Reasoning about Functions

Niki Vazou, Anish Tondwalkar, Ranjit Jhala

UC San Diego

Reasoning about Functions

Niki Vazou, Anish Tondwalkar, Ranjit Jhala

Motivation

Motivation: SMT is Robust!

For “Shallow” Specs in Decidable theories

SMT is Robust For “Shallow” Specs

```
sum n =
  if n <= 0
    then 0
    else n + sum (n - 1)
```

SMT is Robust For “Shallow” Specs

```
sum n =
  if n <= 0
    then 0
    else n + sum (n - 1)

goals =
  [ assert (0 <= sum 3) ]
```

Verify goals

SMT is Robust For “Shallow” Specs

```
sum n =
  @ensures (0 <= res)
  if n <= 0
    then 0
    else n + sum (n - 1)

goals =
  [ assert (0 <= sum 3) ]
```

Verify `goals` using *spec* for `sum`

SMT is Robust For "Shallow" Specs

```
sum :: n:_ -> res:{0 <= res}
sum n =
  if n <= 0
    then 0
    else n + sum (n - 1)

goals =
  [ assert (0 <= sum 3) ]
```

Verify `goals` using *spec* for `sum`

SMT is Robust For "Shallow" Specs

```
sum :: n:_ -> res:{0 <= res}
sum n =
  if n <= 0
    then 0
    else n + sum (n - 1)

goals =
  [ assert (0 <= sum 3) ]
```

Verification Conditions

$$0 < n \Rightarrow 0 \leq \mathsf{sum}(n-1) \Rightarrow 0 \leq n + \mathsf{sum}(n-1)$$

SMT is Robust For "Shallow" Specs

```
sum :: n:_ -> res:{0 <= res}
sum n =
  if n <= 0
    then 0
    else n + sum (n - 1)

goals =
  [ assert (0 <= sum 3) ]
```

Verification Conditions

$$0 < n \Rightarrow 0 \leq \mathsf{sum}(n-1) \Rightarrow 0 \leq n + \mathsf{sum}(n-1)$$

$$0 \leq \mathsf{sum}(3) \Rightarrow 0 \leq \mathsf{sum}(3)$$

SMT is Robust For "Shallow" Specs

```
sum :: n:_ -> res:{0 <= res}
sum n =
  if n <= 0
    then 0
    else n + sum (n - 1)

goals =
  [ assert (0 <= sum 3) ]
```

SMT Solves Verification Conditions

$$0 < n \Rightarrow 0 \leq \mathsf{sum}(n-1) \Rightarrow 0 \leq n + \mathsf{sum}(n-1)$$ ✓

$$0 \leq \mathsf{sum}(3) \Rightarrow 0 \leq \mathsf{sum}(3)$$ ✓

SMT is Robust For “Shallow” Specs

SMT solves decidable* VCs…

*__Quantifier Free__ Equality, UIF, Arithmetic, Sets, Maps, Bitvectors….

SMT is Robust For “Shallow” Specs

SMT solves decidable VCs…

SMT is Brittle For “Deep” Specs

…VCs over user-defined functions

SMT is Brittle For "Deep" Specs

```
sum :: n:_ -> res:{???}
sum n =
  if n <= 0
    then 0
    else n + sum (n - 1)

goals =
  [ assert (sum 3 == 6) ]
```

A suitable *spec* for `sum`?

SMT is Brittle For “Deep” Specs

```
sum :: n:_ -> res:{???}
sum n =
  if n <= 0
    then 0
    else n + sum (n - 1)

goals =
  [ assert (sum 3 == 6) ]
```

A suitable *spec* for `sum` needs axioms!

$$\forall n.\ n \leq 0 \Rightarrow \mathsf{sum}(n) = 0$$

$$\forall n.\ 0 < n \Rightarrow \mathsf{sum}(n) = n + \mathsf{sum}(n-1)$$

SMT is Brittle For "Deep" Specs

A suitable *spec* for `sum` needs axioms!

$$\forall n.\ n \leq 0 \Rightarrow \mathsf{sum}(n) = 0$$

$$\forall n.\ 0 < n \Rightarrow \mathsf{sum}(n) = n + \mathsf{sum}(n-1)$$

SMT is Robust For "Shallow" Specs

SMT solves decidable VCs

SMT is Brittle For "Deep" Specs

VCs over User-defined Functions

VCs over User-defined Functions

… are everywhere!

