Automatic verification of higher order programs

CP3106 Personal Project

ZHOU Litao, Supervised by CHIN Wei Ngan

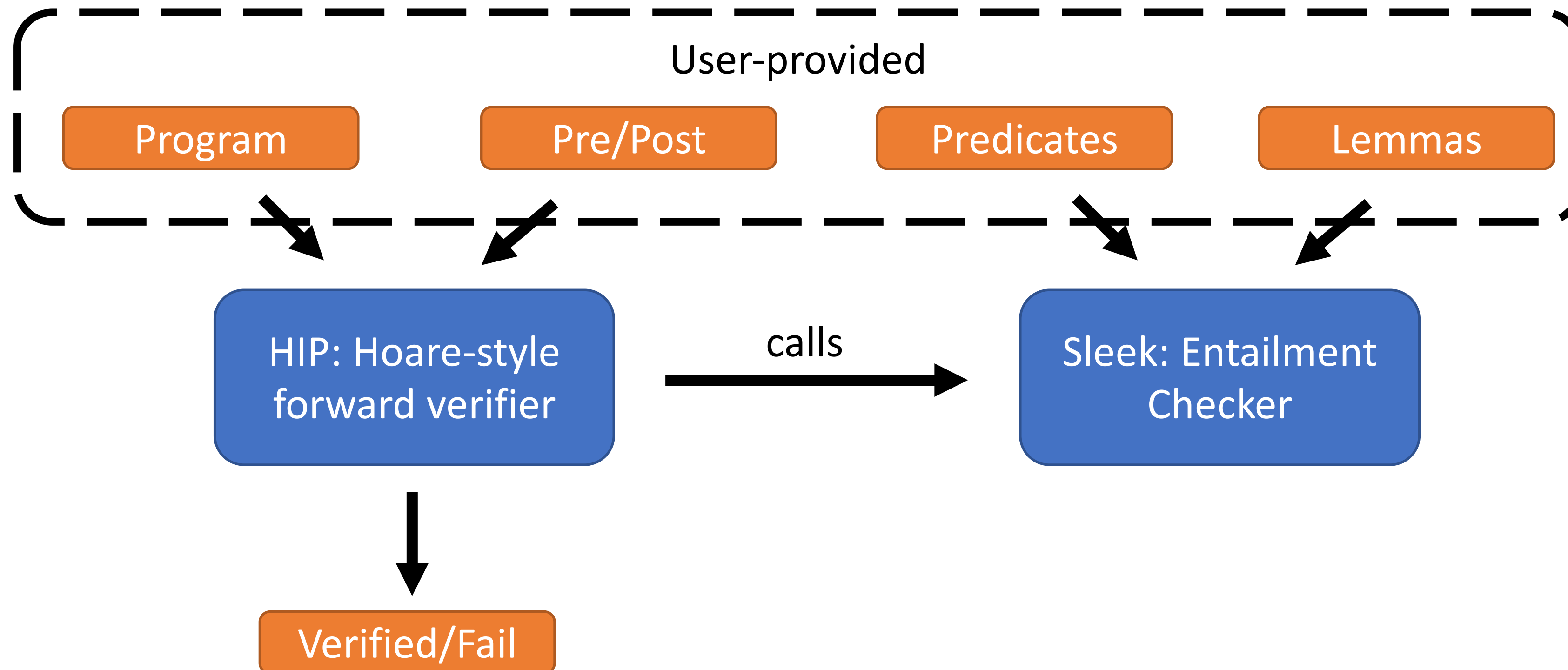
18/11/2021, NUS

Motivation: reliable programs via verification

- **Testing** is not exhaustive, bugs may not be tested

- **Verification**
  - Prove that programs behave according to formal specifications
  - Allows us to reason about all possible executions
  - Takes more efforts than testing

Background: automated verification



- HIP/SLEEK system
  - Automatic verification for <u>first order programs</u>
    - program expressions that operate on individual data elements (e.g., integers, structs, etc.).
  - How about <u>higher order programs</u>?

Background: higher order functions

- Function as parameter / return value
- Higher order functions enable
  - Function abstraction
  - Reusable code
  - Scalable programming
- How to specify/verify the correctness of a higher order function?

