Program Semantics with Interaction Trees

Ben van Wijngaarden

Radboud University

January 19, 2026

The Papers

- Paper 1:

  Interaction Trees

  Representing Recursive and Impure Programs in Rocq

  Xia et al.

  POPL 2020

- Paper 2:

  Program Logics à la Carte

  Vistrup, Sammler & Jung

  POPL 2025

Effectful programs

Effects are everywhere

- I/O
- State
- Failure
- Nondeterminism
- Concurrency

Effectful programs

Effects are everywhere

- I/O
- State
- Failure
- Nondeterminism
- Concurrency

An Example:

```
let x = input in
    if x == 0
        then fail
        else put (1/x); output (1/x)
```

Semantics for effectful programs

Program semantics

- Operational semantics
  - describe execution of programs $e_1 \to e_2 \to \ldots \to e_n$

Semantics for effectful programs

Program semantics

- Operational semantics
  - describe execution of programs $e_1 \to e_2 \to \ldots \to e_n$
  - Features:
    - (+) Intuitive
    - (+) Executable
    - (-) Not composable
    - (-) No Equational reasoning

Semantics for effectful programs

Program semantics

- Operational semantics
  - describe execution of programs $e_1 \to e_2 \to \ldots \to e_n$
  - Features:
    - (+) Intuitive
    - (+) Executable
    - (-) Not composable
    - (-) No Equational reasoning
- Denotational semantics
  - describe programs as mathematical objects $[\![e]\!]$ denotes e

Semantics for effectful programs

Program semantics

- Operational semantics
  - describe execution of programs $e_1 \to e_2 \to \ldots \to e_n$
  - Features:
    - (+) Intuitive
    - (+) Executable
    - (-) Not composable
    - (-) No Equational reasoning
- Denotational semantics
  - describe programs as mathematical objects $[\![e]\!]$ denotes e
  - Features:
    - (+) Composable
    - (+) Equational reasoning
    - (-) Need a Mathematical domain

Semantics for effectful programs

Program semantics

- Operational semantics
  - describe execution of programs $e_1 \to e_2 \to \ldots \to e_n$
  - Features:
    - (+) Intuitive
    - (+) Executable
    - (-) Not composable
    - (-) No Equational reasoning
- Denotational semantics
  - describe programs as mathematical objects $[\![e]\!]$ denotes e
  - Features:
    - (+) Composable
    - (+) Equational reasoning
    - (-) Need a Mathematical domain
- Axiomatic Semantics
  - describing programs with logical rules $\{P\}\ e\ \{Q\}$

Semantics for effectful programs

Program semantics

- Operational semantics
  - describe execution of programs $e_1 \to e_2 \to \ldots \to e_n$
  - Features:
    - (+) Intuitive
    - (+) Executable
    - (-) Not composable
    - (-) No Equational reasoning
- Denotational semantics
  - describe programs as mathematical objects $[\![e]\!]$ denotes e
  - Features:
    - (+) Composable
    - (+) Equational reasoning
    - (-) Need a Mathematical domain
- Axiomatic Semantics
  - describing programs with logical rules $\{P\}\ e\ \{Q\}$
  - Features:
    - (+) Proving general properties
    - (+) Program verification
    - (-) Not Executable
    - Usually requires a soundness proof w.r.t a (different) language semantic model

First paper

Interaction Trees

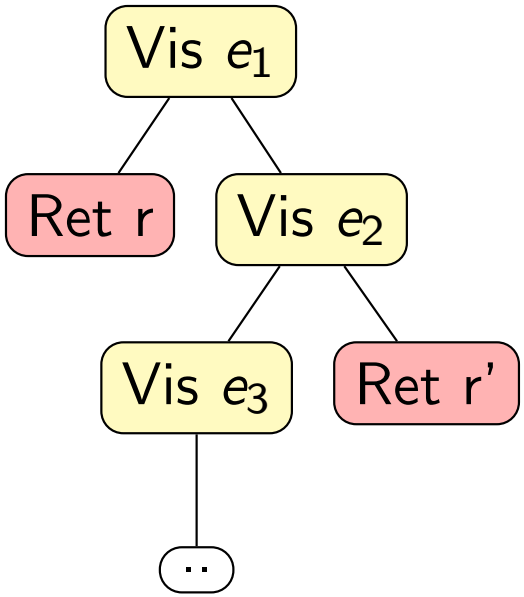
Representing Recursive and Impure Programs in Rocq

Xia et al.

2020

Interaction Trees (1)

ITrees



- Vis = visible effect node (Impure)
- Ret = return node (Pure)

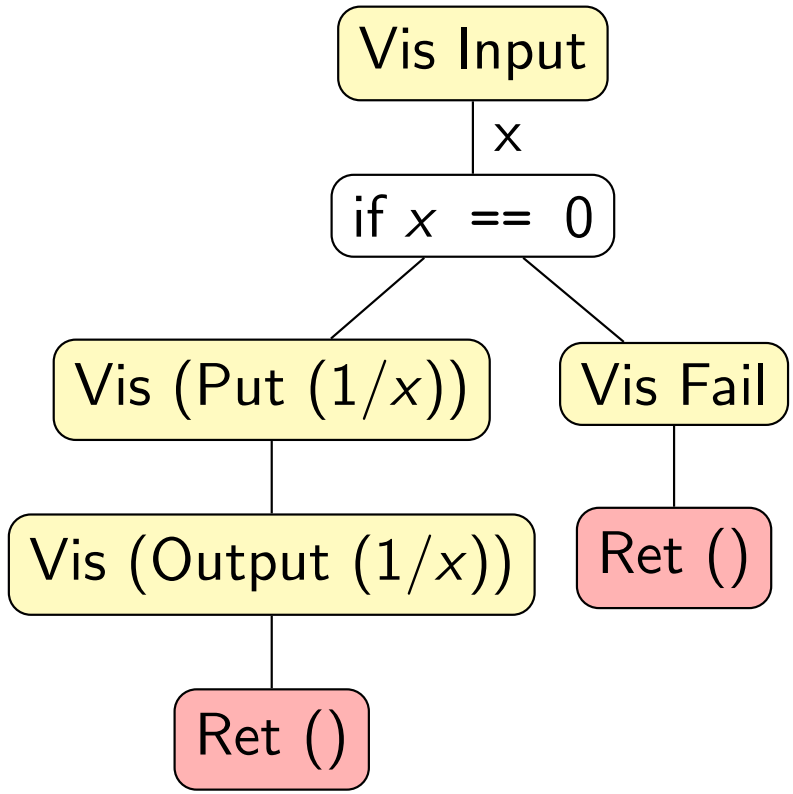
Interaction Trees (2)

Example from earlier

Program

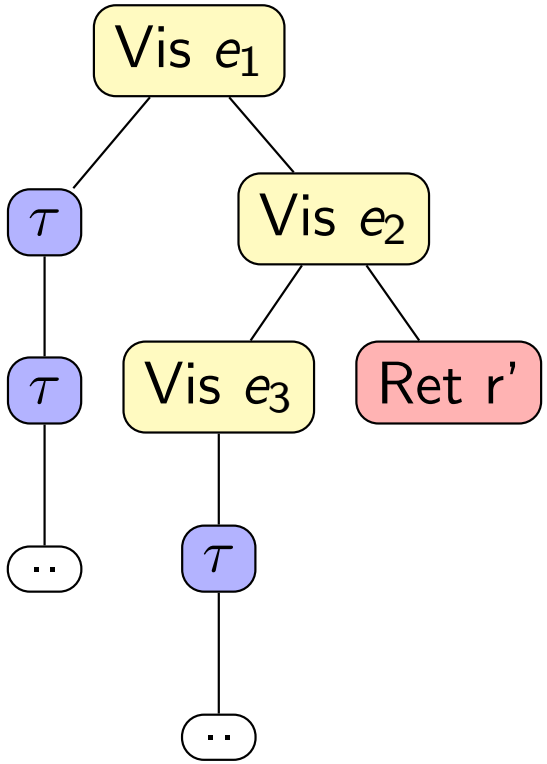
```
let x = input in
    if x == 0
    then fail
    else put (1/x);
         output (1/x)
```

Diagram



Interaction Trees (3)

Representing diverging computations



- Vis = visible effect node (Impure)
- Ret = return node (Pure)
- Tau = silent step (Progress)

Induction vs Coinduction (in Rocq)

Induction (Fixpoint)

- Finite computation
- Destructs inductive data
- Must terminate

Syntactic condition:

- Recursive calls on a structural subterm

Induction vs Coinduction (in Rocq)

Induction (Fixpoint)

- Finite computation
- Destructs inductive data
- Must terminate

Syntactic condition:

- Recursive calls on a structural subterm

Coinduction (CoFixpoint)

- (possibly) Infinite computation
- Constructs coinductive data
- Must be productive

Syntactic condition:

- Recursive calls must be guarded by a constructor

Interaction Trees (4)

The simplest example: Infinite loop

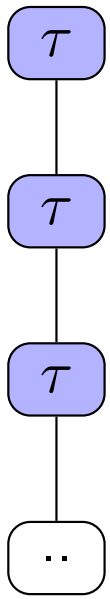
Program

```
loop {}
```

Denotation

```
CoFixpoint loop := Tau loop
```

Diagram



Interaction Trees (5)

ITree definition:

```
CoInductive itree (E: Type → Type) (R: Type) : Type :=
| Ret (r: R)
| Tau (t: itree E R)
| Vis {A: Type} (e : E A) (k : A → itree E R).
```

- `Ret r`, return a value of type `R`
- `Tau t`, silent step

Interaction Trees (5)

ITree definition:

```
CoInductive itree (E: Type → Type) (R: Type) : Type :=
| Ret (r: R)
| Tau (t: itree E R)
| Vis {A: Type} (e : E A) (k : A → itree E R).
```

- `Ret r`, return a value of type `R`
- `Tau t`, silent step
  - diverging computations
  - Guarding recursive calls, due to coinductive definition

Interaction Trees (5)

ITree definition:

```
CoInductive itree (E: Type → Type) (R: Type) : Type :=
| Ret (r: R)
| Tau (t: itree E R)
| Vis {A: Type} (e : E A) (k : A → itree E R).
```

- `Ret r`, return a value of type `R`
- `Tau t`, silent step
- `Vis e k`, execute effect `e` and continue with continuation `k`

Interaction Trees (5)

ITree definition:

```
CoInductive itree (E: Type → Type) (R: Type) : Type :=
| Ret (r: R)
| Tau (t: itree E R)
| Vis {A: Type} (e : E A) (k : A → itree E R).
```

- `Ret r`, return a value of type `R`
- `Tau t`, silent step
- `Vis e k`, execute effect `e` and continue with continuation `k`
  - `E` = type constructor, parameterized by the return type of the effect

Interaction Trees (5)

ITree definition:

```
CoInductive itree (E: Type → Type) (R: Type) : Type :=
| Ret (r: R)
| Tau (t: itree E R)
| Vis {A: Type} (e : E A) (k : A → itree E R).
```

- `Ret r`, return a value of type `R`
- `Tau t`, silent step
- `Vis e k`, execute effect `e` and continue with continuation `k`
  - `E` = type constructor, parameterized by the return type of the effect
  - `e : E A`, here `A` is the return type of effect `E`.