VCs over User-defined Functions

Laws

Transitivity, Associativity…

Optimizations

Optimization preserves behavior …

Code Invariants

Higher-order Contract Specifications…

Functional Correctness

Equivalence w.r.t. to reference implementation

Motivation

VCs over User-defined Functions

Motivation

SMT Reasoning about Functions

LEON [“Satisfiability Modulo Recursive Functions”, Suter et al. 2011]
DAFNY [“Computing with an SMT Solver”, Amin et al. 2014]

SMT Reasoning about Functions

I

Equational Proof

V

II

Proof Synthesis

MC

III

Synthesis Terminates

AI

SMT Reasoning about Functions

I

Equational Proof

V

II

Proof Synthesis

MC

III

Synthesis Terminates

AI

I
Equational Proof

A suitable *spec* for sum?

```
sum :: n:_ -> res:{???}
sum n =
  if n <= 0
    then 0
    else n + sum (n - 1)

goals =
  [ assert (sum 3 == 6) ]
```

A suitable *spec* for sum?

reflect implementation as the specification

```
{-@ reflect sum @-}
sum n =
  if n <= 0
    then 0
    else n + sum (n - 1)

goals =
  [ assert (sum 3 == 6) ]
```

```
sum :: n:_ -> v:{v = if n <= 0 then 0 else n + sum(n-1)}
```

A suitable *spec* for `sum`?

`reflect` implementation as the specification

```
sum :: n:_ -> v:{v = if n <= 0 then 0 else n + sum(n-1)}
```

A. `sum` Must Terminate on All Inputs

Ensures soundness

A suitable *spec* for `sum`?

`reflect` implementation as the specification

```
sum :: n:_ -> v:{v = if n <= 0 then 0 else n + sum(n-1)}
```

B. `sum` is an uninterpreted function

$$\forall x, y : \; x = y \Rightarrow f(x) = f(y)$$

A suitable *spec* for `sum`?

`reflect` implementation as the specification

```
sum :: n:_ -> v:{v = if n <= 0 then 0 else n + sum(n-1)}
```

B. `sum` is an uninterpreted function

Ensures SMT can decide VCs

A suitable *spec* for `sum`?

`reflect` implementation as the specification

A. `sum` Must Terminate on All Inputs

Ensures soundness

B. `sum` is an uninterpreted function

Ensures SMT can decide VCs

Equational Proof

Step 1

`reflect` implementation as the specification

Step 2

Call function to "unfold" definition

Call function to "unfold" definition

```
{-@ reflect sum @-}
sum n =
  if n <= 0
    then 0
    else n + sum (n - 1)

goals =
  [ assert (sum 0 == 0) ]
```

Verification Condition*

$$\boxed{(\mathsf{sum}(0) = \mathit{if}\ \ (0 \leq 0)\ \ \mathit{then}\ 0\ \ \mathit{else}\ \ \ldots)} \Rightarrow \mathsf{sum}(0) = 0$$

* At callsite, substitute *actuals* for *formals* in Post-Condition [Floyd-Hoare]

Call function to “unfold” definition

```
{-@ reflect sum @-}
sum n =
  if n <= 0
    then 0
    else n + sum (n - 1)

goals =
  [ assert (sum 0 == 0) ]
```

Verification Condition

$$(\mathsf{sum}(0) = \mathit{if}\ (0 \leq 0)\ \mathit{then}\ 0\ \mathit{else}\ \ldots) \Rightarrow \boxed{\mathsf{sum}(0) = 0}$$

Call function to "unfold" definition

```
{-@ reflect sum @-}
sum n =
  if n <= 0
    then 0
    else n + sum (n - 1)

goals =
  [ assert (sum 0 == 0) ]
```

Verification Condition

$$(\mathsf{sum}(0) = \mathit{if}\ (0 \leq 0)\ \mathit{then}\ 0\ \mathit{else}\ \ldots) \Rightarrow \mathsf{sum}(0) = 0 \quad ✓$$

Call function to "unfold" definition

```
{-@ reflect sum @-}
sum n =
  if n <= 0
    then 0
    else n + sum (n - 1)

goals =
  [ assert (sum 2 == 3) ]
```

Verification Condition Invalid

$$(\mathsf{sum}(2) = \mathit{if}\ 2 \leq 0\ \mathit{then}\ 0\ \mathit{else}\ 2 + \boxed{\mathsf{sum}(1)}) \Rightarrow \mathsf{sum}(2) = 3$$ ✗

* VC has no information about `sum(1)`

Call function to “unfold” definition

```
{-@ reflect sum @-}
sum n =
  if n <= 0
    then 0
    else n + sum (n - 1)

goals =
  [ assert (sum 2 == 3) ]
```

If at first you don’t succeed…

Verification Condition Invalid

$$(\mathsf{sum}(2) = \mathit{if}\ 2 \leq 0\ \mathit{then}\ 0\ \mathit{else}\ 2 + \boxed{\mathsf{sum}(1)}) \Rightarrow \mathsf{sum}(2) = 3$$ ✗

