Proof Axiom Strengthening via Interactive Verification

Kevin Yu
Purdue University
Department of Computer Science
West Lafayette, Indiana, US

Abstract

There has been recent research into "coverage" types, which are refinement types with special semantics that allow type systems to validate *coverage* of e.g. input generators. When developing refinement type systems, developers often use automated theorem provers to check that typing obligations are satisfied. To allow solvers to reason about inductive data types, language developers define type predicates and impose meaning on those predicates via axioms passed to the solver.

However, developing axioms is error-prone and time consuming; presented with an opaque solver error, it can be hard for users to ascertain if additional axioms are needed to discharge the query. To address this, we have implemented a technique of translating failed verification queries to an interactive theorem prover (Rocq) allowing users to inspect the cause of the failure. In the interactive setting, users can use provided definitions of predicates to prove that added axioms are valid, ensuring the language's type system remains sound. To demonstrate generalization to real-world use cases, we apply this system to test cases from a typechecker, Cobb.

1 Introduction

A refinement type is a base type T qualified by some constraint ϕ, denoted

$$\{v : T \mid \phi\}$$

For example, we can define the type of non-negative integers as

$$\{v : \text{int} \mid v \geq 0\}$$

Formally, an expression has a refinement type $\{v : T \mid \phi\}$ if it always evaluates to a value satisfying ϕ. Thus,

$$\vdash 0 : \{v : \text{int} \mid v \geq 0\}$$

but

$$\nvdash -4 : \{v : \text{int} \mid v \geq 0\}$$

There has also been recent research into so-called *coverage* (or "underapproximate") refinement types, denoted by

$$[v : T \mid \phi]$$

whose semantics require that the set of values that the expression can evaluate to must fully contain, or *cover*, the underapproximate refinement type [Zhou et al. 2023]. These types allow program writers to perform "must-style" analysis: useful, for example, for enforcing at a type level that a generator function truly can generate all of a range of values.

To examine semantics of coverage types, Zhou et al. define a language and corresponding typechecker (Poirot) equipped with a type system supporting over- and underapproximate refinement types.

2 Motivation

In order to typecheck a Poirot program, the typechecker passes typechecking queries to an SMT solver (here, Z3) in order to verify that type constraints are satisfied.

For an example of the role of Z3 in the Poirot typechecker, consider the refinement type subtyping query

$$\{v : T \mid \phi_1\} <: \{v : T \mid \phi_2\}$$

(i.e. we want to claim that values of type $\{v : T \mid \phi_1\}$ can be used in place of values of type $\{v : T \mid \phi_2\}$). Under the rules for (underapproximate) refinement types, to prove such a relation is valid we need to prove that the set of values represented by the left type is contained in the set of values represented by the right:

$$\{v \in T \mid \phi_1\} \subseteq \{v \in T \mid \phi_2\}$$

In other words, we need to prove the proposition

$$\forall v, \phi_1(v) \implies \phi_2(v)$$

holds for ϕ_1 and ϕ_2, which we pass to Z3 to verify. For a coverage subtyping query

$$[v : T \mid \phi_1] <: [v : T \mid \phi_2]$$

we instead need to prove the opposite: that the set of values represented by the left type fully covers the set of values represented by the right, i.e. we need to prove that

$$\{v \in T \mid \phi_2\} \subseteq \{v \in T \mid \phi_1\}$$

or equivalently that

$$\forall v, \phi_2(v) \implies \phi_1(v)$$

Concretely, Poirot can validate the following subtyping query

$$[v : \text{int} \mid v \geq 0] <: [v : \text{int} \mid v \geq 3]$$

as long as Z3 can validate the below proposition:

$$\forall v, v \geq 3 \implies v \geq 0$$

Beyond integers, Poirot also has the capability to reason about inductive data structures like lists, trees, option types, red-black trees, and more. As an additional part of the Poirot type system, Zhou et al. define special "type predicate" functions used in refinement types to qualify these data structures defined by the language.

As an example, Poirot includes built-in predicates relating lists with a notion of emptiness or an integer length:

```
val len : 'a list -> int -> bool
val emp : 'a list -> bool
```

These predicates allow us to define types like

$$[v : \text{int list} \mid \text{emp}\ v]$$

as the underapproximate refinement type covering all empty lists, or

$$[v : \text{int list} \mid \exists n, k, \text{len}\ v\ n\ \wedge n = 2k]$$

as the underapproximate refinement type covering all lists with even length.

