Proof Axiom Strengthening via Interactive Verification

Kevin Yu

Background

- A **refinement type** is a type qualified under a constraint phi, denoted $\{v : T \mid \phi\}$
- Example: non-negative `int`s $\{v : \text{int} \mid v \geq 0\}$

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

- A **coverage type** is like a refinement type, but an expression has a coverage type if its set of possible values fully *covers* the type; write $[v : T \mid \phi]$
- Why? Can be used to e.g. verify that a generator function can actually generate a full range of inputs (PBT)

$$[v : \text{int} \mid 0 \leq v \leq s]$$

"This function must be able to generate all ints in [0, s]"

Queries, subtyping

- To typecheck programs with refinement types, a typechecker needs to make queries to an **automated theorem prover** (here Z3).
- Main example: **subtyping queries** (can type A be used in place of type B?); write "A <: B"

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

implies that...

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

implies that...

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

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

implies that...

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

implies that...

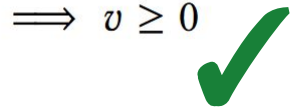
$$\forall v, \phi_2(v) \implies \phi_1(v)$$

We can assert...

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

so long as Z3 can prove...

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



Predicates, axioms, motivation

- To enable reasoning about arbitrary data types, we can declare "**type predicates**", functions with some semantic meaning ex.

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

$[v : \text{int list} \mid \text{emp } v]$ Covers all empty lists

$[v : \text{int list} \mid \exists n, k, \text{len } v\ n\ \wedge n = 2k]$ Covers all even-length lists

- Z3 has no innate knowledge about these functions (uninterpreted)
- Instead, impose meaning by providing Z3 with axioms:

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

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

- Axiom development is hard: need to ensure axioms encode all expected meaning of predicates, and none are wrong (but there's very little feedback)

High level overview

- Add a module to the type checker that generates an **interactive theorem prover** (here Rocq) file so that developers can debug axioms directly

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

↓ Z3 query

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

↓ on failure, generate Rocq file

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

↓ once proved, translate new axioms back

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

✓

↓ re-run the typecheck

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

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.
```

Results

- Axioms can be proved → catch incorrect axioms
- Caught axiom / predicate definition mismatch
- Rocq and Z3 don't always line up

```
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))

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))
```

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				
rbtree	14	16	156	✓

```
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)
```

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	0	0	0	
duplicate_emp_rev	0	1	10	✓
duplicate_non_emp	4	6	30	✓
duplicate_non_emp_rev	3	5	32	✓
leaf	0	2	12	✓
leaf_rev	0	0	0	
complete_leaf	0	1	8	✓
complete_leaf_rev	0	0	0	
complete_non_empty	0	3	46	✓
complete_non_empty_rev	2	8	71	✓
bst_leaf	0	1	9	✓
bst_leaf_rev	0	0	0	
bst_node	2	9	140	✓
bst_node_rev				
bst_non_zero	–	–	–	
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	✗