Mutation Analysis for Coq

Pengyu Nie[1]
Ahmet Celik[1], Karl Palmskog[1], Marinela Parovic[1],
Emilio Jesús Gallego Arias[2], and Milos Gligoric[1]

ASE 2019

1 THE UNIVERSITY OF TEXAS AT AUSTIN

2 MINES ParisTech | PSL

Program Verification Using Proof Assistants

- Verified software: encode program in formalism, write **specifications** for functions and prove them
- Proof assistants: prove specifications **interactively**
  - Coq, HOL4, HOL Light, Isabelle/HOL, Lean, Nuprl, ...
- Verified executable systems built using proof assistants are reaching unprecedented scale
  - CompCert (C compiler), 8 person years, 120k LOC
  - seL4 (OS kernel), 25 person years, 200k LOC
  - Verdi Raft (consensus protocol), 2 person years, 50k LOC
- These systems are being deployed
  - CompCert – embedded systems
  - seL4 – military autonomous vehicles

An Example of a Verified Function

- github.com/uwplse/StructTact
- Using Coq proof assistant
- Dependency of large verified systems, including Verdi Raft and Oeuf compiler

```
From mathcomp Require Import all_ssreflect.

Fixpoint before_func A (f g : A → bool) (l : list A) : bool :=
 match l with
 | [::] ⇒ false
 | a :: l' ⇒ (f a == true) || (g a == false && before_func A f g l')
 end.

Lemma before_func_app : ∀ A (f g : A → bool) (l l' : list A),
 before_func A f g l → before_func A f g (l ++ l').
Proof.
intros;induction l⇒ /=; intuition; move/orP: H; case; [by move/eqP→ |].
by move/andP⇒  [H1 H2]; rewrite H1 /=; apply/orP; right; apply IHl.
Qed.
```

Before.v

An Example of a Verified Function

- github.com/uwplse/StructTact
- Using Coq proof assistant
- Dependency of large verified systems, including Verdi Raft and Oeuf compiler

Import

```
From mathcomp Require Import all_ssreflect.

Fixpoint before_func A (f g : A → bool) (l : list A) : bool :=
 match l with
 | [::] ⇒ false
 | a :: l' ⇒ (f a == true) || (g a == false && before_func A f g l')
 end.

Lemma before_func_app : ∀ A (f g : A → bool) (l l' : list A),
 before_func A f g l → before_func A f g (l ++ l').
Proof.
intros;induction l⇒ /=; intuition; move/orP: H; case; [by move/eqP→ |].
by move/andP⇒  [H1 H2]; rewrite H1 /=; apply/orP; right; apply IHl.
Qed.
```

Before.v

An Example of a Verified Function

- github.com/uwplse/StructTact
- Using Coq proof assistant
- Dependency of large verified systems, including Verdi Raft and Oeuf compiler

Function

```
From mathcomp Require Import all_ssreflect.

Fixpoint before_func A (f g : A → bool) (l : list A) : bool :=
 match l with
 | [::] ⇒ false
 | a :: l' ⇒ (f a == true) || (g a == false && before_func A f g l')
 end.

Lemma before_func_app : ∀ A (f g : A → bool) (l l' : list A),
 before_func A f g l → before_func A f g (l ++ l').
Proof.
intros;induction l⇒ /=; intuition; move/orP: H; case; [by move/eqP→ |].
by move/andP⇒  [H1 H2]; rewrite H1 /=; apply/orP; right; apply IHl.
Qed.
```

Before.v

An Example of a Verified Function

- github.com/uwplse/StructTact
- Using Coq proof assistant
- Dependency of large verified systems, including Verdi Raft and Oeuf compiler

Specification

```
From mathcomp Require Import all_ssreflect.

Fixpoint before_func A (f g : A → bool) (l : list A) : bool :=
 match l with
 | [::] ⇒ false
 | a :: l' ⇒ (f a == true) || (g a == false && before_func A f g l')
 end.

Lemma before_func_app : ∀ A (f g : A → bool) (l l' : list A),
 before_func A f g l → before_func A f g (l ++ l').
Proof.
intros;induction l⇒ /=; intuition; move/orP: H; case; [by move/eqP→ |].
by move/andP⇒  [H1 H2]; rewrite H1 /=; apply/orP; right; apply IHl.
Qed.
```

