Refinement Reflection: Complete Verification with SMT

Niki Vazou[1], Anish Tondwalkar[2],
Vikraman Choudhury[3], Ryan Scott[3], Ryan Newton[3],
Philip Wadler[4], and Ranjit Jhala[2]

[1]University of Maryland
[2]UC San Diego
[3]Indiana University
[4]University of Edinburgh

Verification with SMT

```
fib :: i:{Int|0≤i} -> {v:Int|0<v∧i≤v}
fib i
  | i≤1       = 1
  | otherwise = fib (i-1) + fib (i-2)
```

Verification with SMT

```
fib :: i:{Int|0≤i} -> {v:Int|0<v∧i≤v}
fib i
  | i≤1       = 1 ✓
  | otherwise = fib (i-1) + fib (i-2)
```

$$i \leq 1 \ \wedge \ v = \underline{1} \ \Rightarrow \ 0 < v \wedge i \leq v$$

$\boxed{\text{SMT-Valid}}$

Verification with SMT

```
fib :: i:{Int|0≤i} -> {v:Int|0<v∧i≤v}
fib i
  | i≤1       = 1 ✓
  | otherwise = fib (i-1) + fib (i-2) ✓
```

$$\begin{array}{c} \texttt{not } i \leq 1 \\ 0 < v_{i-1} \wedge i-1 \leq v_{i-1} \\ 0 < v_{i-2} \wedge i-2 \leq v_{i-2} \\ v = v_{i-1} + v_{i-2} \end{array} \Rightarrow 0 < v \wedge i \leq v$$

SMT-Valid

Verification with SMT

```
fib :: i:{Int|0≤i} -> {v:Int|0<v∧i≤v}
fib i
  | i≤1       = 1 ✓
  | otherwise = fib (i-1) + fib (i-2) ✓
```

Verified

Verification with SMT

To be **Practical**, SMT queries are **Decidable**

Can verification be Complete?

Complete Verification with SMT

```
fib :: i:{Int|0≤i} -> {v:Int|0<v∧i≤v}
fib i
  | i≤1       = 1
  | otherwise = fib (i-1) + fib (i-2)
```

How to express **theorems** about functions?

```
\forall i. 0 ≤ i => fib i ≤ fib (i+1)
```

How to express **theorems** about functions?

Step 1: Definition

In SMT `fib` is “Uninterpreted Function”

```
\forall i j. i = j => fib i = fib j
```

How to connect logic fib with target fib?

How to connect logic fib with target fib?

```
fib :: i:{Int|0≤i} -> {v:Int|0<v∧i≤v}
fib i
  | i≤1       = 1
  | otherwise = fib (i-1) + fib (i-2)
```

Not Decidable

SMT Axiom

```
\forall i.
   if i≤1 then fib i = 1
   else fib i = fib (i-1) + fib (i-2)
```

How to connect logic fib with target fib?

```
fib :: i:{Int|0≤i} -> {v:Int|0<v∧i≤v}
fib i
  | i≤1       = 1
  | otherwise = fib (i-1) + fib (i-2)
```

Refinement Reflection

```
fib :: i:{Int|0≤i} -> {v:Int| v=fib i ∧
   if i≤1 then fib i = 1
   else fib i = fib (i-1) + fib (i-2)
    }
```

Refinement Reflection

Step 1: Definition

Step 2: Reflection

```
fib :: i:{Int|0≤i} -> {v:Int| v=fib i ∧
    if i≤1 then fib i = 1
    else fib i = fib (i-1) + fib (i-2)
     }
```

Refinement Reflection

Step 1: Definition

Step 2: Reflection

```
fib :: i:{Int|0≤i} -> {v:Int| v=fib i ∧
   if i≤1 then fib i = 1
   else fib i = fib (i-1) + fib (i-2)
    }
```

Step 3: Application

```
fib 1 :: {v:Int| v=fib 1 ∧ fib 1 = 1}
```

Application is Type Level Computation

`fib 1`

`fib 1 = 1`

Application	Type Level Computation
`fib 1`	`fib 1 = 1`
`fib 0`	`fib 0 = 1`
`fib 2`	`fib 2 = fib 1 + fib 0`
`fib i`	?

?
```
if i≤1 then fib i = 1
else fib i = fib (i-1) + fib (i-2)
```

Application Type Level Computation

Application	Type Level Computation
`fib 1`	`fib 1 = 1`
`fib 0`	`fib 0 = 1`
`fib 2`	`fib 2 = fib 1 + fib 0`

```
if 1<i then
```

`fib i`	`fib i = fib (i-1) + fib (i-2)`
`fib (i+1)`	`fib (i+1) = fib i + fib (i-1)`

Theorem Proving

```
fibUp :: i:Nat -> {v:() | fib i ≤ fib (i+1)}
fibUp i
 | i == 0
 =   fib 0 <. fib 1
 *** QED
 | i == 1
 =   fib 1 <=. fib 1 + fib 0 <=. fib 2
 *** QED
 | otherwise
 =   fib i
 <=. fib i + fib (i-1)
 <=. fib (i+1)
 *** QED
```

Reflection for Theorem Proving

Theorems are refinement types.