Interaction Trees (5)

ITree definition:

```
CoInductive itree (E: Type → Type) (R: Type) : Type :=
| Ret (r: R)
| Tau (t: itree E R)
| Vis {A: Type} (e : E A) (k : A → itree E R).
```

- `Ret r`, return a value of type `R`
- `Tau t`, silent step
- `Vis e k`, execute effect `e` and continue with continuation `k`
  - `E` = type constructor, parameterized by the return type of the effect
  - `e : E A`, here `A` is the return type of effect `E`.
    - Example:

```
Inductive IO : Type → Type :=
| Input : IO nat
| Output : nat → IO unit.
```

Interaction Trees (6)

Program:

```
let x = input in
    if x == 0
        then fail
        else put (1/x); output (1/x)
```

Denotation:

```
Vis Input (fun x ⇒
    if x == 0
        then Vis Fail (fun _ ⇒ Ret ())
        else Vis (Put (1/x)) (fun _ ⇒
                Vis (Output (1/x)) (fun _ ⇒ Ret ())))
```

Interaction Trees (6)

Program:

```
let x = input in
    if x == 0
        then fail
        else put (1/x); output (1/x)
```

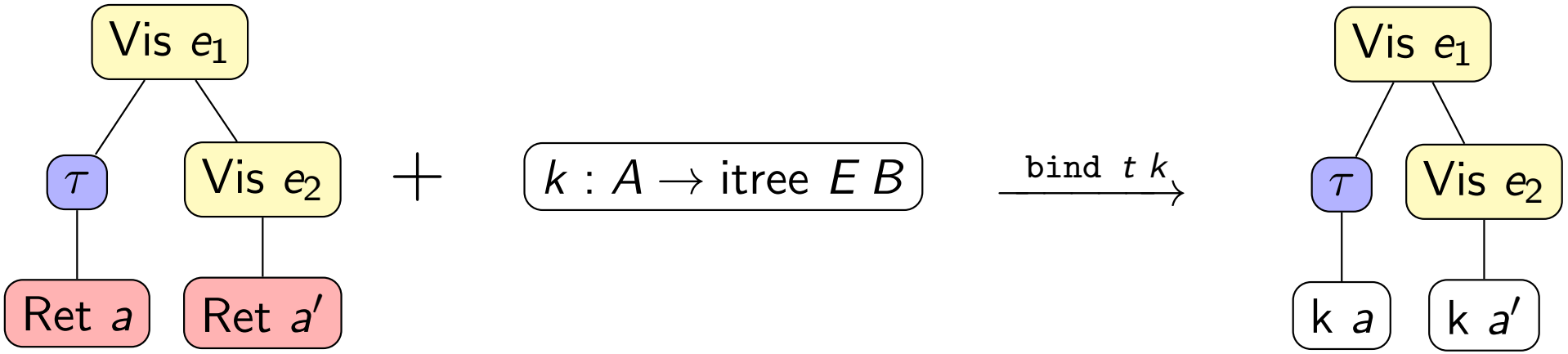
Denotation:

```
trigger e = Vis e (fun x ⇒ Ret x)

Vis Input (fun x ⇒
    if x == 0
        then trigger Fail
        else Vis (Put (1/x)) (fun _ ⇒
                trigger (Output (1/x))))
```

ITrees are Monads

itree E is a monad for every E.

- `ret : A → itree E A`
  - `ret x := Ret x`
- `bind : itree E A → (A → itree E B) → itree E B`
  - `x ← t1 ; t2 := (bind t1 (fun x ⇒ t2))`



Interaction Trees (6)

Program:

```
let x = input in
    if x == 0
        then fail
        else put (1/x); output (1/x)
```

Denotation:

```
x ← trigger Input ;
if x == 0
    then trigger Fail
    else _ ← trigger (Put (1/x)) ;
           trigger (Output (1/x))

trigger e := Vis e (fun x ⇒ x)
x ← t1 ; t2 := bind t1 (fun x ⇒ t2)
```

Interaction Trees (7)

What can we do with ITrees

- Representing programs using ITrees
  - Effectful programs
  - Diverging programs
  - Recursive programs

Interaction Trees (7)

What can we do with ITrees

- Representing programs using ITrees
  - Effectful programs
  - Diverging programs
  - Recursive programs
- Program Equivalence
  - by equational reasoning on ITrees

Interaction Trees (7)

What can we do with ITrees

- Representing programs using ITrees
  - Effectful programs
  - Diverging programs
  - Recursive programs
- Program Equivalence
  - by equational reasoning on ITrees
- Interpreting ITrees
  - by writing interpreters for effects

Interaction Trees (7)

What can we do with ITrees

- Representing programs using ITrees
  - Effectful programs
  - Diverging programs
  - Recursive programs
- Program Equivalence
  - by equational reasoning on ITrees
- Interpreting ITrees
  - by writing interpreters for effects
- ITree Building Blocks
  - Combinators to make denotation easier
  - Equations on combinators for equational reasoning

Interaction Trees (7)

What can we do with ITrees

- Representing programs using ITrees
  - Effectful programs
  - Diverging programs
  - Recursive programs
- Program Equivalence
  - by equational reasoning on ITrees
- Interpreting ITrees
  - by writing interpreters for effects
- ITree Building Blocks
  - Combinators to make denotation easier
  - Equations on combinators for equational reasoning

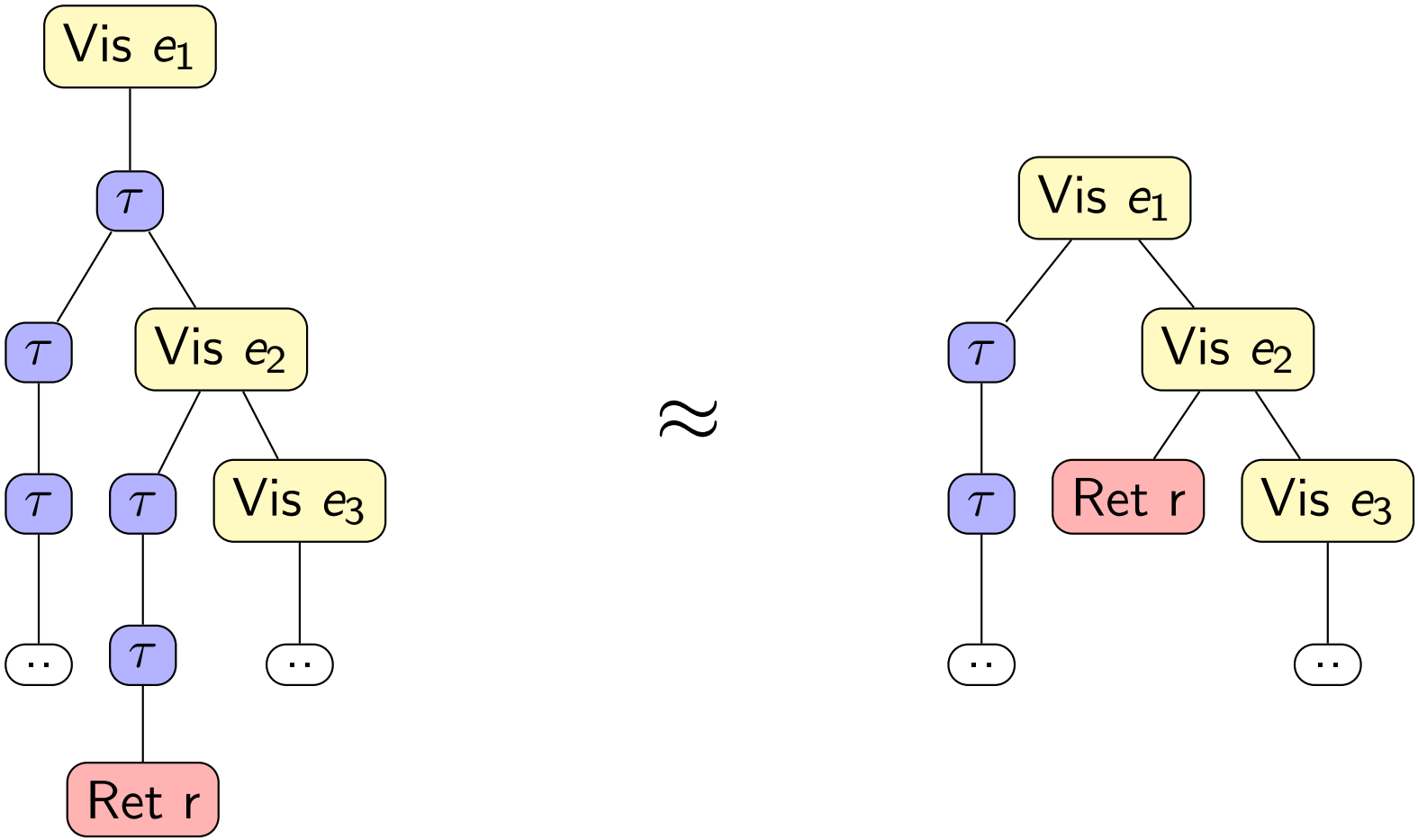
The paper contributes all of the above in a Rocq library

Interaction Trees (7)

What can we do with ITrees

- Representing programs using ITrees
  - *Effectful programs*
  - *Diverging programs*
  - Recursive programs
- **Program Equivalence**
  - by equational reasoning on ITrees
- Interpreting ITrees
  - by writing interpreters for effects
- ITree Building Blocks
  - Combinators to make denotation easier
  - Equations on combinators for equational reasoning