However, Z3 treats predicates as uninterpreted symbols: there is no mechanism by which we can supply the definitions of these data structures or type predicates to Z3 such that it can automatically reason about the "meaning" of arbitrary predicates.

To work around this limitation, Poirot encodes the meaning of type predicates via pre-defined axioms supplied to Z3 that allow the SMT solver to relate these opaque symbols with each other. For example, in Poirot's OCaml axiom syntax we can write

```
let[@axiom] list_len_0_emp (l : int list) =
  (emp l) #==> (len l 0)
```

to define the axiom:

$$\text{list\_len\_0\_emp} \equiv \forall l, \text{emp}\ l \implies \text{len}\ l\ 0$$

allowing Z3 to conclude that if a list l is empty (emp l), we must have that its length is 0 (len l 0).

When typechecking programs with refinement types, it is thus important that sufficient user-supplied axioms exist to encode the expected meaning of predicates, while also ensuring that all supplied axioms are valid.

However, as the list of axioms grows (and especially with complicated queries), when a type-check fails it can be very hard to manually determine which specific axioms are missing from only the failing query.

3 Overview

As a very simple example, take the subtyping query that asserts that:

$$[v : \text{int list} \mid \exists n, k, \text{len}\ v\ n\ \wedge n = 2k] <: [v : \text{int list} \mid \text{emp}\ v]$$

(in other words, we want to assert that the type covering all even-length int lists can be used anywhere we require a type covering all empty int lists). Without the list_len_0_emp axiom, however, Z3 will fail to verify this.

Recall that in a coverage subtyping check we need to prove that the left type fully covers the right; in set notation, we want

$$\{v \in \text{int list} \mid \text{emp}\ v\} \subseteq \{v \in \text{int list} \mid \exists n, k, \text{len}\ v\ n\ \wedge n = 2k\}$$

or in other words, we want to be able to prove that

$$\forall v, \text{emp}\ v \implies \exists n, k, \text{len}\ v\ n \wedge n = 2k$$

To aid with axiom development, this project adds a module to the Poirot compiler that, when an SMT query fails, lifts the query (and all the predefined axioms passed to Z3) from Poirot's proposition AST to an interactive theorem prover (here, Rocq): at the cost of some automation, users can leverage Rocq's richer logic to reason directly about the semantics of predicates like emp and len. From Rocq, users can be directly exposed to the steps required to prove the query, the process of which should illuminate any specific missing implications in the current set of axioms.

In the earlier example, when the Z3 query fails, a Rocq file will be generated with the following goal:

```
Theorem goal : forall (v : list Z), emp v ->
  (exists (n : Z), exists (k : Z),
    len v n /\ n = (2 * k)).
```

We can now see that the list_len_0_emp axiom is sufficient to discharge the goal, and we can define the signature of the axiom in Rocq as:

```
Axiom list_len_0_emp :
  forall (l : list Z), emp l -> len l 0.
```

To satisfy the signature, we can add list_len_0_emp as a lemma, which also lets us use the provided definitions of emp and len to prove its correctness (via case analysis on lists):

```
Lemma list_len_0_emp :
  forall (l : list Z), emp l -> len l 0.
Proof.
  intros [| x] H.
  - simpl. reflexivity.
  - contradiction.
Qed.
```

Now that we've defined list_len_0_emp, we can prove our original goal:

```
Theorem goal : forall (v : list Z), emp v ->
  (exists (n : Z), exists (k : Z),
    len v n /\ n = (2 * k)).
Proof.
  intros v He.
  exists 0. exists 0.
  intuition.
  apply (list_len_0_emp v He).
Qed.
```

We can now translate the new axiom back to OCaml,

```
let[@axiom] list_len_0_emp =
  fun (l : (int list)) -> ((emp l) #==> (len l 0))
```

update Poirot's predefined axioms file, and rerun the subtyping check to see that it now passes.

4 Methods

Specifically, when a Z3 query fails, the module generates a Rocq file including translated axioms and the target query as a goal. To help users ensure that added axioms maintain

type system soundness, the generated Rocq file is split into three sections:

- The Signatures module type, which defines the public interface for the Axioms module (the signatures of the type predicates and axioms that the user will have access to when proving the goal).
- The Axioms module, which defines the (proofs of) axioms specified in the above signatures. In the axioms module, we also dump the definitions of all data structures and type predicates provided by Poirot, so that added axioms can be proven natively in Rocq to be correct.
- The Goal module, in which the axioms can be used to prove the original SMT query. Critically, we do not expose the predicate definitions in this module, forcing Rocq and the user to treat the predicates as opaque symbols (as Z3 would).