Before.v

An Example of a Verified Function

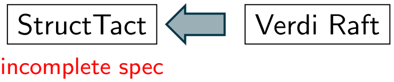
- github.com/uwplse/StructTact
- Using Coq proof assistant
- Dependency of large verified systems, including Verdi Raft and Oeuf compiler

```
From mathcomp Require Import all_ssreflect.

Fixpoint before_func A (f g : A → bool) (l : list A) : bool :=
 match l with
 | [::] ⇒ false
 | a :: l' ⇒ (f a == true) || (g a == false && before_func A f g l')
 end.

Lemma before_func_app : ∀ A (f g : A → bool) (l l' : list A),
 before_func A f g l → before_func A f g (l ++ l').
Proof.
intros;induction l⇒ /=; intuition; move/orP: H; case; [by move/eqP→ |].
by move/andP⇒  [H1 H2]; rewrite H1 /=; apply/orP; right; apply IH1.
Qed.
```

Proof

Before.v

Problem: Incomplete and Missing Specifications



- Specifications might not cover the core parts of the function
- Some functions might have no specification at all
- Usage of such functions could lead to surprises and even bugs
- **How do we detect incomplete and missing specifications?**

Our Contributions

1. Introduce **mutation proving** for proof assistants libraries
   - Mutate functions and check if any lemma fails
   - No lemma fails (live mutant) may indicate incomplete spec
   - Analogous to mutation testing
2. Implement mutation proving for Coq libraries, **mCoq**
3. Optimize mCoq with selective and parallel proof checking
4. Quantitatively evaluate mCoq on 12 popular Coq libraries
5. Qualitatively evaluate dozens of live mutants and report incomplete specifications

Mutation Proving Example, Original Code

```
From mathcomp Require Import all_ssreflect.

Fixpoint before_func A (f g : A → bool) (l : list A) : bool :=
 match l with
 | [::] ⇒ false
 | a :: l' ⇒ (f a == true) || (g a == false && before_func A f g l')
 end.

Lemma before_func_app : ∀ A (f g : A → bool) (l l' : list A),
 before_func A f g l → before_func A f g (l ++ l').
Proof.
intros;induction l⇒ /=; intuition; move/orP: H; case; [by move/eqP→ |].
by move/andP⇒ [H1 H2]; rewrite H1 /=; apply/orP; right; apply IHl.
Qed.
```

Before.v

✓ Proof passes

Mutation Proving Example, Mutated Code

```
From mathcomp Require Import all_ssreflect.

Fixpoint before_func A (f g : A → bool) (l : list A) : bool :=
 match l with
 | [::] ⇒ false
- | a :: l' ⇒ (f a == true) || (g a == false && before_func A f g l')
+ | a :: l' ⇒ (f a == true) || (g a == true && before_func A f g l')
 end.

Lemma before_func_app : ∀ A (f g : A → bool) (l l' : list A),
 before_func A f g l → before_func A f g (l ++ l').
Proof.
intros;induction l⇒ /=; intuition; move/orP: H; case; [by move/eqP→ |].
by move/andP⇒ [H1 H2]; rewrite H1 /=; apply/orP; right; apply IHl.
Qed.
```

Before.v

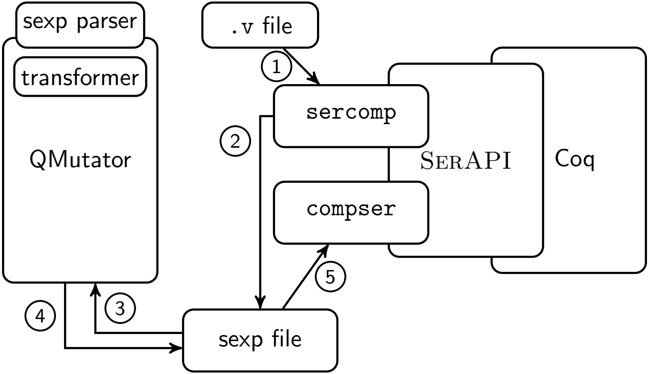
Mutation Proving Example, Mutated Code

```
From mathcomp Require Import all_ssreflect.

Fixpoint before_func A (f g : A → bool) (l : list A) : bool :=
 match l with
 | [::] ⇒ false
- | a :: l' ⇒ (f a == true) || (g a == false && before_func A f g l')
+ | a :: l' ⇒ (f a == true) || (g a == true && before_func A f g l')
 end.

Lemma before_func_app : ∀ A (f g : A → bool) (l l' : list A),
 before_func A f g l → before_func A f g (l ++ l').
Proof.
intros;induction l⇒ /=; intuition; move/orP: H; case; [by move/eqP→ |].
by move/andP⇒  [H1 H2]; rewrite H1 /=; apply/orP; right; apply IHl.
Qed.
```