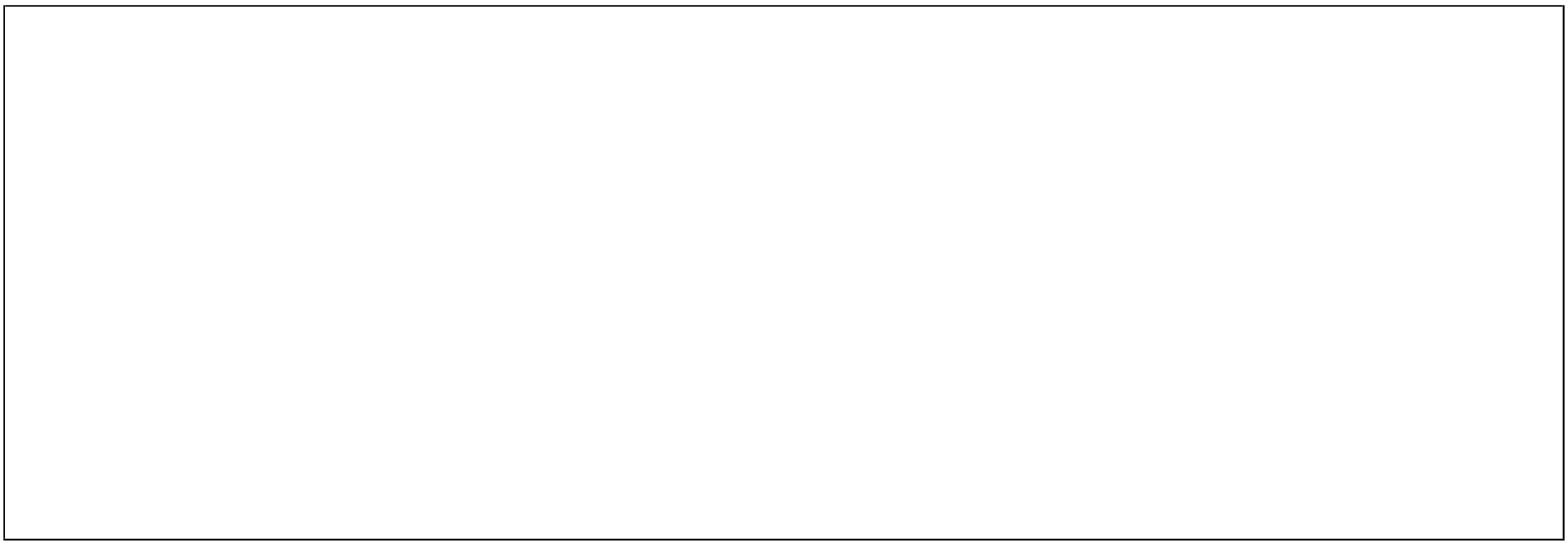
Equivalence (1)



Equivalence (2)

Equivalence up to tau: $\approx_{\text{sim}}$

$\approx_{\text{sim}}$: `itree E A` $\rightarrow$ `itree E A` $\rightarrow$ `Prop` (`inductive`)



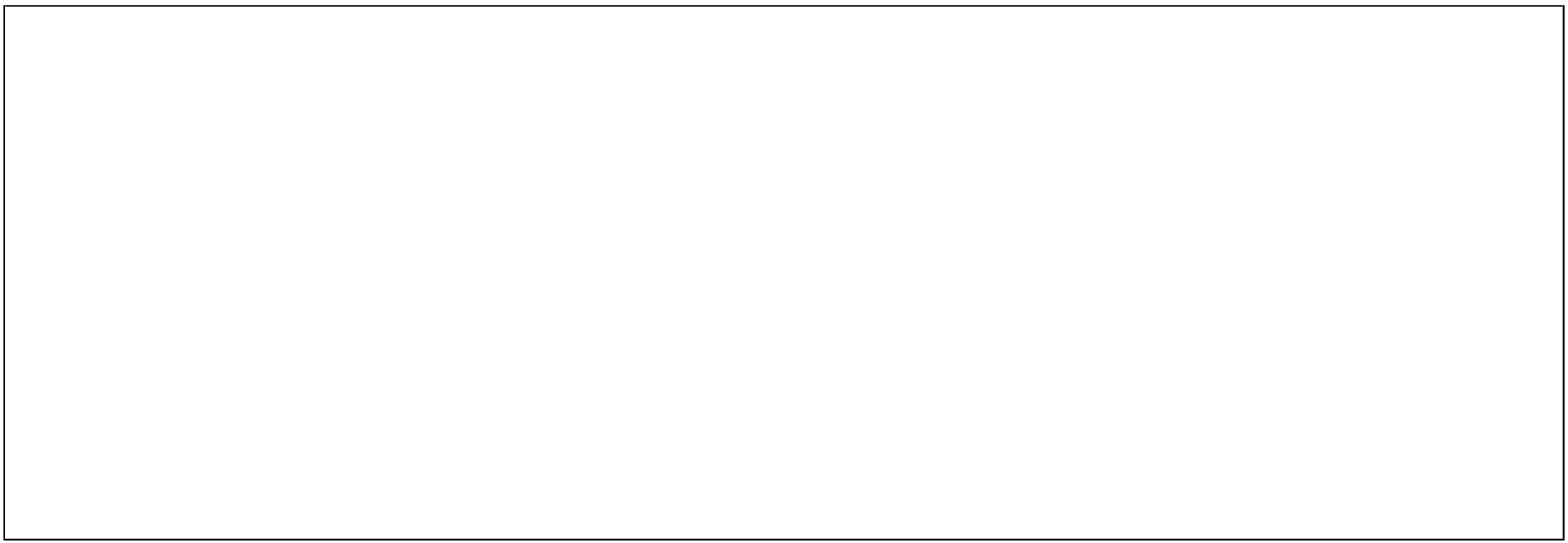
Equivalence (2)

Equivalence up to tau: $\approx_{\mathsf{sim}}$

$\approx_{\mathsf{sim}}$: `itree E A` → `itree E A` → `Prop` (`inductive`)

- Paramaterized by:

  `sim` : `itree E A` → `itree E A` → `Prop` (`coinductive`)



Equivalence (2)

Equivalence up to tau: $\approx_{\text{sim}}$

$\approx_{\text{sim}}$: `itree E A` → `itree E A` → `Prop` (`inductive`)

- Paramaterized by:

  `sim` : `itree E A` → `itree E A` → `Prop` (`coinductive`)

$$\frac{a = a'}{\text{Ret } a \approx_{\text{sim}} \text{Ret } a'}\ [\text{EqRet}]$$

Equivalence (2)

Equivalence up to tau: $\approx_{\text{sim}}$

$\approx_{\text{sim}}$: `itree E A` → `itree E A` → `Prop` (`inductive`)

- Paramaterized by:

  `sim` : `itree E A` → `itree E A` → `Prop` (`coinductive`)

$$\frac{\text{a} = \text{a'}}{\text{Ret a} \approx_{\text{sim}} \text{Ret a'}}\ [\text{EqRet}] \qquad \frac{\forall v, \mathit{sim}\ (k_1\, v)\ (k_2\, v)}{\text{Vis e } k_1 \approx_{\text{sim}} \text{Vis e } k_2}\ [\text{EqVis}]$$

$$\frac{\mathit{sim}\ t_1\ t_2}{\text{Tau } t_1 \approx_{\text{sim}} \text{Tau } t_2}\ [\text{EqTau}]$$

Equivalence (2)

Equivalence up to tau: $\approx_{\text{sim}}$

$\approx_{\text{sim}}$: `itree E A → itree E A → Prop (inductive)`

- Paramaterized by:

  `sim : itree E A → itree E A → Prop (coinductive)`

$$\frac{a = a'}{\text{Ret } a \approx_{\text{sim}} \text{Ret } a'}\ [\text{EqRet}] \qquad \frac{\forall v, \mathit{sim}\ (k_1\, v)\ (k_2\, v)}{\text{Vis e } k_1 \approx_{\text{sim}} \text{Vis e } k_2}\ [\text{EqVis}]$$

$$\frac{\mathit{sim}\ t_1\ t_2}{\text{Tau } t_1 \approx_{\text{sim}} \text{Tau } t_2}\ [\text{EqTau}] \qquad \frac{t_1 \approx_{\text{sim}} t_2}{\text{Tau } t_1 \approx_{\text{sim}} t_2}\ [\text{EqTauL}]$$

$$\frac{t_1 \approx_{\text{sim}} t_2}{t_1 \approx_{\text{sim}} \text{Tau } t_2}\ [\text{EqTauR}]$$

Equivalence (2)

Equivalence up to tau: $\approx_{\text{sim}}$

$\approx_{\text{sim}}$: `itree E A` → `itree E A` → `Prop` (`inductive`)

- Paramaterized by:

  `sim` : `itree E A` → `itree E A` → `Prop` (`coinductive`)

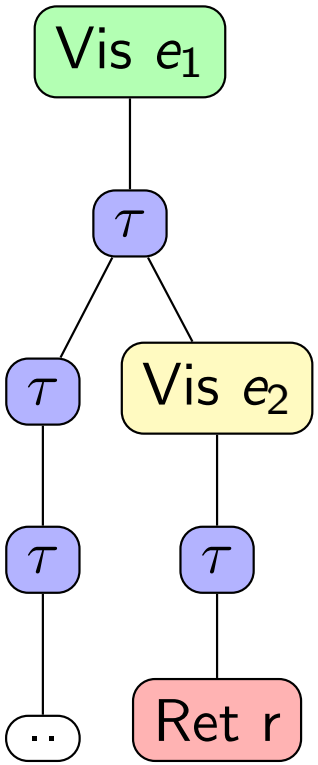
$$\frac{a = a'}{\text{Ret } a \approx_{\text{sim}} \text{Ret } a'}\ [\text{EqRet}] \qquad \frac{\forall v, \mathit{sim}\ (k_1\, v)\ (k_2\, v)}{\text{Vis e } k_1 \approx_{\text{sim}} \text{Vis e } k_2}\ [\text{EqVis}]$$

$$\frac{\mathit{sim}\ t_1\ t_2}{\text{Tau } t_1 \approx_{\text{sim}} \text{Tau } t_2}\ [\text{EqTau}] \qquad \frac{t_1 \approx_{\text{sim}} t_2}{\text{Tau } t_1 \approx_{\text{sim}} t_2}\ [\text{EqTauL}]$$

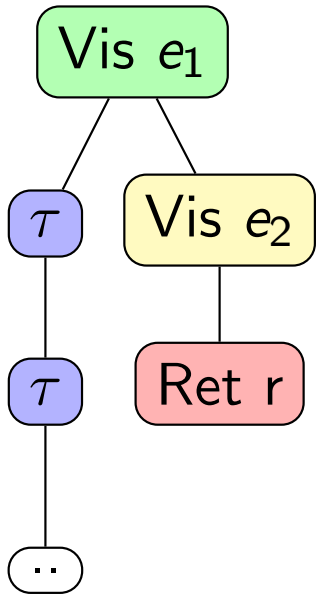
$$\frac{t_1 \approx_{\text{sim}} t_2}{t_1 \approx_{\text{sim}} \text{Tau } t_2}\ [\text{EqTauR}] \qquad \boxed{\frac{t_1 \approx_{\text{sim}} t_2}{\text{sim } t_1\ t_2}\ [\text{sim}_c]}$$

Equivalence (3)

$$\frac{\forall v, \mathit{sim}\ (k_1\, v)\ (k_2\, v)}{\text{Vis e } k_1 \approx_{\text{sim}} \text{Vis e } k_2}\ [\text{EqVis}]$$