Together, these constraints ensure not only that if the query is provable in Rocq with a modified set of axioms that those axioms are likely to be sufficient for Z3 to discharge the original typecheck, but also that the modified set of axioms are actually correct (the user did not accidentally introduce a false statement as an axiom).

Once the query has been proven in Rocq, the (modified) list of axioms can be automatically translated back to Poirot's OCaml axiom language, added back to Poirot, and Poirot's type checker can be rerun with the updated axioms on the failing test case.

5 Results

In practice, many of the queries Poirot needs to make to Z3 during type checking aren't as simple as the example in section 3. To ensure that the tool generalizes to real-world use cases, the project was benchmarked against coverage subtype- and type-checking test cases imported from an existing project, Cobb [LaFontaine et al. 2025] (see table 1 for results).

Benchmarks were run with Rocq version 9.1.0 and Z3 version 4.15.2. For consistency, each benchmark was run with a script that

1. Runs the subtype or type check with an insufficient set of axioms causing Poirot fail and emit a Rocq file.
2. Edits in the required axioms, proofs of the axioms, and proof of the query in the Rocq file.
3. Runs rocq c to verify that the query is now proven in Rocq.
4. Runs the Rocq → axiom converter to generate a new set of axioms in Poirot's OCaml syntax.
5. Sets Poirot to use the new axioms, and rerun the same subtype / type check (asserting that Z3 now is able to successfully validate the query).

Using this framework, successful typing and subtyping benchmarks are laid out as subdirectories in the data/bench/typing

Table 1. Cobb benchmark results. Metrics: axioms each benchmark was run with (bef), # tactics required to prove axioms and the query (tac), and final axiom count (aft). Subtyping benchmarks are listed in the top section, and type checks in the bottom. A checkmark represents a passing benchmark, and an X represents a benchmark that was proven in Rocq, but failed to verify in Z3.

Benchmark name	Bef.	Aft.	Tac.	Pass
emp	0	1	7	✓
unique_emp	1	2	7	✓
unique_emp_rev	1	2	10	✓
unique_non_emp	2	4	42	✓
unique_non_emp_rev	4	6	29	✓
even_list_empty	1	2	8	✓
even_list_empty_rev	1	2	14	✓
even_list_singleton	4	5	23	✓
even_list_singleton_rev	6	7	32	✓
sorted_non_emp	4	7	53	✓
sorted_non_emp_rev	3	5	39	✓
duplicate_emp_rev	0	1	10	✓
duplicate_non_emp	4	6	30	✓
duplicate_non_emp_rev	3	5	32	✓
leaf	0	2	12	✓
complete_leaf	0	1	8	✓
complete_non_empty	0	3	46	✓
complete_non_empty_rev	2	8	71	✓
bst_leaf	0	1	9	✓
bst_node	2	9	140	✓
bst_node_rev	12	14	77	✓
bst_non_zero	11	17	137	✗
rbleaf	0	3	20	✓
rbleaf_rev	0	2	27	✓
rbleaf2	0	3	20	✓
rbleaf2_rev	0	2	22	✓
rbnode_single	3	5	51	✓
rbnode_single_rev	0	5	94	✓
rbnode_color_true	0	3	44	✓
rbnode_color_true_rev	4	10	87	✗
rbnode_color_false_node	3	4	37	✓
rbnode_color_false_node_rev	11	15	90	✗
rbnode_color_false_node_true	1	2	19	✓
rbnode_color_false_node_true_rev	8	10	46	✗
sizedlist	3	6	56	✓
uniquelist	2	8	77	✓
even_list	5	7	67	✓
sortedlist	6	8	57	✓
duplicatelist	6	8	44	✓
depthtree	5	9	78	✓
complete_tree	5	9	89	✓
depth_bst	15	17	109	✗
rbtree	14	16	156	✓

and data/bench/subtyping directories, respectively. For each test case, the type context and axioms are given in basic_typing.ml, refinement_typing.ml, and axioms_pre.ml. The Rocq file with proven query is given in query.v, and the final (generated) set of axioms are given in axioms_post.ml.