* VC has no information about $\mathsf{sum}(1)$

Call function to “unfold” definition

```
{-@ reflect sum @-}
sum n =
  if n <= 0
    then 0
    else n + sum (n - 1)

goals =
  [ assert (sum 1 == 1)
  , assert (sum 2 == 3) ]
```

VC has no information about `sum(1)`

Call `sum(1)` to unfold specification…

Call function to "unfold" definition

```
{-@ reflect sum @-}
sum n =
  if n <= 0
    then 0
    else n + sum (n - 1)

goals =
  [ assert (sum 1 == 1)
  , assert (sum 2 == 3) ]
```

VC has no information about `sum(0)`

Call `sum(0)` to unfold specification…

Call function to “unfold” definition

```haskell
{-@ reflect sum @-}
sum n =
  if n <= 0
    then 0
    else n + sum (n - 1)

goals =
  [ assert (sum 0 == 0)
  , assert (sum 1 == 1)
  , assert (sum 2 == 3) ]
```

Call function to "unfold" definition

```
{-@ reflect sum @-}
sum n =
  if n <= 0
    then 0
    else n + sum (n - 1)

goals =
  [ assert (sum 0 == 0) ✓
  , assert (sum 1 == 1)
  , assert (sum 2 == 3) ]
```

VC

✓

$$(\mathsf{sum}(0) = \mathit{if}\ 0 \leq 0\ \mathit{then}\ 0\ \mathit{else}\ 0 + \mathsf{sum}(0-1)) \Rightarrow \mathsf{sum}(0) = 0$$

Call function to "unfold" definition

```
{-@ reflect sum @-}
sum n =
  if n <= 0
    then 0
    else n + sum (n - 1)

goals =
  [ assert (sum 0 == 0) ✓
  , assert (sum 1 == 1) ✓
  , assert (sum 2 == 3) ]
```

VC

$$
\begin{aligned}
&(\mathsf{sum}(0) = \mathit{if}\ 0 \leq 0\ \mathit{then}\ 0\ \mathit{else}\ 0 + \mathsf{sum}(0-1)) \\
\wedge &(\mathsf{sum}(1) = \mathit{if}\ 1 \leq 0\ \mathit{then}\ 0\ \mathit{else}\ 1 + \mathsf{sum}(1-1)) \Rightarrow \mathsf{sum}(1) = 1
\end{aligned}
$$

✓

Call function to "unfold" definition

```
{-@ reflect sum @-}
sum n =
  if n <= 0
    then 0
    else n + sum (n - 1)

goals =
  [ assert (sum 0 == 0) ✓
  , assert (sum 1 == 1) ✓
  , assert (sum 2 == 3) ✓]
```

VC

$$
\begin{aligned}
&(\mathsf{sum}(0) = \mathit{if}\ 0 \leq 0\ \mathit{then}\ 0\ \mathit{else}\ 0 + \mathsf{sum}(0-1)) \\
\wedge&(\mathsf{sum}(1) = \mathit{if}\ 1 \leq 0\ \mathit{then}\ 0\ \mathit{else}\ 1 + \mathsf{sum}(1-1)) \\
\wedge&(\mathsf{sum}(2) = \mathit{if}\ 2 \leq 0\ \mathit{then}\ 0\ \mathit{else}\ 2 + \mathsf{sum}(2-1)) \Rightarrow \mathsf{sum}(2) = 3
\end{aligned}
$$

✓

Equational Proof

Step 1

`reflect` implementation as the specification

Step 2

Call function to “unfold” definition

Equational Proof

Step 1

`reflect` implementation as the specification

Step 2

Call function to “unfold” definition (repeatedly!)

Tedious to unfold repeatedly!

Equational Proof

Step 1

`reflect` implementation as the specification

Step 2

Call function to “unfold” definition (repeatedly!)

Step 3

Combinators structure calls as *equations*

Equational Proof

Combinators structure calls as equations

```
(===) :: x:_-> y:{y=x} -> {v:v=x && v=y}
```

Combinator's Precondition

Input arguments must be equal

Equational Proof

Combinators structure calls as equations

```
(===) :: x:_-> y:{y=x} -> {v:v=x && v=y}
```

Combinator's Postcondition

Output value equals inputs

Equational Proof

Combinators structure calls as equations

```
goal2 () =
  assert (sum 2 == 3)
```

Verification `goal`

Equational Proof

Combinators structure calls as equations

```
goal2 () =
  @ensures (sum 2 == 3)
```