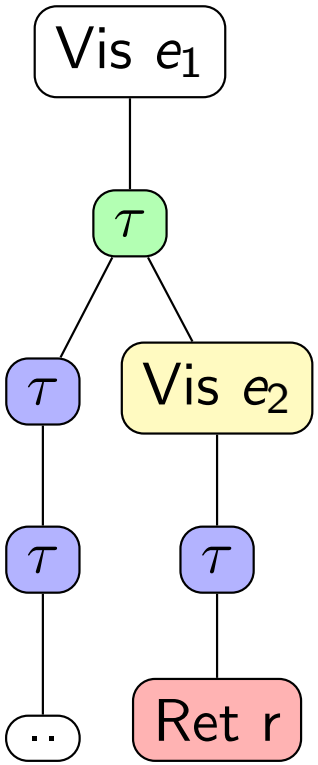


$\approx$

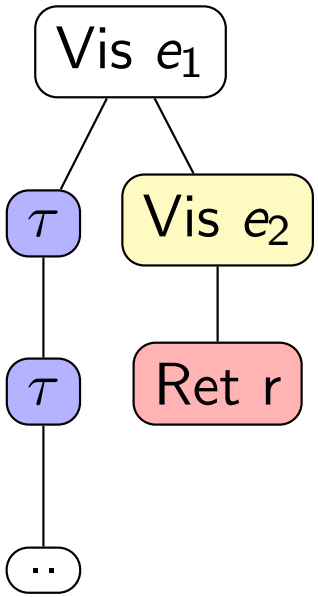


Equivalence (4)

$$\frac{t_1 \approx_{\mathsf{sim}} t_2}{\mathsf{Tau}\ t_1 \approx_{\mathsf{sim}} t_2}\ [\mathsf{EqTauL}]$$



$\approx$



Equivalence (5)

Depending on v (left branch):

$$\frac{\mathit{sim}\ t_1\ t_2}{\mathsf{Tau}\ t_1 \approx_{\mathsf{sim}} \mathsf{Tau}\ t_2}\ [\mathsf{EqTau}]$$

Overview

What we can do with ITrees

- Representing programs using ITrees
  - *Effectful programs*
  - *Diverging programs*
  - Recursive programs
- *Program Equivalence*
  - by equational reasoning on ITrees
- Interpreting ITrees
  - by writing interpreters for effects
- ITree Building Blocks
  - Combinators to make denotation easier
  - Equations on combinators for equational reasoning

Overview

What we can do with ITrees

- Representing programs using ITrees
  - *Effectful programs*
  - *Diverging programs*
  - Recursive programs
- *Program Equivalence*
  - by equational reasoning on ITrees
- **Interpreting ITrees**
  - by writing interpreters for effects
- ITree Building Blocks
  - Combinators to make denotation easier
  - Equations on combinators for equational reasoning

Interpreting ITrees (1)

Interpreting/executing ITrees

Interpreting ITrees (1)

Interpreting/executing ITrees

- Write an interpreter for each effect

Interpreting ITrees (1)

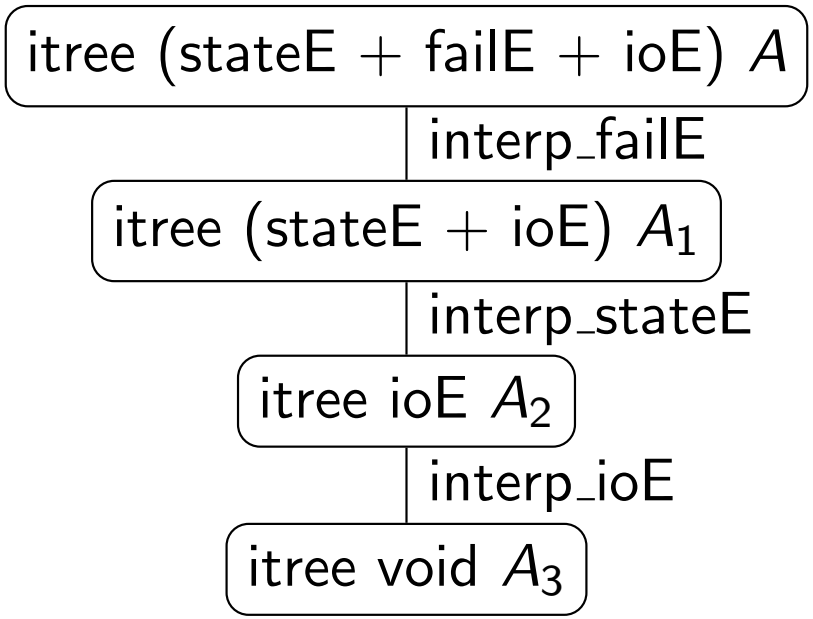
Interpreting/executing ITrees

- Write an interpreter for each effect
- Modular: Easy to extend with new effects

Interpreting ITrees (1)

Interpreting/executing ITrees

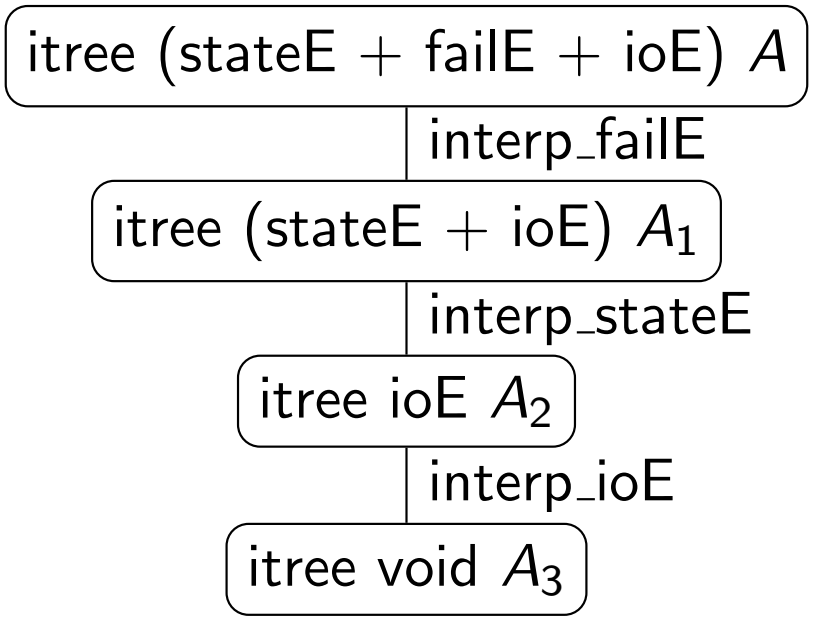
- Write an interpreter for each effect
- Modular: Easy to extend with new effects



Interpreting ITrees (1)

Interpreting/executing ITrees

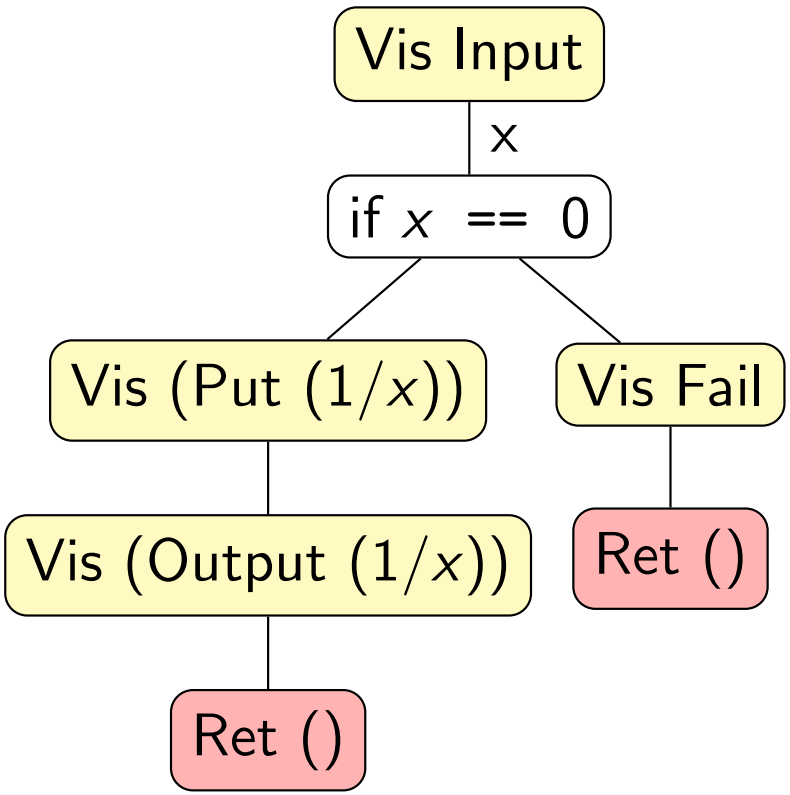
- Write an interpreter for each effect
- Modular: Easy to extend with new effects



- ITree without effects: t : itree void A
  - $t \approx_{\text{sim}}$ Ret a
  - **OR**: t is an infinte chain of Tau nodes

Interpreting ITrees (2): Example

```
t : itree (stateE + failE + ioE) unit
```



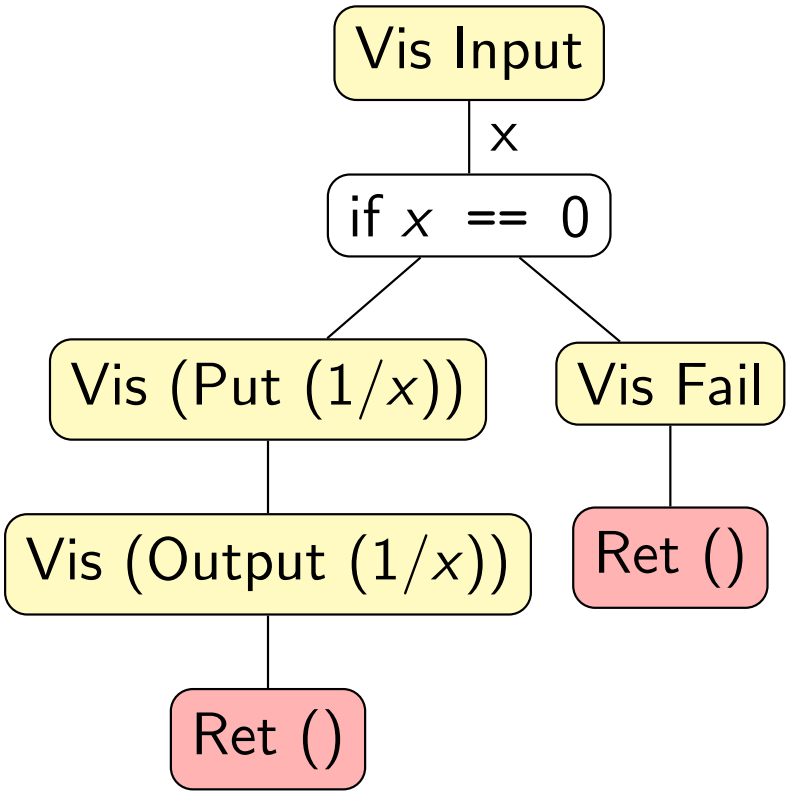

```
Ind failE : Type → Type :=
| Fail : failE void.

Ind stateE (S : Type)
    : Type → Type :=
| Get : stateE S S
| Put : S → stateE S unit.

Ind ioE : Type → Type :=
| Input : IO nat
| Output : nat → IO unit.
```