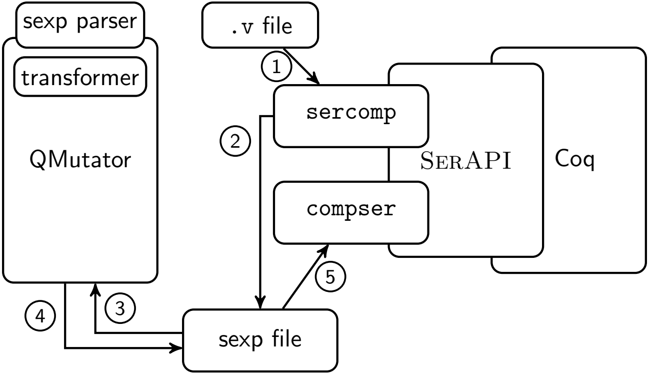
Before.v

✓ Proof still passes → Live mutant → Incomplete specification

mCoq Architecture and Workflow

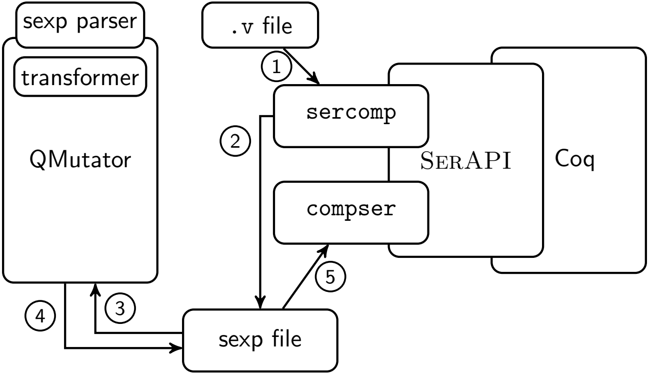


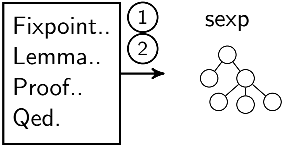
mCoq Architecture and Workflow



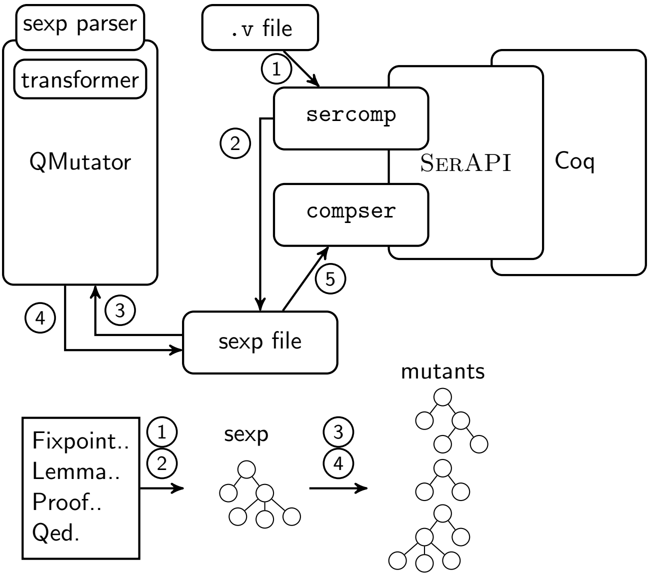
Fixpoint..
Lemma..
Proof..
Qed.