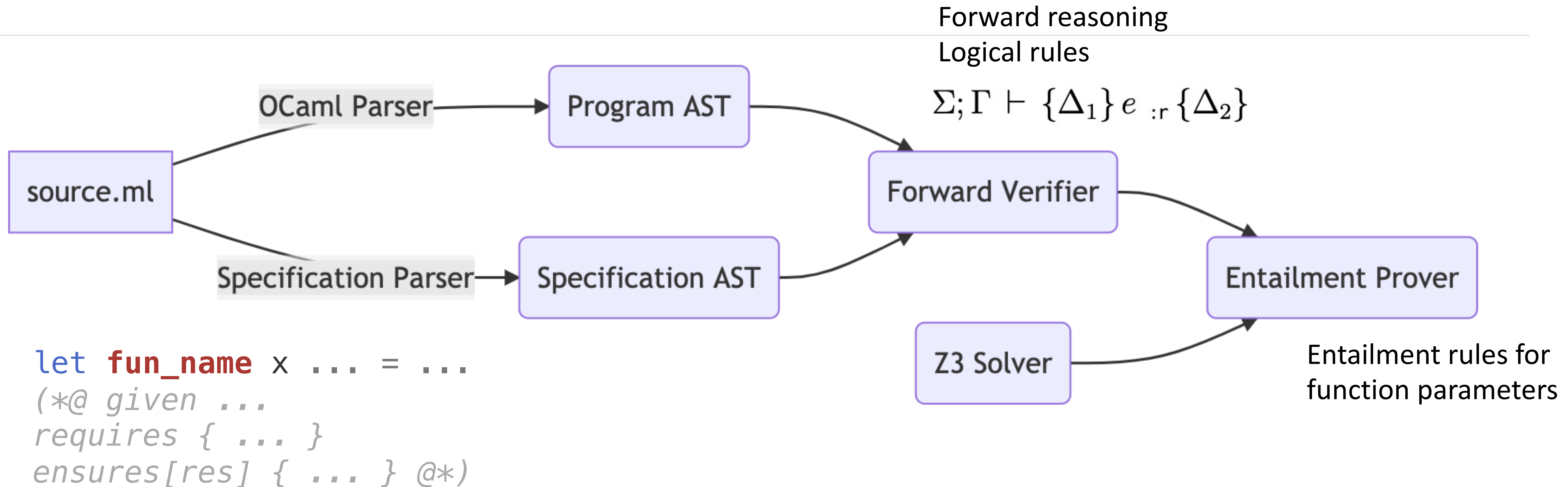
```
let rec fold_right f xs y =
match xs with
| Nil -> y
| Cons (x, xs') -> f x (fold_right f
xs' y)

let length xs =
  fold_right (fun x y -> y + 1) xs 0

let sum xs =
  fold_right (fun x y -> x + y) xs 0

let concat xs =
  fold_right (fun s1 s2 -> s1^s2) xs ""
```

Overview of the system



Forward Verifier: Hoare Logic

- *Hoare Triple:* $\Sigma; \Gamma \vdash \{\Delta_1\}\, e \;{:_r}\, \{\Delta_2\}$
  - *if we evaluate e in the initial state satisfying Δ1, then it terminates with a value, name it r, and a final state, which together satisfy Δ2*
  - Σ – variable context
  - Γ – specification context
- Method Specification:

$$\text{function specification}: \quad \mathcal{F} := \quad \mathtt{f}(\vec{\mathtt{x}}) \vDash \text{Given } \vec{y},\ \Phi_{\text{pre}} \rightarrowtail_{\mathbf{r}} \Phi_{\text{post}}$$

$$\textsc{Fv-Method}\quad \frac{\mathcal{F} = \text{Given } \vec{y}, \Phi_{\text{pre}} \rightarrowtail_{\mathbf{r}} \Phi_{\text{post}} \qquad \vec{x}, \vec{y}; P \vdash \{\Phi_{\text{pre}}\}\, e \;{:_r}\, \{\Phi_{\text{post}}\}}{\vdash \mathtt{let}\ \mathtt{f}{=}\lambda\vec{\mathtt{x}}.e\ \mathtt{where}\ \mathcal{F}}$$

