or a let definition. Instead of giving names to variables, we identify them by a natural number. In particular, variable `0` is the most recently let bound variable; variable `1` the one bound before that, and so on. This construction is similar to de Bruijn indices. We parametrize the data-structure by a type-variable `a`, which represents the type of values our expressions can take. The following definition captures this description.

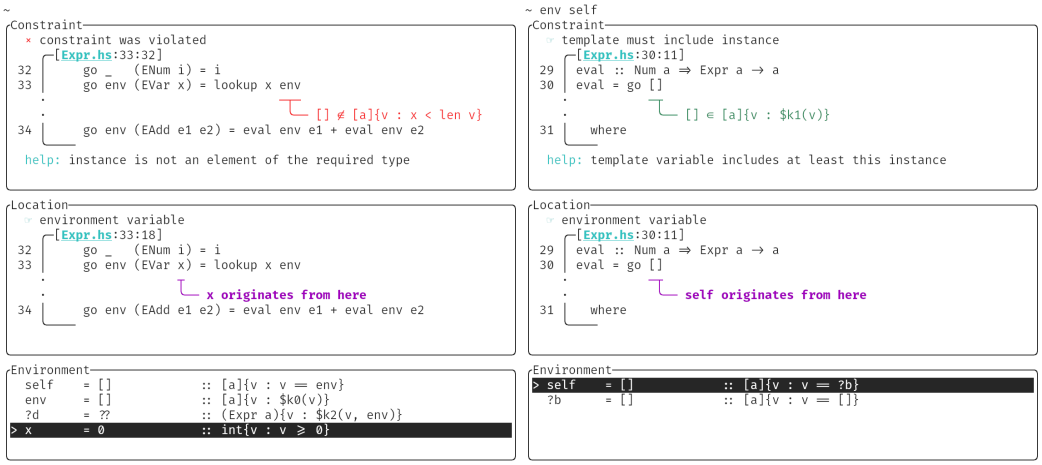
```
data Expr a
= ENum a
  | EVar Nat
  | EAdd (Expr a) (Expr a)
  | ELet (Expr a) (Expr a)
```

Next, we define an evaluator for these expressions. Our evaluator maintains an environment in the form of a list of values, which can be indexed by the identifier of an `EVar`. We use function `lookup` to map variables to their value. The type of `lookup` ensures that the given identifier is indeed a valid index into the list.

```
lookup :: x:Nat -> [a]{v:x < len v} -> a
```

With this, we can define the evaluator. Helper function `go` recursively traverses the expression. The evaluation of an expression starts with an empty environment.

```
eval :: Num a => Expr a -> a
eval = go []
```



(a) The root violation. (b) Unfolding of must-instantiation.

Fig. 15. A counterexample of `eval` as displayed by Explorer.

```
  where
    go _   (ENum i) = i
    go env (EVar x) = lookup x env
    go env (EAdd e1 e2) = eval env e1 + eval env e2
    go env (ELet e1 e2) = eval (eval env e1:env) e2
```

Unfortunately, our definition of `eval` is incorrect as it can produce out of bounds accesses to the environment. An expression can only be evaluated if it is closed, that is, if all variables are bound by a let expression.

HayStack locates the problem to `lookup`, specifically, it pinpoints the input `env` as violating its contract. We can then use Explorer to help us identify concrete instances for this violation.

Figure 15 shows two screenshots of the counterexample for `eval`, as produced by Explorer. The left screenshot in Figure 15a shows the top-level counterexample screen, which tells us that the empty list `[]` is not a list whose length is strictly larger than `x`. In the bottom of the screen, Explorer displays the current environment, which tells us that `x` is instantiated with `0`. Explorer also tells us which constraints were placed on `x`, here that `x` is a natural number. By clicking on `env`, we can get to the screen shown in Figure 15b. This screen displays an expansion of the template variable on `env`, which shows us the origin of must-instantiation `[]`. Using the counterexample, it becomes clear that `eval` should only accept closed expressions, which we can specify via a refinement.

7 RELATED WORK

Counterfactual Symbolic Execution Closest to our work is [Hallahan et al. 2019], which shares the goal of providing counterexamples for programs with refinement types. In addition to counterexamples due to errors in the code, [Hallahan et al. 2019] defines a notion of counterfactual evaluation, which allows them to give counterexamples for verification failures that stem from imprecise specifications, like `ex1` from Section 2. In contrast to our work, counterexamples are defined in a non-compositional fashion, in terms of program traces, rather than compositionally, in terms

of the type-derivation itself. This has several implications. First, finding counterexamples requires modeling the source semantics of the original language which, for Haskell, is non-trivial due to lazy evaluation. By contrast, refinement type refutations are directly expressed over the subtyping constraints and are thus, source language agnostic. Second, modeling counterexamples in terms of program traces requires explicit enumeration of program runs via symbolic execution, which can be slow. By contrast, HaySTack directly refutes the type derivation, and can therefore benefit from the abstraction the type system naturally offers. This advantage can be seen in our experimental evaluation, where HaySTack outperforms [Hallahan et al. 2019] by an order of magnitude.