Interpreting ITrees (3): Example

Interpreting failure

$$\texttt{t}$$

$$\texttt{itree}\ (\texttt{stateE} + \texttt{failE} + \texttt{ioE})\ \texttt{unit}$$



$$\texttt{t'} = \texttt{interp\_failE}\ \texttt{t}$$

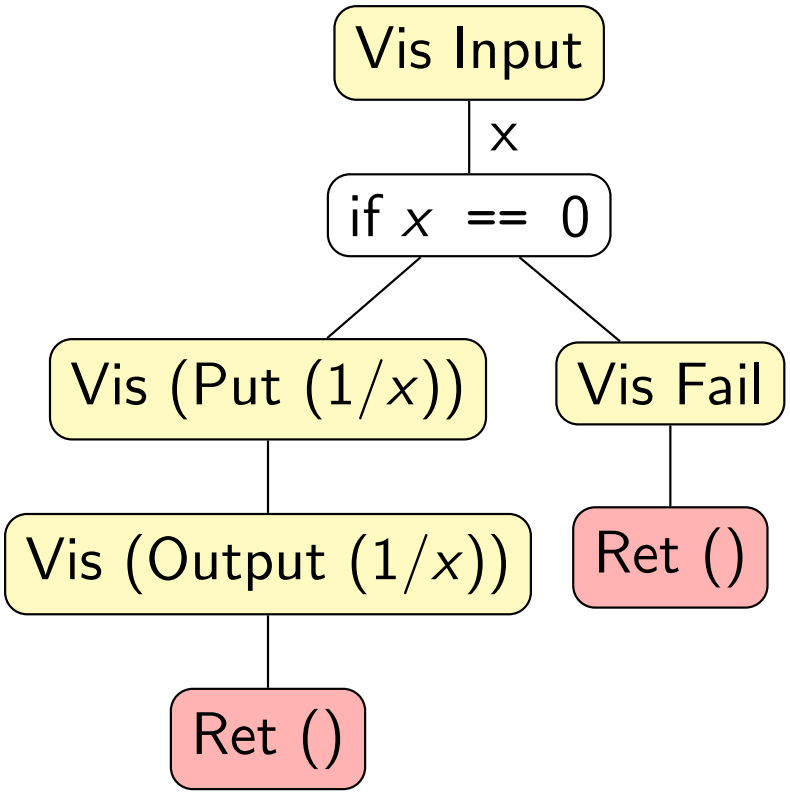
$$\texttt{itree}\ (\texttt{stateE} + \texttt{ioE})\ (\texttt{option}\ \texttt{unit})$$

Interpreting ITrees (3): Example

Interpreting failure

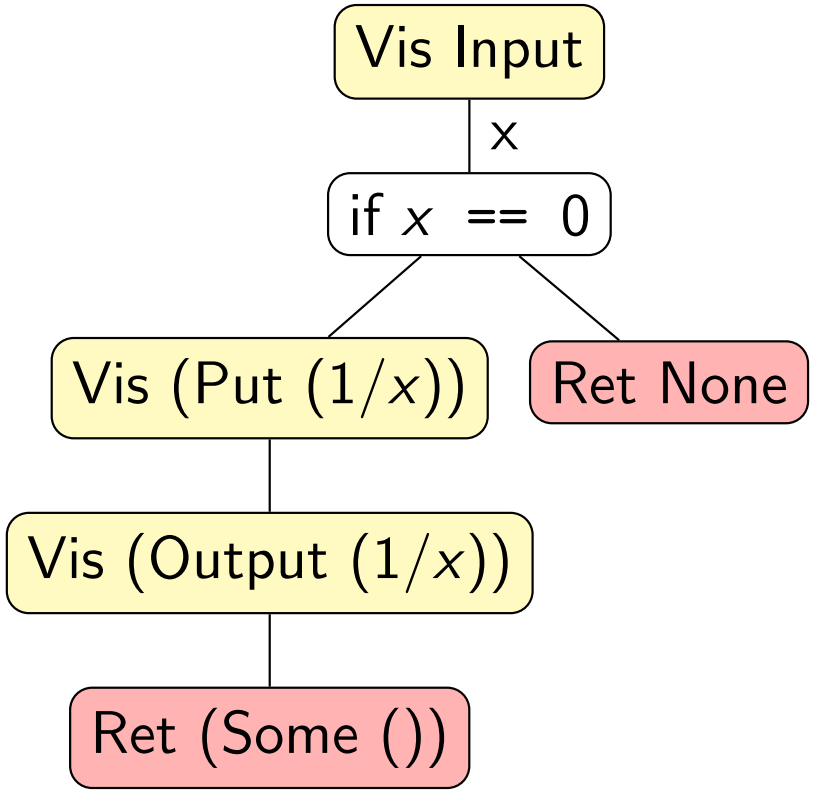
`t`

`itree (stateE + failE + ioE) unit`



`t' = interp_failE t`

`itree (stateE + ioE) (option unit)`



Interpreting ITrees (4): Example

```
CoFixpoint interp_failE : itree (failE + E) A → itree E (option A) := ..
```

Interpreting ITrees (4): Example

```
CoFixpoint interp_failE (t : itree (failE + E) A) : itree E (option A) :=
    match t with
    | Ret r       ⇒ Ret (Some r)
    | Tau t       ⇒ Tau (interp_failE t)
    | Vis Fail k ⇒ Ret None
    | Vis e    k ⇒ Vis e (fun x ⇒ interp_failE (k x))
  end.
```

Overview

What we can do with ITrees

- Representing programs using ITrees
  - *Effectful programs*
  - *Diverging programs*
  - Recursive programs
- *Program Equivalence*
  - by equational reasoning on ITrees
- *Interpreting ITrees*
  - by writing interpreters for effects
- ITree Building Blocks
  - Combinators to make denotation easier
  - Equations on combinators for equational reasoning

Overview

What we can do with ITrees

- Representing programs using ITrees
  - *Effectful programs*
  - *Diverging programs*
  - Recursive programs
- *Program Equivalence*
  - by equational reasoning on ITrees
- *Interpreting ITrees*
  - by writing interpreters for effects
- **ITree Building Blocks**
  - Combinators to make denotation easier
  - Equations on combinators for equational reasoning

Denoting a simple language: λ_Z (1)

We define a pure lambda calculus: λ_Z

- Syntax of λ_Z

$$
\begin{aligned}
v \in \mathsf{Val} &:= z \mid \lambda x.e \ (z \in \mathbb{Z}) \\
e \in \mathsf{Expr} &:= v \mid x \mid e_1 \hat{+} e_2 \mid e_1(e_2) \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\
&\quad \mid \text{while } e_1 \text{ do } e_2
\end{aligned}
$$

Denoting a simple language: λ_Z (1)

We define a pure lambda calculus: λ_Z

- Syntax of λ_Z

$$v \in \mathsf{Val} := z \mid \lambda x.e \ (z \in \mathbb{Z})$$
$$e \in \mathsf{Expr} := v \mid x \mid e_1 \hat{+} e_2 \mid e_1(e_2) \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$$
$$\mid \mathsf{while}\ e_1\ \mathsf{do}\ e_2$$

- Denotation

$$[\![v]\!] := \mathsf{Ret}\ v$$

Denoting a simple language: λ_Z (1)

We define a pure lambda calculus: λ_Z

- Syntax of λ_Z

$$
\begin{aligned}
v \in \mathsf{Val} &:= z \mid \lambda x.e \ (z \in \mathbb{Z}) \\
e \in \mathsf{Expr} &:= v \mid x \mid e_1 \hat{+} e_2 \mid e_1(e_2) \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\
&\quad \mid \mathsf{while}\ e_1\ \mathsf{do}\ e_2
\end{aligned}
$$

- Denotation

$$
\begin{aligned}
[\![v]\!] &:= \mathsf{Ret}\ v \\
[\![e_1\ +\ e_2]\!] &:= v_1 \leftarrow [\![e_1]\!];\ v_2 \leftarrow [\![e_2]\!];\ z_2 \leftarrow \mathsf{to\_int}\ v_2;\ z_1 \leftarrow \mathsf{to\_int}\ v_1; \\
&\quad\ \mathsf{Ret}\ (z_1 + z_2) \\
[\![e_1(e_2)]\!] &:= v_1 \leftarrow [\![e_1]\!];\ v_2 \leftarrow [\![e_2]\!];\ (x, e) \leftarrow \mathsf{to\_lam}\ v_1;\ [\![e[v_2/x]]\!]
\end{aligned}
$$

Denoting a simple language: λ_Z (1)

We define a pure lambda calculus: λ_Z

- Syntax of λ_Z

$$
\begin{aligned}
v \in \mathsf{Val} &:= z \mid \lambda x.e \ (z \in \mathbb{Z}) \\
e \in \mathsf{Expr} &:= v \mid x \mid e_1 \hat{+} e_2 \mid e_1(e_2) \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\
&\quad \mid \mathsf{while}\ e_1\ \mathsf{do}\ e_2
\end{aligned}
$$

- Denotation

$$
\begin{aligned}
[\![v]\!] &:= \mathsf{Ret}\ v \\
[\![e_1\ +\ e_2]\!] &:= v_1 \leftarrow [\![e_1]\!];\ v_2 \leftarrow [\![e_2]\!];\ z_2 \leftarrow \mathsf{to\_int}\ v_2;\ z_1 \leftarrow \mathsf{to\_int}\ v_1; \\
&\quad \mathsf{Ret}\ (z_1 + z_2) \\
[\![e_1(e_2)]\!] &:= v_1 \leftarrow [\![e_1]\!];\ v_2 \leftarrow [\![e_2]\!];\ (x, e) \leftarrow \mathsf{to\_lam}\ v_1;\ [\![e[v_2/x]]\!] \\
[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!] &:= v_1 \leftarrow [\![e_1]\!];\ z_1 \leftarrow \mathsf{to\_int}\ v_1;\ \mathsf{if}\ z_1 \neq 0\ \mathsf{then}\ [\![e_2]\!]\ \mathsf{else}\ [\![e_3]\!]
\end{aligned}
$$

Denoting a simple language: λ_Z (1)

We define a pure lambda calculus: λ_Z

- Syntax of λ_Z

$$
\begin{aligned}
v \in \mathsf{Val} &:= z \mid \lambda x.e \ (z \in \mathbb{Z}) \\
e \in \mathsf{Expr} &:= v \mid x \mid e_1 \hat{+} e_2 \mid e_1(e_2) \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\
&\quad \mid \mathsf{while}\ e_1\ \mathsf{do}\ e_2
\end{aligned}
$$