Verification goal

Rephrased as *post-condition*

Equational Proof

Combinators structure calls as equations

```
goal2 :: () -> { sum 2 == 3 }
```

Verification `goal`

Rephrased as *output-type*

Equational Proof

Combinators structure calls as equations

```
goal2 :: () -> {sum 2 == 3}
goal2 ()
  =    sum 2
  === 3
```

Invalid VC

VC has no information about `sum(1)`

Equational Proof

Combinators structure calls as equations

```
goal2 :: () -> {sum 2 == 3}
goal2 ()
  =    sum 2
  === 2 + sum 1
  === 3
```

Invalid VC

VC has no information about `sum(0)`

Equational Proof

Combinators structure calls as equations

```
goal2 :: () -> {sum 2 == 3}
goal2 ()
   =    sum 2
   === 2 + sum 1
   === 2 + 1 + sum 0
   === 3
```

✓

Equational Proof

Combinators structure calls as equations

```
(==?) :: x:_-> y:_ -> {y=x} -> {v:v=x && v=y}
```

Ternary "Because" Combinator

Third input "asserts" that first two are equal

Equational Proof

Combinators structure calls as equations

```
goal3 :: () -> {sum 3 == 6}
```

Equational Proof

Combinators structure calls as equations

```
goal3 :: () -> {sum 3 == 6}
goal3 ()
  =    sum 3
  === 3 + sum 2
  === 6
```

Invalid VC

VC has no information about `sum(2)`

Equational Proof

Combinators structure calls as equations

```
goal3 :: () -> {sum 3 == 6}
goal3 ()
  =    sum 3
  === 3 + sum 2
  ==? 3 + 3          ? goal2()
```

Post-condition adds sum(2) to VC

```
goal2 :: () -> {sum 2 == 3}
```

Equational Proof

Combinators structure calls as equations

```
goal3 :: () -> {sum 3 == 6}
goal3 ()
   =    sum 3
   === 3 + sum 2
   ==? 6               ? goal2()
```

✓

Equational Proof

Enables “deep” verification

Equational Proof

$$\forall 0 \leq n.\ 2 \times \mathsf{sum}(n) = n \times (n+1)$$

[Demo]

Equational Proof

$$\forall 0 \leq n.\ 2 \times \mathsf{sum}(n) = n \times (n+1)$$

```
sumPf :: n:{0 <=n} -> {2*sum n == n*(n+1)}

sumPf 0 =   2 * sum 0
        === 0

sumPf n =   2 * sum n
        === 2 * (n + sum (n-1))
        ==? 2 * n + (n-1) * n     ? sumPf (n-1)
        === n * (n+1)
```

Induction Hypothesis

Equational Proof

$$\forall xs, ys, zs.\ (xs +\!\!+\ ys) +\!\!+\ zs = xs +\!\!+\ (ys +\!\!+\ zs)$$

[Demo]

Equational Proof

$$\forall xs, ys, zs.\ (xs +\!\!+\ ys) +\!\!+\ zs = xs +\!\!+\ (ys +\!\!+\ zs)$$

```
appendPf :: xs:_ -> ys:_ -> zs:_ ->
              {(xs ++ ys) ++ zs = xs ++ (ys ++ zs)}

appendPf []       ys zs
  =    ([] ++ ys) ++ zs
  ===  [] ++ (ys ++ zs)

appendPf (x:xs) ys zs
  =    ((x:xs) ++ ys) ++ zs
  ===  (x : (xs ++ ys)) ++ zs
  ===  x : ((xs ++ ys) ++ zs)
  ==?  x : (xs ++ (ys ++ zs)) ? appendPf xs ys zs
  ===  (x:xs) ++ (ys ++ zs)
```

Induction Hypothesis

Equational Proof

Step 1

`reflect` implementation as the specification

Step 2

Call function to “unfold” definition (repeatedly!)

Step 3

Combinators structure calls as equations

SMT Reasoning about Functions

I

Equational Proof

V

II

Proof Synthesis

MC

III

Synthesis Terminates

AI

SMT Reasoning about Functions

I

Equational Proof

V

II

Proof Synthesis

MC

III

Synthesis Terminates

AI

II
Proof Synthesis

Proof Synthesis

Equational Proof is *very* expressive

Manual unfolding is tedious!

Manual unfolding is tedious!

$$\forall n.\ n > 2 \ \Rightarrow\ \mathsf{sum}(n) > 5 + \mathsf{sum}(n-3)$$

```
n:{n > 2} -> {sum n > 5 + sum(n-3)}
```

Manual unfolding is tedious!

```
ex :: n:{n > 2} -> {sum n > 5 + sum(n-3)}
```

Proof Synthesis

```
ex :: n:{n > 2} -> {sum n > 5 + sum(n-3)}
ex n  =   sum n
      === n + sum (n-1)
      === n + (n-1) + sum (n-2)
      === n + (n-1) + (n-2) + sum (n-3)
        > 5 + sum (n-3)
```

Manual unfolding is tedious!