Assertion language

- Assertion Syntax: allow nested function specifications

$$
\begin{array}{rrl}
\text{function specification}: & \mathcal{F} := & \mathsf{f}(\vec{\mathsf{x}}) \vDash \text{Given } \vec{y},\ \Phi_{\text{pre}} \rightarrowtail_{\text{r}} \Phi_{\text{post}} \\
\text{assertions}: & \Phi := & \bigvee ((\exists \vec{v}.\pi))^{*} \wedge \mathcal{F} \\
\text{pure predicates}: & \pi := & s_1 = s_2 \mid s_1 \leq s_2 \quad \text{(arithmetic judgements)} \\
 & \mid & \mathsf{x} :: \mathsf{cname}(\vec{y}) \quad \text{(datatype judgements)} \\
 & \mid & \pi_1 \wedge \pi_2 \mid \neg \pi \\
\text{logical expressions}: & s := & v \mid f(\vec{s}) \mid c \\
 & \mid & s_1 + s_2 \mid -s \mid s_1 \times s_2 \\
 & \mid & s_1 =? s_2 \mid s1 \leq? s_2
\end{array}
$$

- Example: specification for twice function

```
let twice f x = f (f x)
(*@ given     fpure(int,int)
requires       { true with
                  f(a) |= { true } *->:res0 {fpure(a,res0)} }
ensures[res] {EX (n:int), fpure(x,n) & fpure(n,res) } @*)
```

Forward Verifier: Logical Rules

- Function application rule:
  - First evaluate function body
  - Next evaluate argument
  - Check the current state satisfy precondition
  - Do the substitution

FV-APP-FULL

$$\frac{\Sigma;\Gamma \vdash \{\Delta_0\}\ f\ _{:\mathsf{g}}\ \{\Delta_1\} \qquad \Sigma,\mathsf{g};\Gamma \vdash \{\Delta_1\}\ x\ _{:\mathsf{v}}\ \{\Delta_2\} \quad \mathsf{g}(\mathsf{a}) \vDash \text{Given } \vec{y}, \Phi_{\text{pre}} \rightarrowtail_{\mathtt{r}} \Phi_{\text{post}} \in \Gamma \cup \mathit{Spec}(\Delta_2) \quad \Delta_2 \vdash \Phi_{\text{pre}}(\vec{y})[\mathtt{v}/\mathtt{a}] \qquad \Phi_{\text{post}}(\vec{y})[\mathtt{v}/\mathtt{a}, \mathtt{res}/\mathtt{r}] \vdash \Delta_3}{\Sigma;\Gamma \vdash \{\Delta_0\}\ f\ x\ _{:\mathsf{res}}\ \{\Delta_3\}}$$

FV-VAR

$$\frac{\mathsf{v} \in \Sigma \text{ or } \mathsf{v}(\vec{x}) \vDash \mathcal{F} \in \Gamma}{\Sigma;\Gamma \vdash \{\Delta\}\ \mathsf{v}\ _{:\mathsf{res}}\ \{\Delta \wedge \mathsf{res} = \mathsf{v}\}}$$

```
let twice f x = f (f x)
(*@ given     fpure(int,int)
requires      { true &
 f(a) |=        { true }
     *->:res0 {fpure(a,res0)} }
```

```
f(a) |= { true } *->:res0 {fpure(a,res0)}
{ r1=f } f:r2 { r1=f & r2=f }
{ r1=f & r2=f } x:r3 { r1=f & r2=f & r3=x }
-------------------------------------------------------------------------
{ true } f:r1 { r1=f }   { r1=f } (f x):r4 { r1=f & r2=f & r3=x & fpure(r3,r4)}
------------------------------------------------------------------------------
    { true } f (f x):r5 { r1=f & r2=f & r3=x & fpure(r3,r4) & fpure(r4,r5)}
```

```
ensures[res] {EX (n:int), fpure(x,n) & fpure(n,res) } @*)
```