- Denotation

$$
\begin{aligned}
[\![v]\!] &:= \mathsf{Ret}\ v \\
[\![e_1\ +\ e_2]\!] &:= v_1 \leftarrow [\![e_1]\!];\ v_2 \leftarrow [\![e_2]\!];\ z_2 \leftarrow \mathsf{to\_int}\ v_2;\ z_1 \leftarrow \mathsf{to\_int}\ v_1; \\
&\quad \mathsf{Ret}\ (z_1 + z_2) \\
[\![e_1(e_2)]\!] &:= v_1 \leftarrow [\![e_1]\!];\ v_2 \leftarrow [\![e_2]\!];\ (x, e) \leftarrow \mathsf{to\_lam}\ v_1;\ [\![e[v_2/x]]\!] \\
[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!] &:= v_1 \leftarrow [\![e_1]\!];\ z_1 \leftarrow \mathsf{to\_int}\ v_1;\ \mathsf{if}\ z_1 \neq 0\ \mathsf{then}\ [\![e_2]\!]\ \mathsf{else}\ [\![e_3]\!] \\
\mathsf{to\_int}\ v &:= \mathsf{match}\ v\ \mathsf{with}\ z \Rightarrow \mathsf{Ret}\ z \mid \_ \Rightarrow \mathsf{fail}\ \mathsf{end} \\
\mathsf{to\_lam}\ v &:= \mathsf{match}\ v\ \mathsf{with}\ \lambda x.\, e \Rightarrow \mathsf{Ret}\ (x, e) \mid \_ \Rightarrow \mathsf{fail}\ \mathsf{end}
\end{aligned}
$$

Denoting a simple language: λ_Z (1)

We define a pure lambda calculus: λ_Z

- Syntax of λ_Z

$$v \in \mathsf{Val} := z \mid \lambda x.e \ (z \in \mathbb{Z})$$
$$e \in \mathsf{Expr} := v \mid x \mid e_1 \hat{+} e_2 \mid e_1(e_2) \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \mathsf{while}\ e_1\ \mathsf{do}\ e_2$$

- Denotation

$$[\![v]\!] := \mathsf{Ret}\ v$$
$$[\![e_1\ +\ e_2]\!] := v_1 \leftarrow [\![e_1]\!];\ v_2 \leftarrow [\![e_2]\!];\ z_2 \leftarrow \mathsf{to\_int}\ v_2;\ z_1 \leftarrow \mathsf{to\_int}\ v_1;\ \mathsf{Ret}\ (z_1 + z_2)$$
$$[\![e_1(e_2)]\!] := v_1 \leftarrow [\![e_1]\!];\ v_2 \leftarrow [\![e_2]\!];\ (x, e) \leftarrow \mathsf{to\_lam}\ v_1;\ [\![e[v_2/x]]\!]$$
$$[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!] := v_1 \leftarrow [\![e_1]\!];\ z_1 \leftarrow \mathsf{to\_int}\ v_1;\ \mathsf{if}\ z_1 \neq 0\ \mathsf{then}\ [\![e_2]\!]\ \mathsf{else}\ [\![e_3]\!]$$
$$\mathsf{to\_int}\ v := \mathsf{match}\ v\ \mathsf{with}\ z \Rightarrow \mathsf{Ret}\ z \mid \_ \Rightarrow \mathsf{fail}\ \mathsf{end}$$
$$\mathsf{to\_lam}\ v := \mathsf{match}\ v\ \mathsf{with}\ \lambda x.\, e \Rightarrow \mathsf{Ret}\ (x, e) \mid \_ \Rightarrow \mathsf{fail}\ \mathsf{end}$$

Note that we already use an effect (`fail := trigger Fail`) in the denotation

Denoting a simple language: λ_Z (2)

Denoting λ_Z

- Syntax of λ_Z

$$v \in \mathsf{Val} := z \mid \lambda x.e \ (z \in \mathbb{Z})$$
$$e \in \mathsf{Expr} := v \mid x \mid e_1 \hat{+} e_2 \mid e_1(e_2) \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3$$
$$\mid \text{while } e_1 \text{ do } e_2$$

Denoting a simple language: λ_Z (2)

Denoting λ_Z

- Syntax of λ_Z

$$v \in \mathsf{Val} := z \mid \lambda x.e \ (z \in \mathbb{Z})$$
$$e \in \mathsf{Expr} := v \mid x \mid e_1 \hat{+} e_2 \mid e_1(e_2) \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3$$
$$\mid \text{while } e_1 \text{ do } e_2$$

- What about while?

Denoting a simple language: λ_Z (2)

Denoting λ_Z

- Syntax of λ_Z

$$
\begin{aligned}
v \in \mathsf{Val} &:= z \mid \lambda x.e \ (z \in \mathbb{Z}) \\
e \in \mathsf{Expr} &:= v \mid x \mid e_1 \hat{+} e_2 \mid e_1(e_2) \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\
&\quad \mid \text{while } e_1 \text{ do } e_2
\end{aligned}
$$

- What about while?

$$
\begin{aligned}
[\![\text{while } e_1 \text{ do } e_2]\!] := v_1 \leftarrow [\![e_1]\!];\ & z_1 \leftarrow \text{to\_int } v_1; \\
& \text{if } z_1 \neq 0 \\
& \text{then } \_ \leftarrow [\![e_2]\!];\ \text{Tau } ([\![\textbf{while } \boldsymbol{e_1} \textbf{ do } \boldsymbol{e_2}]\!]) \\
& \text{else Ret } ()
\end{aligned}
$$

Denoting a simple language: λ_Z (2)

Denoting λ_Z

- Syntax of λ_Z

$$v \in \mathsf{Val} := z \mid \lambda x.e \ (z \in \mathbb{Z})$$
$$e \in \mathsf{Expr} := v \mid x \mid e_1 \hat{+} e_2 \mid e_1(e_2) \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \mathsf{while}\ e_1\ \mathsf{do}\ e_2$$

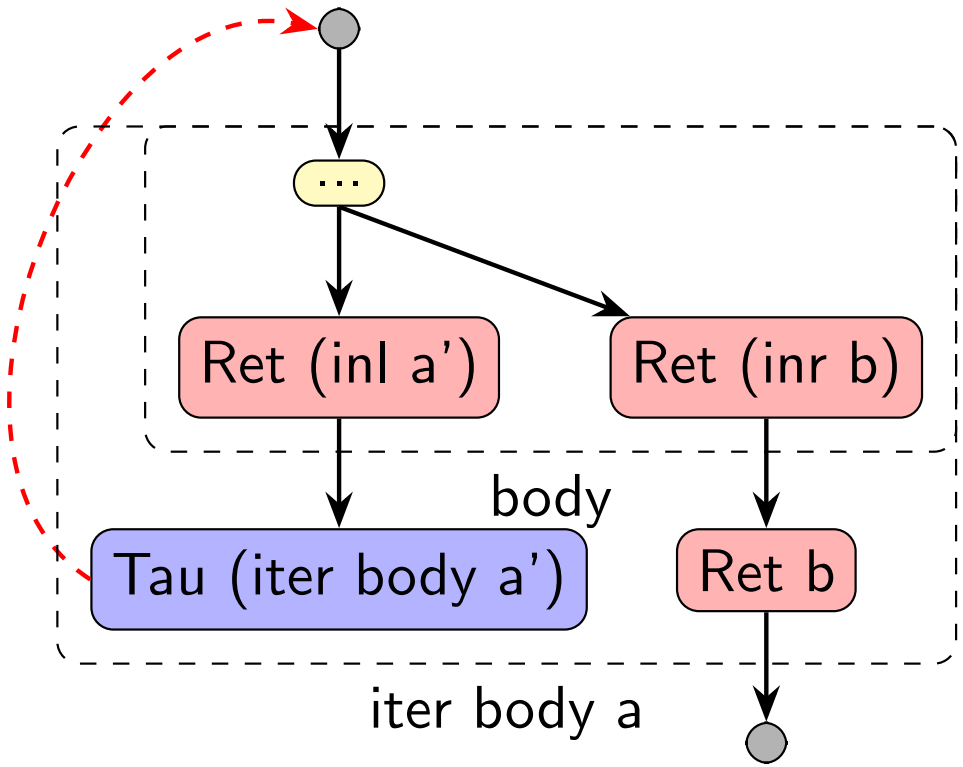
- What about while?

$$\begin{aligned}[\![\mathsf{while}\ e_1\ \mathsf{do}\ e_2]\!] := {} & v_1 \leftarrow [\![e_1]\!];\ z_1 \leftarrow \mathsf{to\_int}\ v_1; \\ & \mathsf{if}\ z_1 \neq 0 \\ & \mathsf{then}\ \_ \leftarrow [\![e_2]\!];\ \mathsf{Tau}\ ([\![\mathbf{while}\ e_1\ \mathbf{do}\ e_2]\!]) \\ & \mathsf{else}\ \mathsf{Ret}\ ()\end{aligned}$$

Note that this denotation is corecursive

Building blocks (1): iter

$$\text{iter} : \underbrace{(A \to \text{itree } E\ (A + B))}_{\textit{body}} \to (A \to \text{itree } E\ B)$$



Denoting a simple language: λ_Z (3)

Back to while

$$
\begin{aligned}
[\![\text{while } e_1 \text{ do } e_2]\!] := \text{iter } (\lambda\_. \\
& v_1 \leftarrow [\![e_1]\!];\ z_1 \leftarrow \text{to\_int } v_1; \\
& \text{if } z_1 \neq 0 \\
& \text{then } \_ \leftarrow [\![e_2]\!];\ \text{Ret } (\text{inl } ()) \\
& \text{else Ret } (\text{inr } ()) \\
&)
\end{aligned}
$$

Denoting a simple language: λ_Z (3)

Back to while

$$
\begin{aligned}
[\![\text{while } e_1 \text{ do } e_2]\!] := \text{iter } (\lambda\_. \\
& v_1 \leftarrow [\![e_1]\!];\ z_1 \leftarrow \text{to\_int } v_1; \\
& \text{if } z_1 \neq 0 \\
& \text{then } \_ \leftarrow [\![e_2]\!];\ \text{Ret } (\text{inl } ()) \\
& \text{else Ret } (\text{inr } ()) \\
&)
\end{aligned}
$$

Now the corecursion is hidden away

Building blocks (2): iter

Use cases of `iter`

Building blocks (2): iter

Use cases of `iter`

- `iter` makes denoting loops easy

Building blocks (2): iter

Use cases of `iter`

- `iter` makes denoting loops easy
- `iter` makes reasoning about loops easier

Building blocks (2): iter

Use cases of `iter`

- `iter` makes denoting loops easy
- `iter` makes reasoning about loops easier
- Examples of proven equations for `iter`

```
(* helpers *)
⋙ : (A → itree E B) → (B → itree E C) → (A → itree E C)
case_ : (A → itee E C) → (B → itree E C) → ((A + B) → itree E C)
bimap : (A → itree E C) → (B → itree E D) → ((A + B) → itree E (C + D))
...

(* Loop unfolding *)
iter f      ≈ f ⋙ case_ (iter f) id_
(* iter f then g = iter f with g in last iteration*)
iter f ⋙ g ≈ iter (f ⋙ bimap id_ g)
...
```