mCoq Architecture and Workflow

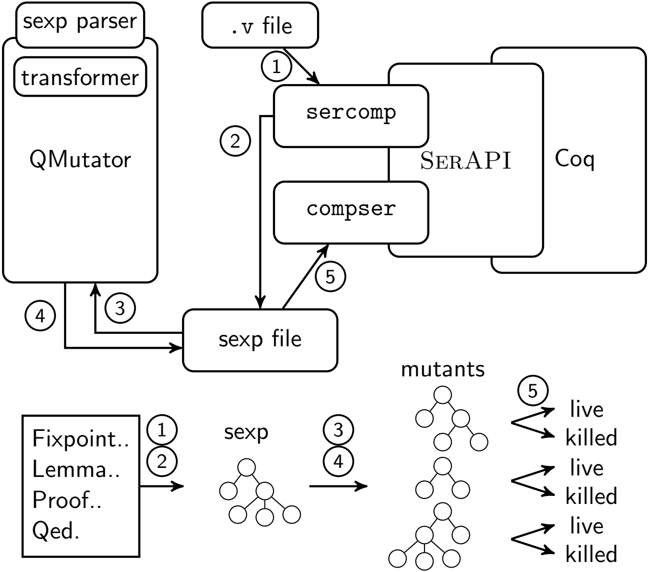




mCoq Architecture and Workflow



mCoq Architecture and Workflow



mCoq Components (1): Serialization and Deserialization

SerAPI extended OCaml library supporting full (de)serialization of Coq code

`sercomp` command-line SerAPI-based OCaml program which takes Coq `.v` file and outputs lists of sexps

`compser` command-line SerAPI-based program which takes lists of sexps and performs proof checking with Coq

mCoq Components (2): QMutator and Runner

QMutator
: sexp transformation library in Java that performs mutations given mutation operator and location

Runner
: driver program in Java and bash to orchestrate components and compute mutation scores

Mutation Operators

- Inspired by our experience (17 years cumulative) and mutation operators for functional languages

Category	Name	Description
General	GIB	Reorder branches in if-else expression
	GIC	Reverse constructor order in inductive type
	GME	Replace exp in the 2nd match case with 1st case exp
Lists	LRH	Replace list with head singleton list
	LRT	Replace list with its tail
	LRE	Replace list with empty list
	LAR	Reorder arguments to the list append operator
	LAF	Replace list append expression with first argument
	LAS	Replace list append expression with second argument
Numbers	NPM	Replace plus with minus
	NZO	Replace zero with one
	NSZ	Replace successor constructor with zero
	NSA	Replace successor constructor with its argument
Booleans	BFT	Replace `false` with `true`
	BTF	Replace `true` with `false`

Mutation Procedure Simplified

Require: G – Dependency Graph
Require: rG – Reverse Dependency Graph
Require: op – Mutation operator
Require: $sVFs$ – Topologically sorted `.v` files
Require: v – Set of visited `.v` files
Require: vF – `.v` file

procedure CHECKOPVFILE(G, rG, op, $sVFs$, v, vF)
    $sF \leftarrow \texttt{sercomp}(vF)$
    $mc \leftarrow countMutationLocations(sF, op)$
    $mi \leftarrow 0$
    while $mi < mc$ **do**
        $mSF \leftarrow mutate(sF, op, mi)$
        CHECKOPSEXPFILE($G, rG, sVFs, v, vF, mSF$)
        $mi \leftarrow mi + 1$
    end while
    revertFile(vF)
end procedure

Optimizations and mCoq Modes

Default	Simple mode, checks every file sequentially
RDeps	Advanced mode which checks only affected files and caches proof checking for unmodified files

Optimizations and mCoq Modes

Default	Simple mode, checks every file sequentially
RDeps	Advanced mode which checks only affected files and caches proof checking for unmodified files
ParMutant	Like RDeps, but checks each mutant in parallel
6-RDeps	Organizes operators into six groups, and runs each group in parallel using RDeps

More optimizations and modes in the paper

Evaluation: Research Questions

RQ1 What is the **number of mutants** of libraries and what are their **mutation scores**?

RQ2 What is the cost of mutation proving in terms of **execution time** and what are benefits of **optimizations**?

RQ3 Why are some mutants (not) killed?

RQ4 How does mutation proving compare to dependency analysis for finding incomplete and missing specifications?

Evaluation: Subjects and Environment

Library	#Files	Spec. LOC	Pr. LOC
ATBR	42	4123	5567
FCSL PCM	12	2939	2851
Flocq	29	5955	18044
Huffman	26	1878	4011
MathComp	89	37520	46040
PrettyParsing	14	1221	705
Bin. Rat. Numb	37	5500	29541
Quicksort Compl	36	2617	6202
Stalmarck	38	3552	7698
Coq-std++	43	6882	6852
StructTact	19	2008	2333
TLC	49	13217	7802
Avg.	36.16	7284.33	11470.50
Total	434	87412	137646

- 6-core Intel Core i7-8700 CPU @ 3.20GHz
- 64GB of RAM
- Ubuntu 18.04.1
- #parallel processes $\leq$ #CPU cores.