Proof Synthesis

```
ex :: n:{n > 2} -> {sum n > 5 + sum(n-3)}

ex n  =    sum n
      === n + sum (n-1)
      === n + (n-1) + sum (n-2)
      === n + (n-1) + (n-2) + sum (n-3)
        > 5 + sum (n-3)
```

How to *automate* unfolding?

How to automate unfolding?

Problem

Completeness vs. *Termination*

[LEON] [DAFNY]

How to automate unfolding?

Problem

Completeness vs. Termination

Solution

Unfold if you *must*

Logical Evaluation

Unfold if you *must*

Logical Evaluation

Step 1

Represent functions in *guarded* form*

```
{-@ reflect sum @-}
sum n =
  if n <= 0
    then 0
    else n + sum (n-1)
```

Logical Evaluation

Step 1

Represent functions in *guarded* form*

```
sum n =
  | n <= 0 = 0
  | 0 <  n = n + sum (n-1)
```

| guard$_i$ = body$_i$

* *Every sub-term* in body$_i$ is evaluated when guard$_i$ is true

Logical Evaluation

Step 1

Represent functions in *guarded* form

Step 2

Unfold calls whose guard *is valid*

Logical Evaluation

Step 1

Represent functions in *guarded* form

Step 2

Unfold calls whose guard *is valid*

Logical Evaluation

Unfold calls whose guard *is valid*

```
n:{n > 2} -> {sum n > 5 + sum(n-3)}
```

Logical Evaluation

Unfold calls whose guard *is valid*

Assume	**Prove**
`n > 2`	`sum n > 5 + sum(n-3)`

Logical Evaluation

Unfold calls whose guard *is valid*

Assume	**Calls**	
`n > 2`	`sum n`	`sum(n-3)`

Unfold calls whose guard *is valid*

Assume		*Is valid?*
`n > 2`	$\Rightarrow$	$\boxed{\texttt{n > 0}}$ ✓
$\boxed{\texttt{sum(n) = n + sum(n-1)}}$		

Calls

$\boxed{\texttt{sum n}} \longrightarrow$ `sum(n-1)`

`sum(n-3)`

Unfold calls whose guard *is valid*

Assume *Is valid?*

```
                              n > 2
sum(n)   = n   + sum(n-1)   ⟹   n-1 > 0 ✓
sum(n-1) = n-1 + sum(n-2)
```

Calls

```
sum n ⟶ sum(n-1) ⟶ sum(n-2)
sum(n-3)
```

Unfold calls whose guard *is valid*

Assume *Is valid?*

```
                          n > 2
sum(n)    = n   + sum(n-1)
sum(n-1) = n-1 + sum(n-2)        ⇒     n-2 > 0  ✓
sum(n-2) = n-2 + sum(n-3)
```

Calls

```
sum n → sum(n-1) → sum(n-2) → sum(n-3)
sum(n-3)
```

Unfold calls whose guard *is valid*

	Assume	**Is valid?**
	`n > 2`	
	`sum(n)   = n   + sum(n-1)`	
	`sum(n-1) = n-1 + sum(n-2)`	
	`sum(n-2) = n-2 + sum(n-3)`	$\Longrightarrow$ $\boxed{\texttt{n-3 > 0}}$ ✗

Calls

$\texttt{sum n} \longrightarrow \texttt{sum(n-1)} \longrightarrow \texttt{sum(n-2)} \longrightarrow \boxed{\texttt{sum(n-3)}}$

Unfold calls whose guard *is valid*

Assume

```
                                  n > 2
sum(n)    = n   + sum(n-1)
sum(n-1) = n-1 + sum(n-2)
sum(n-2) = n-2 + sum(n-3)
```

Fixpoint!

Calls

`sum n` → `sum(n-1)` → `sum(n-2)` → `sum(n-3)`

Unfold calls whose guard *is valid*

Assume

```
                               n > 2
sum(n)   = n   + sum(n-1)
sum(n-1) = n-1 + sum(n-2)
sum(n-2) = n-2 + sum(n-3)
```

Fixpoint!