Building blocks (3): mrec

The mrec combinator

- mrec is defined using iter
- iteration on effects/itrees instead of values
- Allows for easy encoding of recursion in effects
- Supports equational reasoning

Overview

What we can do with ITrees

- Representing programs using ITrees
  - **Effectful programs**
  - **Diverging programs**
  - Recursive programs
- **Program Equivalence**
  - by equational reasoning on ITrees
- **Interpreting ITrees**
  - by writing interpreters for effects
- **ITree Building Blocks**
  - Combinators to make denotation easier
  - Equations on combinators for equational reasoning

Program Logics à la Carte

Vistrup, Sammler & Jung

2025

Program logics (1)

- ITree interpretation and small step semantics are 'execution' models
  - Given some input, what is the result

Program logics (1)

- ITree interpretation and small step semantics are 'execution' models
  - Given some input, what is the result
- Reasoning about programs
  - Example:
    Given: if $x < y$ then $r = x$ else $r = y$
    Property: for all x and y, r is always the minimum of x and y

Program logics (1)

- ITree interpretation and small step semantics are 'execution' models
  - Given some input, what is the result
- Reasoning about programs
  - Example:
    Given: if $x < y$ then $r = x$ else $r = y$
    Property: for all x and y, r is always the minimum of x and y
- Program logics
  - Example: Hoare logic ($\{P\}\ e\ \{Q\}$)
  - Program verification: We can verify properties for all inputs

Program logics (3)

Hoare logic vs Seperation logic

- The paper uses the Iris seperation logic framework
  - Hoare logic
    - Hoare triple: $\{P\}\ e\ \{Q\}$
    - precondition, program, postcondition

Program logics (3)

Hoare logic vs Seperation logic

- The paper uses the Iris seperation logic framework
  - Hoare logic
    - Hoare triple: $\{P\}\ e\ \{Q\}$
    - precondition, program, postcondition
  - Seperation logic
    - Extension of Hoare logic
    - Seperating conjunction: $*$ (treat as conjunction)
    - Seperating implication/magic wand: $-\!\!*$ (treat as implication)
    - Weakest precondition: wp $e\ \{Q\}$
    - Relation to Hoare triple: $\{P\}\ e\ \{Q\} := P -\!\!*\ \text{wp}\ e\ \{Q\}$

Program logics (3)

To define a program logic you need:

Program logics (3)

To define a program logic you need:

- Language semantics
  - Typically a small step semantic model
  - Example rule:

$$\frac{[\![b]\!](s) = \text{false} \qquad \langle p_2, s\rangle \to s'}{\langle \text{if } b \text{ then } p_1 \text{ else } p_2, s\rangle \to s'} \text{ [If-False]}$$

Program logics (3)

To define a program logic you need:

- Language semantics
  - Typically a small step semantic model
  - Example rule:

$$\frac{[\![b]\!](s) = \text{false} \qquad \langle p_2, s\rangle \to s'}{\langle \text{if } b \text{ then } p_1 \text{ else } p_2, s\rangle \to s'} \text{ [If-False]}$$

- Program logic rules
  - Example rule:

$$\frac{\{P \wedge b\}\ C_1\ \{Q\} \qquad \{P \wedge \neg b\}\ C_2\ \{Q\}}{\{P\}\ \text{if } b \text{ then } C_1 \text{ else } C_2\ \{Q\}} \text{ [If]}$$

Program logics (3)

To define a program logic you need:

- Language semantics
  - Typically a small step semantic model
  - Example rule:

$$\frac{[\![b]\!](s) = \text{false} \qquad \langle p_2, s\rangle \to s'}{\langle \text{if } b \text{ then } p_1 \text{ else } p_2, s\rangle \to s'} \text{ [If-False]}$$

- Program logic rules
  - Example rule:

$$\frac{\{P \wedge b\}\ C_1\ \{Q\} \qquad \{P \wedge \neg b\}\ C_2\ \{Q\}}{\{P\}\ \text{if } b \text{ then } C_1 \text{ else } C_2\ \{Q\}} \text{ [If]}$$

- Soundness proof
  - Can't proof anything false w.r.t the language semantics
  - Example theorem:

Theorem (Soundness of Hoare Logic)

If $\{P\}\ p\ \{Q\}$ is derivable, then for every state s such that $P(s)$ and every s' such that $\langle p, s\rangle \to s'$, we have $Q(s')$.

Program logics (4)

This approach is not ideal:

- For every new language
  - New language semantics
  - New program logic rules
  - New soundness proof
- Adding language feature
  - Same story
- Non-modularity is the problem
  - Idea: use ITrees as underlying language semantics

Program logics à la carte

What the paper contributes

Program logics à la carte

What the paper contributes

- A new way to define program logics for itrees

Program logics à la carte

What the paper contributes

- A new way to define program logics for itrees
- A Rocq implementation of program logic fragments for
  - safe and unsafe program termination (failure)
  - state
  - non-determinism (demonic & angelic)
  - concurrency

Program logics à la carte

What the paper contributes

- A new way to define program logics for itrees
- A Rocq implementation of program logic fragments for
  - safe and unsafe program termination (failure)
  - state
  - non-determinism (demonic & angelic)
  - concurrency
- Soundness proofs of these fragments
  - w.r.t itree interpretation

Program logics à la carte

What the paper contributes

- A new way to define program logics for itrees
- A Rocq implementation of program logic fragments for
  - safe and unsafe program termination (failure)
  - state
  - non-determinism (demonic & angelic)
  - concurrency
- Soundness proofs of these fragments
  - w.r.t itree interpretation
- A port of two existing program logics
  - HeapLang (imperative programs with effects)
  - Islaris (machine code programs)

Program logics à la carte

For this presentation

- A new way to define program logics for itrees
- crash/failure program logic fragment
- Building a program logic for λ_Z

A program logic for: λ_Z

We define a pure lambda calculus: λ_Z

- Syntax of λ_Z

$$v \in \mathsf{Val} := z \mid \lambda x.e \ (z \in \mathbb{Z})$$
$$e \in \mathsf{Expr} := v \mid x \mid e_1 \hat{+} e_2 \mid e_1(e_2) \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3$$

- Denotation

$$\begin{aligned}
[\![v]\!] &:= \mathsf{Ret}\ v \\
[\![e_1 + e_2]\!] &:= v_1 \leftarrow [\![e_1]\!];\ v_2 \leftarrow [\![e_2]\!];\ z_2 \leftarrow \mathsf{to_int}\ v_2;\ z_1 \leftarrow \mathsf{to_int}\ v_1; \\
&\quad \mathsf{Ret}\ (z_1 + z_2) \\
[\![e_1(e_2)]\!] &:= v_1 \leftarrow [\![e_1]\!];\ v_2 \leftarrow [\![e_2]\!];\ (x, e) \leftarrow \mathsf{to_lam}\ v_1;\ [\![e[v_2/x]]\!] \\
[\![\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\!] &:= v_1 \leftarrow [\![e_1]\!];\ z_1 \leftarrow \mathsf{to_int}\ v_1;\ \text{if } z_1 \neq 0 \text{ then } [\![e_2]\!] \text{ else } [\![e_3]\!] \\
\mathsf{to_int}\ v &:= \mathsf{match}\ v\ \mathsf{with}\ z \Rightarrow \mathsf{Ret}\ z \mid _ \Rightarrow \mathsf{fail}\ \mathsf{end} \\
\mathsf{to_lam}\ v &:= \mathsf{match}\ v\ \mathsf{with}\ \lambda x.\, e \Rightarrow \mathsf{Ret}\ (x, e) \mid _ \Rightarrow \mathsf{fail}; \mathsf{end}
\end{aligned}$$

Note that we already use an effect ($\texttt{fail} := \texttt{trigger Fail}$) in the denotation

Basic Rules

Program logic rules: wp e $\{\Phi\}$

Basic Rules

Program logic rules: $\mathsf{wp}\ e\ \{\Phi\}$

$$\frac{\forall r.\ \Phi(r) \vdash \Psi(r) \qquad \mathsf{wp}\ e\ \{\Phi\}}{\mathsf{wp}\ e\ \{\Psi\}}\ \text{[WpConsequence]}$$

Basic Rules

Program logic rules: $\mathsf{wp}\ e\ \{\Phi\}$

$$\frac{\forall r.\ \Phi(r) \vdash \Psi(r) \qquad \mathsf{wp}\ e\ \{\Phi\}}{\mathsf{wp}\ e\ \{\Psi\}}\ \text{[WpConsequence]} \qquad \frac{\Phi(v)}{\mathsf{wp}\ v\ \{\Phi\}}\ \text{[WpVal]}$$

Basic Rules

Program logic rules: $\mathsf{wp}\ e\ \{\Phi\}$

$$\frac{\forall r.\ \Phi(r) \vdash \Psi(r) \qquad \mathsf{wp}\ e\ \{\Phi\}}{\mathsf{wp}\ e\ \{\Psi\}}\ [\text{WpConsequence}] \qquad \frac{\Phi(v)}{\mathsf{wp}\ v\ \{\Phi\}}\ [\text{WpVal}]$$

$$\frac{\mathsf{wp}\ e_1\ \{v_1.\ \mathsf{wp}\ v_1 \mathbin{\hat{+}} e_2\ \{\Phi\}\}}{\mathsf{wp}\ e_1 \mathbin{\hat{+}} e_2\ \{\Phi\}}\ [\text{WpBindPlusL}]$$

Basic Rules

Program logic rules: $\mathsf{wp}\ e\ \{\Phi\}$

$$\frac{\forall r.\ \Phi(r) \vdash \Psi(r) \qquad \mathsf{wp}\ e\ \{\Phi\}}{\mathsf{wp}\ e\ \{\Psi\}}\ [\mathsf{WpConsequence}] \qquad \frac{\Phi(v)}{\mathsf{wp}\ v\ \{\Phi\}}\ [\mathsf{WpVal}]$$

$$\frac{\mathsf{wp}\ e_1\ \{v_1.\ \mathsf{wp}\ v_1 \mathbin{\hat{+}} e_2\ \{\Phi\}\}}{\mathsf{wp}\ e_1 \mathbin{\hat{+}} e_2\ \{\Phi\}}\ [\mathsf{WpBindPlusL}] \qquad \frac{\Phi(z_1 + z_2)}{\mathsf{wp}\ z_1 \mathbin{\hat{+}} z_2\ \{\Phi\}}\ [\mathsf{WpPlus}]$$