RQ1: Metrics

RQ1 What is the **number of mutants** of libraries and what are their **mutation scores**?

Live Mutants mutants that pass all proof checking

Killed Mutants mutants that cause a failing proof of any lemma

Mutation Score percentage of killed mutants out of all mutants

RQ1: Number of Mutants

Library	Total	Killed	Live
ATBR	355	335	20
FCSL PCM	115	112	3
Flocq	382	349	33
Huffman	369	366	3
MathComp	1037	1025	12
PrettyParsing	282	235	47
Bin. Rat. Numbers	365	352	13
Quicksort Compl.	681	637	44
Stalmarck	565	526	39
Coq-std++	564	515	49
StructTact	104	100	4
TLC	400	306	94
Avg.	434.91	404.83	30.08
Total	5219	4858	361

RQ1: Mutation Scores

Library	Score
ATBR	95.44
FCSL PCM	99.11
Flocq	93.31
Huffman	99.18
MathComp	98.84
PrettyParsing	83.33
Bin. Rat. Numbers	97.23
Quicksort Compl.	93.81
Stalmarck	93.26
Coq-std++	91.63
StructTact	96.15
TLC	76.88
Avg.	93.18

- Most have high mutation scores
  - Proof code is brittle
  - Specifications highly coupled to functions and datatypes
- Although mutation scores are high, each library has some live mutants

RQ2: Mutation Cost

RQ2 What is the cost of mutation proving in terms of **execution time** and what are benefits of **optimizations**?

Execution Time in Seconds

Library	Default	RDeps	ParMutant	6-RDeps
ATBR	2157.68	1760.27	596.21	755.40
FCSL PCM	153.22	150.88	53.33	109.51
Flocq	725.82	547.06	156.63	199.02
Huffman	188.64	185.70	62.46	72.38
MathComp	9962.99	8480.79	4053.67	3943.05
PrettyParsing	278.56	216.98	66.06	90.21
Bin. Rat. Numbers	1022.61	925.50	264.85	578.94
Quicksort Compl.	1594.66	1064.64	362.38	553.53
Stalmarck	805.84	498.01	192.78	230.62
Coq-std++	3187.80	2597.54	776.77	1137.16
StructTact	55.90	41.62	18.84	19.35
TLC	3128.85	1739.27	519.59	693.88
Avg.	1938.54	1517.35	593.63	698.58
Total	23262.57	18208.26	7123.57	8383.05

- ParMutant mode saves 70% time compared to Default mode

RQ3: Qualitative Analysis Procedure

RQ3 Why are some mutants (not) killed?

Goal Inspect 10% or more of all live mutants for each operator, and 10% or more of all live mutants for each library

1. Randomly choose 5% mutants to inspect from the set of all live mutants
2. Inspect all MathComp mutants
3. Reach the goal by sampling from underrepresented subsets

RQ3: Live Mutants Taxonomy

We manually inspected 74 live mutants (out of 361), which we labeled with one of:

- UnderspecifiedDef: The live mutant pinpoints a definition which lacks lemmas for certain cases (33 mutants)
- DanglingDef: The live mutant pinpoints a definition that has no associated lemma (30 mutants)
- SemanticallyEq: The live mutant is semantically equivalent to the original library (11 mutants)

RQ3: MathComp Live LRT Mutant

```
Fixpoint merge_sort_push (s1 : list T) (ss : list (list T)) :=
match ss with
| [::] :: ss' | [::] as ss' ⇒ s1 :: ss'
| s2 :: ss' ⇒
-  [::] :: merge_sort_push (merge s1 s2) ss'
+  merge_sort_push (merge s1 s2) ss'
end.
```

- UnderspecifiedDef
- Time complexity: $O(n \log n)$ to $O(n^2)$
- The key but unstated invariant of `ss` is that its ith item has size 2^i if it is not empty, so that `merge_sort_push` only performs perfectly balanced merges [...] without the `[::]` placeholder the MathComp sort becomes two element-wise insertion sort.