Assume *strengthened by* unfolded calls

Unfold calls whose guard *is valid*

Assume

```
                n > 2
sum(n)   = n   + sum(n-1)
sum(n-1) = n-1 + sum(n-2)
sum(n-2) = n-2 + sum(n-3)
```

$\Rightarrow$

Prove

```
sum n > 5 + sum(n-3)
```

✓

Assume *strengthened by* unfolded calls

Logical Evaluation

Step 1

Represent functions in *guarded* form

Step 2

Unfold calls whose guard *is valid*

Logical Evaluation

```
def PLE(D, A, G):

  C  = [x = f(t) for f(t) in G, x fresh]
  A* = A ∪ C

  while A ⊂ A*:
    A  = A*
    A* = Unfold(D, A)

  return IsValid(A* ⟹ G)
```

Algorithm: PLE

Logical Evaluation

```
def PLE(D, A, G):

  C  = [x = f(t) for f(t) in G, x fresh]
  A* = A ∪ C

  while A ⊂ A*:
    A  = A*
    A* = Unfold(D, A)

  return IsValid(A* ⟹ G)
```

(D)efinitions, (A)ssumptions, (G)oal

Logical Evaluation

```
def PLE(D, A, G):

  C  = [x = f(t) for f(t) in G, x fresh]
  A* = A ∪ C

  while A ⊂ A*:
    A  = A*
    A* = Unfold(D, A)

  return IsValid(A* ⟹ G)
```

Extend (A)ssumptions with calls in (G)oal

Logical Evaluation

```
def PLE(D, A, G):

  C  = [x = f(t) for f(t) in G, x fresh]
  A* = A ∪ C

  while A ⊂ A*:
    A  = A*
    A* = Unfold(D, A)

  return IsValid(A* ⟹ G)
```

Strengthen (A)ssumption with *fixpoint* of unfoldings

Logical Evaluation

```
def PLE(D, A, G):

  C  = [x = f(t) for f(t) in G, x fresh]
  A* = A ∪ C

  while A ⊂ A*:
    A  = A*
    A* = Unfold(D, A)

  return IsValid(A* ⟹ G)
```

Does strengthened (A)ssumption imply (G)oal ?

Logical Evaluation

```
def Unfold(D, A):
  return [ (f(x) = body)[t/x] |
              for f(t) in A
                for <guard = body> in D(f)
                  if IsValid(A ⟹ guard[t/x]) ]
```

Unfold

Returns equations for calls whose guard *implied by* A

Proof Synthesis

```
def PLE(D, A, G):
  …
  while A ⊂ A*:
    A  = A*
    A* = Unfold(D, A)
  …
  return IsValid(A* ⟹ G)
```

Logical Evaluation

Let $A^k = A$ after k loop iterations

Proof Synthesis

```
def PLE(D, A, G):
  …
  while A ⊂ A*:
    A  = A*
    A* = Unfold(D, A)
  …
  return IsValid(A* ⟹ G)
```

Logical Evaluation

Theorem

`IsValid`$(A^k \Longrightarrow G)$ if $A \to G$ with *size* k equational proof

Proof Synthesis

```
def PLE(D, A, G):
  …
  while A ⊂ A*:
    A  = A*
    A* = Unfold(D, A)
  …
  return IsValid(A* ⟹ G)
```

Logical Evaluation

Theorem

`IsValid`(A*⟹G) if A → G with *any* equational proof

Proof Synthesis

$$\forall n.\ n > 2 \ \Rightarrow\ \mathsf{sum}(n) > 5 + \mathsf{sum}(n-3)$$

[Demo]

Proof Synthesis

$$\forall 0 \leq n.\ 2 \times \mathsf{sum}(n) = n \times (n+1)$$

[Demo]