Instantiation of abstract predicates

- Function application rule:
  - First evaluate function body
  - Next evaluate argument
  - Check the current state satisfy precondition (instantiate the GIVEN clause)
  - Do the substitution

$$\textsc{Fv-App-Full}\;\frac{\Sigma;\Gamma \vdash \{\Delta_0\}\, f\;_{:\mathsf{g}}\,\{\Delta_1\} \qquad \Sigma,\mathsf{g};\Gamma \vdash \{\Delta_1\}\, x\;_{:\mathsf{v}}\,\{\Delta_2\} \\ \mathsf{g}(\mathsf{a}) \vDash \text{Given } \vec{y}, \Phi_{\text{pre}} \rightarrowtail_{\mathtt{r}} \Phi_{\text{post}} \in \Gamma \cup \mathit{Spec}(\Delta_2) \\ \Delta_2 \vdash \Phi_{\text{pre}}(\vec{y})[\mathtt{v}/\mathtt{a}] \qquad \Phi_{\text{post}}(\vec{y})[\mathtt{v}/\mathtt{a}, \mathtt{res}/\mathtt{r}] \vdash \Delta_3}{\Sigma;\Gamma \vdash \{\Delta_0\}\, f\; x\;_{:\mathtt{res}}\,\{\Delta_3\}}$$

```
let incr x = x + 1

let incr_twice x2 = twice incr x2
(*@
requires { true }
ensures[r] { r = x2 + 2 }
@*)
```

```
twice(f,x) |= Given fpure,
   { true & f(a) |= { true } *->:r {fpure(a,r)} }
   *->:res {EX n, fpure(x,n) & fpure(n,res) }

incr(x) |= { true } *->:res {res = x + 1}
```

Annotations (replacing struck-out terms): fpure(a,r) → r = a + 1; fpure(x,n) → n = x + 1; fpure(n,res) → res = n + 1

```
{ true }
twice incr x2 : r
{ EX n, n = x2 + 1 & r = n + 1  }
```

Function as return values

- Partial application rule:
  - Evaluate the function body
  - Evaluate the applied arguments
  - Create a new specification for the partially applied function

```
let div y x =  (x / y)
(*@ requires { ~ y = 0 }
    ensures [r]{ r * y = x } @*)

let div_by_one = (div 1)
(*@ requires { true }
   ensures [f]{ true with
                 f(x) |= { true } *->:m { m * 1 = x } } @*)
```

$$\text{Fv-App-Par}\ \frac{\Sigma;\Gamma\vdash\{\Delta_0\}\,f\ {:_{\mathsf{g}}}\,\{\Delta_1\}\qquad \Sigma,\mathsf{g};\Gamma\vdash\{\Delta_1\}\,x\ {:_{\mathsf{v}}}\,\{\Delta_2\}\qquad \mathsf{g}(\mathsf{a},\tilde{\mathsf{b}})\vDash\mathcal{F}\in\Gamma}{\Sigma;\Gamma\vdash\{\Delta_0\}\,f\ x\ {:_{\mathsf{h}}}\left\{\Delta_2\wedge\mathsf{h}(\tilde{\mathsf{b}})\vDash\mathcal{F}[\mathtt{v}/\mathsf{a}]\right\}}$$

```
{ true }   div 1 : r
{ true with
    r(x) |= { ~ 1 = 0 } *->:m { m * 1 = x } }
```

Entailment Checking (with specifications)

- Pure predicate part: encode into SMT formulae
- Specification part: function **subsumption** rule

```
{ true with
    f(x) |= { true }
          *->:m { m * 1 = x } }
```

<:

```
{ true with
    r(x) |= { ~ 1 = 0 }
          *->:m { m * 1 = x } }
```

  - The user expected specification is a **refinement** of the inferred specification