During testing, subtyping queries required between 30 minutes to an hour to prove by hand (including the proofs of a few relevant axioms, and the proof of the query using the added axioms). The generated Rocq proofs did not require complex techniques: axiom proofs used standard destruct and induction tactics, with the query proofs requiring applying and destructing axioms for anything involving predicates (a common pattern was to use axioms to destruct data structures in place of direct case analysis):

```
assert (n > 0) by
  (apply tree_depth_geq_0 in Hdpt; lia).
assert (~leaf v) by
  (apply (tree_positive_depth_is_not_leaf v n);
   intuition).
destruct (tree_no_leaf_exists_root v) as [x Hrt].
destruct (tree_no_leaf_exists_lch v) as [l Hl].
destruct (tree_no_leaf_exists_rch v) as [r Hr].
destruct (tree_depth_exists l) as [nl Hnl].
destruct (tree_depth_exists r) as [nr Hnr].
```

Benchmarking on Cobb also demonstrated the value of proving axioms from definitions before using them in the query. For instance, benchmarking on unique lists caught an old CoverageType axiom that was incorrect:

```
let[@axiom] list_hd_unique (l : int list)
  (l1 : int list) (x : int) =
  (tl l l1 && uniq l && hd l1 x)
  #==> (not (list_mem l1 x))
```

where the correct uniqueness axiom should be written instead as

```
let[@axiom] list_hd_unique (l : int list)
  (l1 : int list) (x : int) =
  (tl l l1 && uniq l && hd l x)
  #==> (not (list_mem l1 x))
```

(for a unique list, it is the head element of the *parent* list that should not be a member of its tail; indeed,

$$\text{head}\ l_1\ x \implies \neg(\text{list\_mem}\ l_1\ x)$$

is always a false statement).

The axioms module also serves to tightly couple users' perceived definitions of predicates with their actual definitions as encoded by the supplied axioms; during benchmarking on Cobb, this coupling was able to identify mismatches between axioms and OCaml definitions of predicates given in Cobb. As one example, Cobb's lower_bound and upper_bound predicates (used for binary search tree queries) were defined using $\leq$, so that a tree with all nodes having the same values is considered a BST,

```
let rec lower_bound t x =
  match t with
  | Leaf -> true
  | Node (y, l, r) -> x <= y && lower_bound l x
    && lower_bound r x

let rec upper_bound t x =
  match t with
  | Leaf -> true
  | Node (y, l, r) -> y <= x && upper_bound l x
    && upper_bound r x

let rec bst t =
  match t with
  | Leaf -> true
  | Node (x, l, r) -> bst l && bst r
    && upper_bound l x && lower_bound r x
```

while the BST queries (and the axioms used to prove such queries) relied on strictly less- and greater-than bounds:

```
let[@axiom] tree_upper_bound_root (l : int tree)
  (x : int) (y : int) =
  (bst l && root l x && upper_bound l y)
  #==> (y > x)
```

As another example, benchmarking caught conflicting meanings of the boolean color field in Cobb's red-black tree predicates:

```
let rec num_black t h : bool =
  match t with
  | Rbtleaf -> h = 0
  | Rbtnode (c, l, _, r) ->
    if c then
      num_black l (h - 1) && num_black r (h - 1)
    else num_black l h && num_black r h

(* No red node has a red child *)
let rec no_red_red t : bool =
  match t with
  | Rbtleaf -> true
  | Rbtnode (c, l, _, r) -> (
    if not c then no_red_red l && no_red_red r
    else (* c is true *)
      match (l, r) with
      | Rbtnode (c', _, _, _),
        Rbtnode (c'', _, _, _) ->
        (not c') && (not c'')
        && no_red_red l && no_red_red r
      | Rbtnode (c', _, _, _), Rbtleaf ->
        (not c') && no_red_red l
      | Rbtleaf, Rbtnode (c'', _, _, _) ->
        (not c'') && no_red_red r
      | Rbtleaf, Rbtleaf -> true)
```

(here, the definition of num_black treats c = true as denoting a black node, while the definition of no_red_red treats c = true as denoting a red node).

In both cases, mistakes were subtle: the fix for the axiom required only a single character change, and the predicate fix a single negation. By translating axioms and definitions into Rocq and forcing proofs of axioms with the given definitions, however, these errors result in unprovable propositions that surface the mistake explicitly.

6 Future work

In its current iteration, one large limitation of the tool is that predicate and data type definitions are hard-coded and need to manually added to the tool. To address this, future work can explore automatic Rocq definition translation from definitions of predicates supplied, for example, in OCaml.