Counterexamples for Deductive Program Verifiers Deductive verifiers often offer little support to users when verification goes wrong. F* [Swamy et al. 2011] and Low* [Protzenko et al. 2017] generate counterexamples from SMT queries, but since these queries often involve quantified formulas, the resulting counterexamples are hard to interpret. It's also often not clear how to map counterexamples which represent concrete valuations that refute SMT queries stemming from verification conditions back to the original typing judgements. By contrast, refinement type refutations are not expressed in terms of verification conditions and directly refute the type derivation itself. There have recently been efforts to improve Dafny's [Leino 2010] support for counterexamples. [Christakis et al. 2016] presents an IDE that helps debug Dafny errors by finding counterexamples via symbolic execution. More recently, [Chakarov et al. 2022] proposed a method to remove irrelevant information from counterexamples to make them more user-friendly. However, Dafny is only concerned with imperative programs. Push-button verifiers [Birgmeier et al. 2014; Clarke et al. 1999; Henzinger et al. 2002] often produce counterexamples, however, these counterexamples are non-modular, often large and represent entire execution traces over the full program state. Moreover, these tools are hard to apply in a functional setting. By contrast, HaySTack produces small counterexamples as it benefits from the abstraction and modularity that is naturally introduced by the refinement type system. As future work, we plan to explore how refinement-type refutations can help find counterexamples in languages like F* and Low*.

Randomized Testing Quickcheck [Claessen and Hughes 2000] and SmallCheck [Runciman et al. 2008] use randomized testing to find violations to user-provided properties. [Seidel et al. 2015] uses bounded, exhaustive testing to validate specification extracted from refinement-type annotations. Similarly, QuickChick [qui 2018] uses randomized testing to debug Coq proofs. [Zhou et al. 2023] presents a type-system that ensures completeness of checking by making sure the random test generators can generate all elements that satisfy a function's input type. While these techniques can help pinpoint assertion violations, they cannot debug verification failures that result from imprecise specifications, such as ex1.

Localizing Errors in Hindley-Milner Style Type Systems [Seidel et al. 2016] uses symbolic execution to find concrete witnesses that explain and visualize typing failures to programming novices, however, the work focuses on Hindley-Milner style type systems, rather than refinement types. Rite [Sakkas et al. 2020] uses a data-driven method to predict possible repairs for type-errors to help novice-users debug them. Unlike HaySTack, these techniques are concerned with the base type-system, whereas we focus on violations of subtyping constraints between refinements.

Incorrectness Logic Incorrectness logic [Le et al. 2022; O'Hearn 2019; Raad et al. 2022], is dual to Hoare logic in that it allows reasoning about program failures by tracking under-approximations of reachable state. This is similar to our notion of must-instantiations, which can also be seen as underapproximate solutions to the underlying subtyping constraints.

REFERENCES

2018. *QuickChick: Property-Based Testing In Coq.* Leonidas Lampropoulos and Benjamin C. Pierce.

Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proc. ACM Program. Lang.* 3, POPL, Article 71 (jan 2019), 31 pages. https://doi.org/10.1145/3290384

J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. 2011. Refinement types for secure implementations. *ACM TOPLAS* (2011).

Johannes Birgmeier, Aaron R. Bradley, and Georg Weissenbacher. 2014. Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR). In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 831–848. https://doi.org/10.1007/978-3-319-08867-9_55

Aleksandar Chakarov, Aleksandr Fedchin, Zvonimir Rakamaric, and Neha Rungta. 2022. Better counterexamples for Dafny. In *TACAS 2022*. https://www.amazon.science/publications/better-counterexamples-for-dafny

Maria Christakis, K. Rustan Leino, Peter Müller, and Valentin Wüstholz. 2016. Integrated Environment for Diagnosing Verification Errors. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636*. Springer-Verlag, Berlin, Heidelberg, 424–441. https://doi.org/10.1007/978-3-662-49674-9_25

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

Edmund M Clarke, Orna Grumberg, and Doron A. Peled. 1999. *Model checking*. MIT Press, London, Cambridge.