SMT Reasoning about Functions

I

Equational Proof

V

II

Proof Synthesis

MC

III

Synthesis Terminates

AI

SMT Reasoning about Functions

I

Equational Proof

V

II

Proof Synthesis

MC

III

Synthesis Terminates

AI

III
Synthesis Terminates

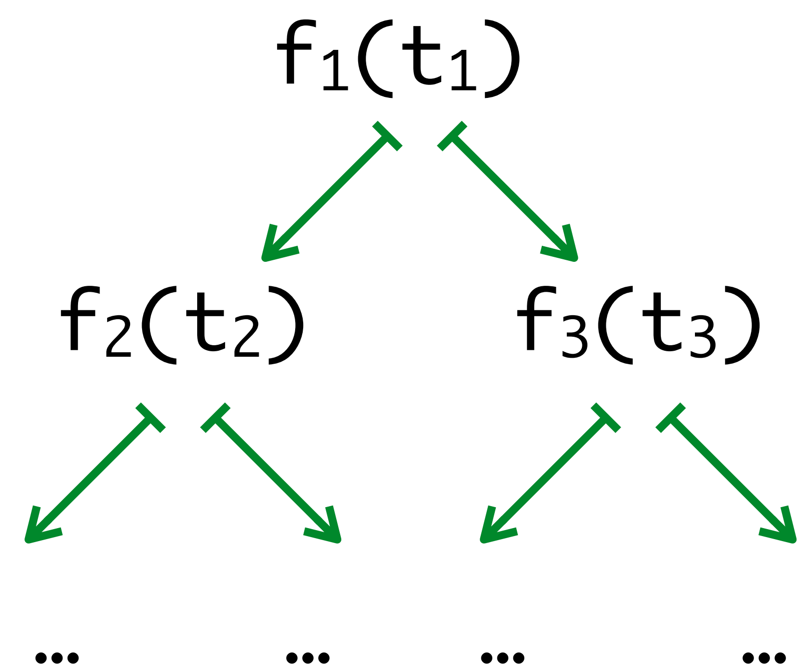
Synthesis Terminates

```
def PLE(D, A, G):
  …
  while A ⊂ A*:
    A  = A*
    A* = Unfold(D, A)
  …
  return IsValid(A* ⟹ G)
```

Why does PLE terminate?

Why does PLE terminate?

```
def PLE(D, A, G):
  …
  while A ⊂ A*:
    A  = A*
    A* = Unfold(D, A)
  …
  return IsValid(A* ⟹ G)
```



(Implicit) Tree of Logical Steps

$f_i(t_i)$ unfolds to body with $f_j(t_j)$

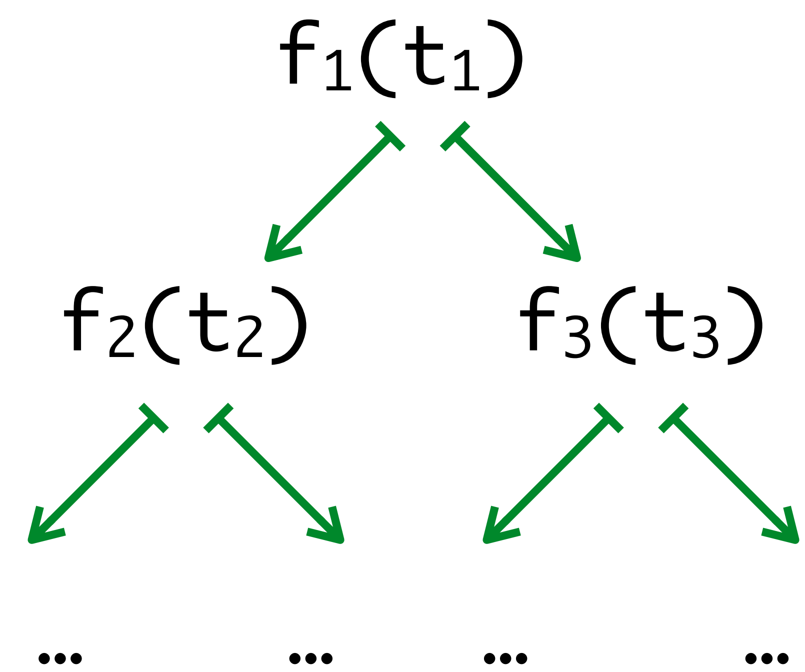
Why does PLE terminate?

PLE diverges

$\Rightarrow$ Tree is infinite

$\Rightarrow$ infinite Logical Path

$\Rightarrow$ infinite Concrete Trace



Reflected Functions Terminate!

(Required for soundness)

Logical Steps

$$D, \;\; A \;\vdash\; \mathsf{f}(\bar{t}) \longmapsto \mathsf{f}'(\overline{t'})$$

A implies *guard* of $\mathsf{f}(\bar{t})$ whose *body* has $\mathsf{f}'(\overline{t'})$

Logical Path $\Rightarrow$ Concrete Trace

Logical Steps are *Must*-Abstractions

If	$D,\ A\ \vdash\ \mathsf{f}(\overline{t}) \longmapsto \mathsf{f}'(\overline{t'})$
Then	$\forall \sigma \in [\![A]\!].\ \sigma(\mathsf{f}(\overline{t})) \hookrightarrow^{*} C[\sigma(\mathsf{f}'(\overline{t'}))]$

A implies *guard* of $\mathsf{f}(\overline{t})$ whose *body* has $\mathsf{f}'(\overline{t'})$

Logical Path $\Rightarrow$ Concrete Trace

Logical Steps are *Must*-Abstractions

$$\textbf{If} \quad D,\ A \ \vdash\ \mathsf{f}(\overline{t}) \longmapsto \mathsf{f}'(\overline{t'})$$

$$\textbf{Then} \quad \forall \sigma \in [\![A]\!].\ \sigma(\mathsf{f}(\overline{t})) \hookrightarrow^{*} C[\sigma(\mathsf{f}'(\overline{t'}))]$$