FUN-SUB

$$\frac{\Sigma \vdash \forall \vec{y}_2, \forall \vec{x}_2, \pi \land \Phi_2 \Rightarrow \exists \vec{y}_1, \forall \vec{x}_1, (\Phi_1 \land (\forall \mathbf{r}, \Psi_1 \Rightarrow \Psi_2))}{\mathsf{f}(\vec{x}) \vDash \text{Given } \vec{y}_1, \Phi_1 \rightarrowtail_{\mathbf{r}} \Psi_1 <:_{\pi,\Sigma} \mathsf{f}(\vec{x}) \vDash \text{Given } \vec{y}_2, \Phi_2 \rightarrowtail_{\mathbf{r}} \Psi_2}$$

```
Forall x, ~ 1 = 0 -> ( true & ( m * 1 = x -> m * 1 = x ))
```

Inductive predicates

- Inductive predicates allow us to describe richer properties, especially for data structures
- Each predicate definition is the disjunctive normal form of pure predicates
- E.g. `LL(n:int, l:list)`
  - `l` is a list of length `n`

```
type list = Nil
| Cons of (int * list)

(*@ pred LL(n:int,l:list)
|= l::Nil<> & n=0
or EX i q, l::Cons<i,q> & LL(n-1, q) @*)
```

```
let rec length lst =
(*@ given (n:int)
requires { LL(n,lst) }
ensures[res] { res = n } @*)
match lst with
| Nil -> 0
| Cons (x, xs) -> 1 + length xs
```

Verify with inductive predicates

```
let rec length lst =
(* LL(n,lst) *)
(* lst::Nil & n=0
 | lst::Cons(i,q) & LL(n-1,q) *)
match lst with
| Nil ->
(* lst::Nil & n=0 *)
 0
(* lst::Nil & n=0 & res=0 *)
                      => res = n

| Cons (x, xs) ->
(* lst::Cons(x,xs) & LL(n-1,xs) *)
1 + length xs
(* lst::Cons(x,xs) & LL(n-1,xs) &
   res=1+r1 & r1=n-1 *)
                      => res = n
```

- Proceed the verification by fold/unfold the inductive predicates
- In implementation, we encode the definition of predicates as an extra constraint to the SMT solver

Verify higher order functions on data structures

- Idea: **Reflect** the behavior of higher order function into predicates
  - `LL_foldr(y,r,xs,fpure):` `r` is the result of folding `xs` from `y` using `fpure`

```
type list = Nil
| Cons of (int * list)
(*@ pred LL_foldr(y,r,xs:list,fpure:int->int->int)
|= xs::Nil<> & y=r
or EX xs' x z, xs::Cons<x,xs'> &  LL_foldl(y,z,xs',fpure) &
fpure(x,z) = r @*)

let rec fold_right f xs y =
(*@ given (fpure(int,int):int), (r:int)
  requires { LL_foldr(y,r,xs,fpure)
             with f(x,y) |= {true} *->:r {r = fpure(x,y)} }
  ensures[res] { res = r } @*)
match xs with
| Nil -> y
| Cons (x, xs') -> f x (fold_right f xs' y)
```

Verify applications of higher order functions

- Instantiate the abstract predicate `fpure(x,y)` with `y+1`
- We are left to prove

```
LL(n,xs) ⇒ LL_foldr(0,n,xs,fpure(x,y)=y+1)
```

```
xs::Nil & n=0 || xs::Cons<p,q> & LL(n-1,q)
=> xs::Nil & n=0 || xs::Cons<p,q> & LL_foldr(0,z,xs',fpure) & z+1 = n
```

```
let rec fold_right f xs y = ...
(*@ given (fpure(int,int):int), (r:int)
  requires { LL_foldr(y,r,xs,fpure) with
f(x,y) |= {true} *->:r {r = fpure(x,y)} }
  ensures[res] { res = r } @*)
```

```
let fold_length x y = y + 1
(*@ requires { true }
    ensures[res] { res=y+1 } @*)
let length xs = fold_right fold_length xs 0
(*@ given (n:int)
    requires { LL(n,xs) }
    ensures[res] { res=n } @*)
```

Summary

Contributions

- A logical system for verifying higher order functions
- A prototype system of forward verifier and entailment checker.
- Various verified higher order function examples

Future Work

- Prove soundness of the logical system
- Extend the logical system to support stateful program with effects
- Make instance inference more automatic

Thank you for your attention

Questions?