R. L. Constable and S. F. Smith. 1987. Partial Objects In Constructive Type Theory. In *LICS*.

Benjamin Cosman and Ranjit Jhala. 2017. Local Refinement Typing. *Proc. ACM Program. Lang.* 1, ICFP, Article 26 (aug 2017), 27 pages. https://doi.org/10.1145/3110270

J. Dunfield. 2007. Refined typechecking with Stardust. In *PLPV*.

Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *Comput. Surveys* 54, 5 (may 2021), 1–38. https://doi.org/10.1145/3450952

C. Fournet, M. Kohlweiss, and P-Y. Strub. 2011. Modular code-based cryptographic verification. In *CCS*.

Catarina Gamboa, Paulo Canelas, Christopher Steven Timperley, and Alcides Fonseca. 2023. Usability-Oriented Design of Liquid Types for Java. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1520–1532. https://doi.org/10.1109/ICSE48619.2023.00132

Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. 2012. Synthesizing Software Verifiers from Proof Rules. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 405–416. https://doi.org/10.1145/2254064.2254112

William T. Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. 2019. Lazy Counterfactual Symbolic Execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 411–424. https://doi.org/10.1145/3314221.3314618

Jad Hamza, Nicolas Voirol, and Viktor Kuncak. 2019. System FR: formalized foundations for the stainless verifier. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 166:1–166:30. https://doi.org/10.1145/3360592

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2002. Lazy Abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon) *(POPL '02)*. Association for Computing Machinery, New York, NY, USA, 58–70. https://doi.org/10.1145/503272.503279

Ranjit Jhala, Rupak Majumdar, and Andrey Rybalchenko. 2011. HMC: Verifying Functional Programs Using Abstract Interpreters. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 470–485. https://doi.org/10.1007/978-3-642-22110-1_38

Ranjit Jhala and Niki Vazou. 2021. Refinement Types: A Tutorial. *Foundations and Trends® in Programming Languages* 6, 3–4 (2021), 159–317. https://doi.org/10.1561/2500000032

M. Kawaguchi, P. Rondon, and R. Jhala. 2009. Type-based Data Structure Verification. In *PLDI*.

Milod Kazerounian, Niki Vazou, Austin Bourgerie, Jeffrey S. Foster, and Emina Torlak. 2017. Refinement Types for Ruby. *CoRR* abs/1711.09281 (2017). arXiv:1711.09281 http://arxiv.org/abs/1711.09281

A. M. Kent, D. Kempe, and S. Tobin-Hochstadt. 2016. Occurrence typing modulo theories. In *PLDI*.

J.C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19(7) (1976), 385–394.

Tristan Knoth, Di Wang, Adam Reynolds, Jan Hoffmann, and Nadia Polikarpova. 2020. Liquid resource types. *Proc. ACM Program. Lang.* 4, ICFP (2020), 106:1–106:29. https://doi.org/10.1145/3408988

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, OOPSLA1, Article 81 (apr 2022), 27 pages. https://doi.org/10.1145/3527325

Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid Types for Rust. *Proc. ACM Program. Lang.* 7, PLDI, Article 169 (jun 2023), 25 pages. https://doi.org/10.1145/3591283

Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan, and Ranjit Jhala. 2021. STORM: Refinement Types for Secure Web Applications. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 441–459. https://www.usenix.org/conference/osdi21/presentation/lehmann

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (Dakar, Senegal) *(LPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 348–370.

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

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (jan 2000), 1–44. https://doi.org/10.1145/345099.345100

Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified Low-Level Programming Embedded in F*. *Proc. ACM Program. Lang.* 1, ICFP, Article 17 (aug 2017), 29 pages. https://doi.org/10.1145/3110261

Azalea Raad, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Concurrent incorrectness separation logic. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. https://doi.org/10.1145/3498695

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell* (Victoria, BC, Canada) *(Haskell '08)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/1411286.1411292

J. Rushby, S. Owre, and N. Shankar. 1998. Subtypes for specifications: predicate subtyping in PVS. *IEEE Transactions on Software Engineering* 24, 9 (1998), 709–720. https://doi.org/10.1109/32.713327

Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, and Ranjit Jhala. 2020. Type Error Feedback via Analytic Program Repair. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 16–30. https://doi.org/10.1145/3385412.3386005

Eric L. Seidel, Ranjit Jhala, and Westley Weimer. 2016. Dynamic Witnesses for Static Type Errors (or, Ill-Typed Programs Usually Go Wrong). In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 228–242. https://doi.org/10.1145/2951913.2951915

Eric L. Seidel, Niki Vazou, and Ranjit Jhala. 2015. Type Targeted Testing. In *Proceedings of the 24th European Symposium on Programming on Programming Languages and Systems - Volume 9032*. Springer-Verlag, Berlin, Heidelberg, 812–836. https://doi.org/10.1007/978-3-662-46669-8_33

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure Distributed Programming with Value-Dependent Types. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. Association for Computing Machinery, New York, NY, USA, 266–278. https://doi.org/10.1145/2034773.2034811

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161

P. Vekris, B. Cosman, and R. Jhala. 2016. Refinement types for TypeScript. In *PLDI*.

Zhe Zhou, Ashish Mishra, Benjamin Delaware, and Suresh Jagannathan. 2023. Covering All the Bases: Type-Based Verification of Test Input Generators. *Proc. ACM Program. Lang.* 7, PLDI, Article 157 (jun 2023), 24 pages. https://doi.org/10.1145/3591271