If A, *every* evaluation of $\mathsf{f}(\overline{t})$ *transitions to* $\mathsf{f}'(\overline{t'})$

Logical Path $\Rightarrow$ Concrete Trace

Logical Path $\Rightarrow$ Concrete Trace

If	$D,\ A\ \vdash\ \mathsf{f}_1(\overline{t_1}) \longmapsto \mathsf{f}_2(\overline{t_2}) \longmapsto\ \ldots$
Then	$\forall \sigma \in [\![A]\!].\ \sigma(\mathsf{f}_1(\overline{t_1})) \hookrightarrow^* C_2[\sigma(\mathsf{f}_2(\overline{t_2}))] \hookrightarrow^* \ldots$

If A, *every* evaluation of $\mathsf{f}(\bar{t})$ *transitions to* $\mathsf{f}'(\overline{t'})$

Logical Path $\Rightarrow$ Concrete Trace

If $D,\ A\ \vdash\ \mathsf{f}_1(\overline{t_1}) \longmapsto \mathsf{f}_2(\overline{t_2}) \longmapsto\ \ldots$

Then $\forall \sigma \in [\![A]\!].\ \sigma(\mathsf{f}_1(\overline{t_1})) \hookrightarrow^* C_2[\sigma(\mathsf{f}_2(\overline{t_2}))] \hookrightarrow^* \ldots$

i.e.

If infinite logical path , $[\![A]\!]$ not empty*

Then infinite concrete trace.

*A is satisfiable

Why does PLE terminate?

`PLE(D,A,G)` diverges

$\Rightarrow$ Tree is infinite

$\Rightarrow$ infinite logical path

$\Rightarrow$ infinite concrete trace.

Why does PLE terminate?

`PLE(D,A,G)` diverges

$\Rightarrow$ Tree is infinite

$\Rightarrow$ infinite logical path

$\Rightarrow$ infinite concrete trace. ✗

Synthesis Terminates

`PLE(D,A,G)` diverges

$\Rightarrow$ Tree is infinite

$\Rightarrow$ infinite logical path

$\Rightarrow$ infinite concrete trace. ✗

$\therefore$ `PLE(D,A,G)` terminates!

Reasoning about Functions

I

Equational Proof

V

II

Proof Synthesis

MC

III

Synthesis Terminates

AI

Reasoning about Functions

Laws

Transitivity, Associativity…

Optimizations

Optimization preserves behavior …

Code Invariants

Higher-order Contract Specifications…

Functional Correctness

Equivalence w.r.t. to reference implementation

Reasoning about Functions

Laws

Transitivity, Associativity…

Optimizations

Optimization preserves behavior …

[Demo]

Code Invariants

Higher-order Contract Specifications…

Functional Correctness

Equivalence w.r.t. to reference implementation

Reasoning about Functions

Benchmark	Common		Without PLE Search			With PLE Search		
	Impl (l)	Spec (l)	Proof (l)	Time (s)	SMT (q)	Proof (l)	Time (s)	SMT (q)
Arithmetic								
Fibonacci	7	10	38	2.74	129	16	1.92	79
Ackermann	20	73	196	5.40	566	119	13.80	846
Class Laws Fig 11								
Monoid	33	50	109	4.47	34	33	4.22	209
Functor	48	44	93	4.97	26	14	3.68	68
Applicative	62	110	241	12.00	69	74	10.00	1090
Monad	63	42	122	5.39	49	39	4.89	250
Higher-Order Properties								
Logical Properties	0	20	33	2.71	32	33	2.74	32
Fold Universal	10	44	43	2.17	24	14	1.46	48
Functional Correctness								
SAT-solver	92	34	0	50.00	50	0	50.00	50
Unification	51	60	85	4.77	195	21	5.64	422
Deterministic Parallelism								
Conc. Sets	597	329	339	40.10	339	229	40.70	861
n-body	163	251	101	7.41	61	21	6.27	61
Par. Reducers	30	212	124	6.63	52	25	5.56	52
Total	1176	1279	1524	148.76	1626	638	150.88	4068

Reasoning about Functions

Equational Proofs

Synthesized by Logical Evaluation

Equational Proofs

Synthesized by Logical Evaluation

SMT Automation is Great …

Short, Readable, High-level Proofs

… Except when A Proof Fails!

Counterexamples for true but *unprovable* facts?

Reasoning about Functions

Equational Proofs, Synthesized by Logical Evaluation

?

LiquidHaskell

bit.ly/liquidhaskell


If at first
you don't
succeed,
call it
version 1.0