—Georges Gonthier

Impact on the Community

- Reported several incomplete or missing specifications, e.g., in StructTact and MathComp
- Improved SerAPI, and `sercomp` and `compser` already integrated
- Improved serialization support in Coq which has been merged to 8.10.0 release
- Discovered a serious bug in Coq related to proof processing using mCoq. We reported this bug and it was immediately fixed by the developers

Conclusion

- Technique for mutation proving for proof assistant libraries
- Coq tool, mCoq, implementing technique and optimizations
- Extensive quantitative and qualitative evaluation
  - mCoq finds incomplete/missing specs
- Impact on the Coq community (e.g., SerAPI)

Contact us: `http://cozy.ece.utexas.edu/mcoq`

THE UNIVERSITY OF TEXAS AT AUSTIN

Our other work: `https://proofengineering.org`

Backup Slides After This Point

Why Not Mutating Coq on Syntax Level

- Extensibility and flexibility of the syntax is a serious obstacle
- Coq supports defining powerful custom notations over existing specifications
- Coq's parser can be extended with large grammars at any point in a source file by loading plugins

Our Working Analogy: Proofs $\sim$ Tests

- tests are "partial functional specifications" of programs
- proofs represent many, usually an infinite number of, tests

```
Fixpoint app {A} (l m:list A)
:= match l with
   | [] ⇒ m
   | a :: l' ⇒ a :: app l' m
   end.
```

1. Coq function

```
Lemma assoc: ∀ A(l m n:list A),
app l(app m n) = app(app l m) n.
Proof.
induction l; intros; auto.
simpl; rewrite IHl; auto.
Qed.
```

2. Coq lemma

```
let test_app_assoc ctxt =
 assert_equal
  (app [1] (app [2] [3]))
  (app (app [1] [2]) [3])
```

3. OCaml test

Example Verified Function, Strong Lemma Added

```
Fixpoint before_func A (f : A → bool) (g : A → bool) (l : list A) : bool :=
 match l with
 | [::] ⇒ false
 | a :: l' ⇒ (f a == true) || (g a == false && before_func A f g l')
 end.

Lemma before_func_app : ∀ A (f g : A → bool) (l l' : list A),
 before_func A f g l → before_func A f g (l ++ l').
Proof.
intros;induction l⇒ /=; intuition; move/orP: H; case; [by move/eqP → |].
by move/andP⇒ [H1 H2]; rewrite H1 /=; apply/orP; right; apply IHl.
Qed.

Lemma before_func_antisym : ∀ A f g l,
 (∀ x, f x == true → g x == true → ⊥) →
 before_func A f g l → before_func A g f l → ⊥.
Proof.
move ⇒ A f g; elim ⇒ //= a l IH Hfg.
case/orP ⇒ Hf; case/orP ⇒ Hg ⇒ //=; first by eauto.
- by move/andP: Hg Hf ⇒ [Hfa Hb]; move/eqP: Hfa → .
- by move/andP: Hf Hg ⇒ [Hfa Hb]; move/eqP: Hfa → .
- by move/andP: Hf ⇒ [Hfa Hb]; move/andP: Hg ⇒ [Hga Hb']; eauto.
Qed.
```

RQ4: Comparison to Dependency Analysis

RQ4 How does mutation proving compare to **dependency analysis** for finding incomplete and missing specifications?

- compared to grep-based baseline ("do names occur in source files?")
- compared to term dependency extraction ("do names occur in elaborated terms?")
- conclusion: baseline is useless, term dependency lists are noisy

See paper for details!

RQ1: Outliers

Library	Score
ATBR	95.44
FCSL PCM	99.11
Flocq	93.31
Huffman	99.18
MathComp	98.84
PrettyParsing	83.33
Bin. Rat. Numbers	97.23
Quicksort Compl.	93.81
Stalmarck	93.26
Coq-std++	91.63
StructTact	96.15
TLC	76.88
Avg.	93.18

- Outliers with lower mutation scores
  - TLC: specifications are put in another library
  - PrettyParsing: many functions describe how prettification is done, but no specification for them

RQ3: Flocq Live GIB Mutant

```
 Definition Bplus op_nan m x y :=
 match x,y with
 | B754_infinity sx, B754_infinity sy ⇒
-  if Bool.eqb sx sy then x else build_nan (plus_nan x y)
+  if Bool.eqb sx sy then build_nan (plus_nan x y) else x
```

- UnderspecifiedDef
- Bplus lemmas rule out infinite cases through guards
- Same problem with Bminus function

Limitations and Future Work

- Design more mutation operators specialized for each library
- Scope of mutation is limited to definitions. This is analogy to mutation testing where mutation is limited to production code rather than test code
- Equivalence filtering uses syntactical equality, other equalities such as convertibility could be used
- Alternative mutation approaches during elaboration phase