Another improvement to the current tool can be axiom / definition filtering when generating Rocq files. Currently, for correctness, the tool emits all signatures in the typechecker's basic type context and axioms passed to the prover as definitions and axioms in the generated proof file. While this did not affect benchmarking (as each benchmark's type context and axiom list was chosen to be minimal), this may prove cumbersome to users in practice— the CoverageType repository, for example, has 23 predefined data type definitions, 87 predicate definitions, and 136 axioms, many of which will be unrelated to any specific query a user needs to prove. To address this, functionality can be added to automatically filter out data type and predicate definitions not referenced by the query, or axioms not involving predicates in the query. In such a case, however, care has to be taken not to filter too aggressively; provided axioms may relate predicates in the query with predicates *not* contained in the query, and the implemented filtering algorithm would need to pull in axioms on those predicates as well (until a fixpoint is reached).

A final consideration are the differences between the way definitions, axioms, and predicates are modeled in Z3 and Rocq. Currently, in Rocq predicates are conservatively modeled as returning Rocq propositions:

```
Definition leaf {a : Type} (t : tree a) : Prop :=
  match t with
  | Leaf _ => True
  | Node _ _ _ _ => False
  end.
```

However, without the axiom of choice this does not allow case analysis on the return value of predicates when doing proofs: in the rbtree benchmark, proving the query in Rocq required adding a choice axiom

```
Lemma rbtree_leaf_or_not : forall (l : rbtree Z),
  rb_leaf l \/ ~(rb_leaf l).
Proof.
  intros [].
  - left. reflexivity.
  - right. auto.
Qed.
```

that was not needed to validate the query in Z3 (as Z3 can natively destruct on whether rb_leaf l returns true or false for any given l).

7 Related work

This project was built on top of (and benchmarked using) existing research into coverage type systems, including Poirot [Zhou et al. 2023] and Cobb [LaFontaine et al. 2025]. This project also depends on interactive theorem provers like Rocq [Herbelin et al. 2026] and SMT solvers like Z3 [de Moura and Bjørner 2008].

There is also existing research into (more general) tools, called hammers, that do the opposite translation: calling an automated theorem prover from an interactive theorem prover like Rocq to automatically discharge proofs [Czajka and Kaliszyk 2018]. While their aims are in some sense "opposite" to ours, tools like hammers can aid in automation of some of the manual parts of the proof process, and help users further verify during the proving process that a successful Rocq proof using only the exposed axioms will indeed imply that given those same axioms, Z3 should be able to prove the query.

8 Conclusion

This paper introduces a framework aiding in the development of solver axioms for refinement type systems qualified with type predicates by translating SMT failures to an interactive theorem prover, allowing users to inspect queries and reason directly about missing axioms. By enabling users to prove each additional axiom using definitions of type predicates, this translation also ensures that users do not accidentally introduce incorrect axioms. Benchmarking on test cases used by Cobb show that the technique is generalizable to real world refinement type systems.

References

Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for Dependent Type Theory. *Journal of Automated Reasoning* 61, 1 (01 Jun 2018), 423–453. https://doi.org/10.1007/s10817-018-9458-4

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

Hugo Herbelin, Pierre-Marie Pédrot, coqbot, Gaëtan Gilbert, Maxime Dénès, letouzey, Emilio Jesús Gallego Arias, Matthieu Sozeau, Théo Zimmermann, Enrico Tassi, Jean-Christophe Filliatre, Pierre Roux, Guillaume Melquiond, Arnaud Spiwack, Jason Gross, barras, Pierre Boutillier, Vincent Laporte, Jim Fehrle, Stéphane Glondu, Yves Bertot, Jasper Hugunin, Clément Pit-Claudel, Ali Caglayan, forestjulien, Pierre Courtieu, Frédéric Besson, Pierre Rousselin, MSoegtropIMC, and Yann Leray. 2026. *rocq-prover/rocq: Rocq 9.1.1.* https://doi.org/10.5281/zenodo.18554917

Patrick LaFontaine, Zhe Zhou, Ashish Mishra, Suresh Jagannathan, and Benjamin Delaware. 2025. We've Got You Covered: Type-Guided Repair of Incomplete Input Generators. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 380 (Oct. 2025), 29 pages. https://doi.org/10.1145/3763158

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 (June 2023), 24 pages. https://doi.org/10.1145/3591271