Proofs are functions.

Check that functions prove theorems.

Proofs are functions.

```
fibUp :: i:Nat -> {fib i ≤ fib (i+1)}
```

Proofs are functions.

```
fibUp :: i:Nat -> {fib i ≤ fib (i+1)}
```

Let's call them!

```
     fibUp 4 :: {fib 4 ≤ fib 5}
     fibUp i :: {fib i ≤ fib (i+1)}
 fibUp (j-1) :: {fib (j-1) ≤ fib j}
```

Proofs are functions. Let's call them!

```
fibMono :: i:Nat -> j:{Nat| i < j}
        -> {fib i ≤ fib j}
fibMono i j
 | i + 1 == j
 =   fib i
 <=. fib (i+1) ? fibUp i
 ==. fib j
 *** QED
 | otherwise
 =   fib i
 <=. fib (j-1) ? fibMono i (j-1)
 <=. fib j     ? fibUp (j-1)
 *** QED
```

Proofs are functions. Let's call them!

```
fibMono :: i:Nat -> j:{Nat| i < j}
        -> {fib i ≤ fib j}
fibMono i j
 | i + 1 == j
 =   fib i
 <=. fib (i+1) ? fibUp i
 ==. fib j
 *** QED
 | otherwise
 =   fib i
 <=. fib (j-1) ? fibMono i (j-1)
 <=. fib j     ? fibUp (j-1)
 *** QED
```

Proofs are functions. Let's abstract them!

```
fibMono :: i:Nat -> j:{Nat| i < j}
        -> fib:(Nat -> Int)
        -> (k:Nat -> {fib k ≤ fib (k+1)})
        -> {fib i ≤ fib j}
fibMono i j fib fibUp
 | i + 1 == j
 =   fib i
 <=. fib (i+1) ? fibUp i
 ==. fib j
 *** QED
 | otherwise
 =   fib i
 <=. fib (j-1) ? fibMono i (j-1)
 <=. fib j     ? fibUp (j-1)
 *** QED
```

Reflection for Theorem Proving

Theorems are refinement types.

Proofs are functions.

Check that functions prove theorems.

Implementation in

LiquidHaskell

Proof by Logical Evaluation (PLE)

Idea: Unfold in you can

Proof by Logical Evaluation (PLE)

Idea: Unfold in you can

```
fibUp :: i:Nat -> {fib i ≤ fib (i+1)}
```

Proof by Logical Evaluation (PLE)

Idea: Unfold in you can

```
fib i     = ✗
fib (i+1) = ✗
```

Proof by Logical Evaluation (PLE)

Idea: Unfold in you can, using context.

if i=0, then

```
fib i     = 1
fib (i+1) = 1
```

Proof by Logical Evaluation (PLE)

Idea: Unfold in you can, using context.

if i=1, then

```
fib i     = 1
fib (i+1) = fib 1 + fib 0
```

Proof by Logical Evaluation (PLE)

Idea: Unfold in you can, using context.

if i=1, then

```
fib i     = 1
fib (i+1) = fib 1 + fib 0
fib 1     = 1
fib 0     = 1
```

Proof by Logical Evaluation (PLE)

Idea: Unfold in you can, using context.

if i>1, then

```
fib i     = fib (i-1) + fib (i-2)
fib (i+1) = fib i     + fib (i-1)
```

Proof by Logical Evaluation (PLE)

Idea: Unfold in you can, using context.

if i>1, then

```
fib i     = fib (i-1) + fib (i-2)
fib (i+1) = fib i     + fib (i-1)
fib (i-1) = ✗
fib (i-2) = ✗
```

Proof by Logical Evaluation (PLE)

```
fibUp :: i:Nat -> {v:() | fib i ≤ fib (i+1)}

fibUp i
 | i == 0
 = ()
 | i == 1
 = ()
 | otherwise
 = ()
```

Proof by Logical Evaluation (PLE)

Idea: Unfold in you can, using context.

Prop I: Termination.

Prop II: Completeness.

Application: Proof Simplification.

Proof Simplification.

$$(\textbf{Spec} + \textbf{Proof}) / \textbf{Impl} = \begin{cases} \text{x2.4, without PLE} \\ \text{x1.6, with PLE} \end{cases}$$

Evaluation

(**Spec** + **Proof**) / **Impl** x2.4 x1.6

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								
Monoid	33	50						
Functor	48	44		x4			x2	
Applicative	62	110						
Monad	63	42						
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						
n-body	163	251		x1.7			x1.3	
Par. Reducers	30	212						
Total	1176	1279	1524	148.76	1626	638	150.88	4068

Complete Verification with SMT

Refinement Reflection

Reflect function definition in result type
Function application gives type level computation

Proof by Logical Evaluation

Unfolds function definitions, if it can

LiquidHaskell

Thanks!