Basic Rules

Program logic rules: $\mathsf{wp}\ e\ \{\Phi\}$

$$\frac{\forall r.\ \Phi(r) \vdash \Psi(r) \qquad \mathsf{wp}\ e\ \{\Phi\}}{\mathsf{wp}\ e\ \{\Psi\}}\ [\mathsf{WpConsequence}] \qquad \frac{\Phi(v)}{\mathsf{wp}\ v\ \{\Phi\}}\ [\mathsf{WpVal}]$$

$$\frac{\mathsf{wp}\ e_1\ \{v_1.\ \mathsf{wp}\ v_1 \mathbin{\hat{+}} e_2\ \{\Phi\}\}}{\mathsf{wp}\ e_1 \mathbin{\hat{+}} e_2\ \{\Phi\}}\ [\mathsf{WpBindPlusL}] \qquad \frac{\Phi(z_1 + z_2)}{\mathsf{wp}\ z_1 \mathbin{\hat{+}} z_2\ \{\Phi\}}\ [\mathsf{WpPlus}]$$

$$\frac{z \neq 0 \qquad \mathsf{wp}\ e_1\ \{\Phi\}}{\mathsf{wp}\ \mathsf{if}\ z\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \{\Phi\}}\ [\mathsf{WpIfTrue}] \qquad \cdots$$

Reusable Rules (1)

- Defining rules like this is standard

Reusable Rules (1)

- Defining rules like this is standard
- We can do better:
  - Idea: Define rules for arbitrary ITrees

Reusable Rules (1)

- Defining rules like this is standard
- We can do better:
  - Idea: Define rules for arbitrary ITrees
  - Weakest precondition for ITrees: $\text{wpi}_H\ t\ \{\Phi\}$
    - t : itree E R
    - H is a **logical** effect handler for every event in E

Reusable Rules (1)

- Defining rules like this is standard
- We can do better:
  - Idea: Define rules for arbitrary ITrees
  - Weakest precondition for ITrees: $\mathsf{wpi}_H\ t\ \{\Phi\}$
    - t : itree E R
    - H is a **logical** effect handler for every event in E
- For λ_Z we get
  - $\mathsf{wp}\ e\ \{\Phi\} := \mathsf{wpi}_{\mathbf{LangH}_Z}\ [\![e]\!]\ \{\Phi\}$

Reusable Rules (1)

- Defining rules like this is standard
- We can do better:
  - Idea: Define rules for arbitrary ITrees
  - Weakest precondition for ITrees: $\mathsf{wpi}_H\ t\ \{\Phi\}$
    - t : itree E R
    - H is a **logical** effect handler for every event in E
- For λ_Z we get
  - $\mathsf{wp}\ e\ \{\Phi\} := \mathsf{wpi}_{\mathbf{LangH}_Z}\ [\![e]\!]\ \{\Phi\}$
  - Our language effects $\mathbf{LangE}_Z :=$ FailE

Reusable Rules (1)

- Defining rules like this is standard
- We can do better:
  - Idea: Define rules for arbitrary ITrees
  - Weakest precondition for ITrees: $\mathsf{wpi}_H\ t\ \{\Phi\}$
    - t : itree E R
    - H is a **logical** effect handler for every event in E
- For λ_Z we get
  - $\mathsf{wp}\ e\ \{\Phi\} := \mathsf{wpi}_{\mathbf{LangH}_Z}\ [\![e]\!]\ \{\Phi\}$
  - Our language effects $\mathbf{LangE}_Z :=$ FailE
  - Our logical effect handler $\mathbf{LangH}_Z :=$ FailH (to be defined)

Reusable Rules (2)

Program logic rules: $\mathsf{wpi}_H\ t\ \{\Phi\}$

Reusable Rules (2)

Program logic rules: $\mathsf{wpi}_H\ t\ \{\Phi\}$

$$\frac{\forall r.\ \Phi(r) \vdash \Psi(r) \qquad \mathsf{wpi}_H\ t\ \{\Phi\}}{\mathsf{wpi}_H\ t\ \{\Psi\}}\ \text{[WpiConsequence]}$$

Reusable Rules (2)

Program logic rules: $\mathsf{wpi}_H\ t\ \{\Phi\}$

$$\frac{\forall r.\ \Phi(r) \vdash \Psi(r) \qquad \mathsf{wpi}_H\ t\ \{\Phi\}}{\mathsf{wpi}_H\ t\ \{\Psi\}}\ [\text{WpiConsequence}]$$

$$\frac{\Phi(r)}{\mathsf{wpi}_H\ \mathsf{Ret}(r)\ \{\Phi\}}\ [\text{WpiRet}]$$

Reusable Rules (2)

Program logic rules: $\mathsf{wpi}_H\ t\ \{\Phi\}$

$$\frac{\forall r.\ \Phi(r) \vdash \Psi(r) \qquad \mathsf{wpi}_H\ t\ \{\Phi\}}{\mathsf{wpi}_H\ t\ \{\Psi\}}\ [\text{WpiConsequence}]$$

$$\frac{\Phi(r)}{\mathsf{wpi}_H\ \mathsf{Ret}(r)\ \{\Phi\}}\ [\text{WpiRet}]$$

$$\frac{t_1 \approx t_2}{\mathsf{wpi}_H\ t_1\ \{\Phi\} \dashv\vdash \mathsf{wpi}_H\ t_2\ \{\Phi\}}\ [\text{WpiEutt}]$$

Reusable Rules (2)

Program logic rules: $\mathsf{wpi}_H\ t\ \{\Phi\}$

$$\frac{\forall r.\ \Phi(r) \vdash \Psi(r) \qquad \mathsf{wpi}_H\ t\ \{\Phi\}}{\mathsf{wpi}_H\ t\ \{\Psi\}}\ [\text{WpiConsequence}]$$

$$\frac{\Phi(r)}{\mathsf{wpi}_H\ \mathsf{Ret}(r)\ \{\Phi\}}\ [\text{WpiRet}]$$

$$\frac{t_1 \approx t_2}{\mathsf{wpi}_H\ t_1\ \{\Phi\} \dashv\vdash \mathsf{wpi}_H\ t_2\ \{\Phi\}}\ [\text{WpiEutt}]$$

$$\frac{\mathsf{wpi}_H\ t\ \{x.\ \mathsf{wpi}_H\ k(x)\ \{\Phi\}\}}{\mathsf{wpi}_H\ x \leftarrow t;\ k(x)\ \{\Phi\}}\ [\text{WpiBind}]$$

Reusable Rules (2)

Program logic rules: $\mathsf{wpi}_H\ t\ \{\Phi\}$

$$\frac{\forall r.\ \Phi(r) \vdash \Psi(r) \qquad \mathsf{wpi}_H\ t\ \{\Phi\}}{\mathsf{wpi}_H\ t\ \{\Psi\}}\ [\text{WpiConsequence}]$$

$$\frac{\Phi(r)}{\mathsf{wpi}_H\ \mathsf{Ret}(r)\ \{\Phi\}}\ [\text{WpiRet}]$$

$$\frac{t_1 \approx t_2}{\mathsf{wpi}_H\ t_1\ \{\Phi\} \dashv\vdash \mathsf{wpi}_H\ t_2\ \{\Phi\}}\ [\text{WpiEutt}]$$

$$\frac{\mathsf{wpi}_H\ t\ \{x.\ \mathsf{wpi}_H\ k(x)\ \{\Phi\}\}}{\mathsf{wpi}_H\ x \leftarrow t;\ k(x)\ \{\Phi\}}\ [\text{WpiBind}]$$

- The previous rules follow from these rules.
- These rules are generic and can be used to define other (pure) language specific rules

Weakest precondition for ITrees (1)

The wpi_H definition:

$$\mathrm{wpi}_H\ t\ \{\Phi\}\ :=\ \begin{cases} \Phi(r) & \text{if } t = \mathsf{Ret}\ r \\ \mathrm{wpi}_H\ t'\ \{\Phi\} & \text{if } t = \mathsf{Tau}\ t' \\ \end{cases}$$

Weakest precondition for ITrees (1)

The wpi_H definition:

$$\mathsf{wpi}_H\ t\ \{\Phi\}\ :=\ \begin{cases} \Phi(r) & \text{if } t = \mathsf{Ret}\ r \\ \mathsf{wpi}_H\ t'\ \{\Phi\} & \text{if } t = \mathsf{Tau}\ t' \\ H_A(\epsilon,\ (\lambda a.\ \mathsf{wpi}_H\ (k\ a)\ \{\Phi\})) & \text{if } t = \mathsf{Vis}_A\ \epsilon\ k \end{cases}$$

Weakest precondition for ITrees (2)

$$\text{wpi}\ \underbrace{(\text{Vis}_A\ \epsilon\ k)}_{\text{itree}}\{\Phi\} := \underbrace{H_A(\epsilon, \underbrace{(\lambda a.\ \text{wpi}\ (k\ a)\ \{\Phi\})}_{\text{logical continuation}})}_{\text{logical effect handler}}$$

- Weakest precondition of effects are defered to logical effect handlers: $H_A(\epsilon, \Psi)$

Weakest precondition for ITrees (2)

$$\text{wpi}\ \underbrace{(\text{Vis}_A\ \epsilon\ k)}_{\text{itree}}\{\Phi\} := \underbrace{H_A(\epsilon, \underbrace{(\lambda a.\ \text{wpi}\ (k\ a)\ \{\Phi\})}_{\text{logical continuation}})}_{\text{logical effect handler}}$$

- Weakest precondition of effects are defered to logical effect handlers: $H_A(\epsilon, \Psi)$
- $H_A(\epsilon, \Psi)$ describes the 'verification' condition of executing the effect ϵ

Weakest precondition for ITrees (2)

$$\text{wpi}\ \underbrace{(\text{Vis}_A\ \epsilon\ k)}_{\text{itree}}\{\Phi\} := \underbrace{H_A(\epsilon, \underbrace{(\lambda a.\ \text{wpi}\ (k\ a)\ \{\Phi\})}_{\text{logical continuation}})}_{\text{logical effect handler}}$$

- Weakest precondition of effects are defered to logical effect handlers: $H_A(\epsilon, \Psi)$
- $H_A(\epsilon, \Psi)$ describes the 'verification' condition of executing the effect ϵ
- The cannonical form: $H_A(\epsilon, \Psi) = P * (\forall a : A. Q\ a \mathrel{-\!\!*} \Psi\ a)$
  - P, precondition for the effect ϵ
  - Q, postcondition for the effect ϵ

Effect: Failure (1)

Back to λ_Z

- λ_Z up to now
  - wp $e\ \{\Phi\} := \mathsf{wpi}_{\mathbf{LangH}_Z}\ [\![e]\!]\ \{\Phi\}$
  - $\mathbf{LangE}_Z :=$ FailE
  - $\mathbf{LangH}_Z :=$ FailH (to be defined)

Effect: Failure (2)

FailH

$$\mathsf{wpi}(\mathsf{Vis}_{\emptyset}\ \mathsf{Fail}\ k)\{\Phi\} := \mathsf{FailH}_{\emptyset}(\mathsf{Fail}, (\lambda a.\ \mathsf{wpi}\ (k\ a)\ \{\Phi\}))$$

Effect: Failure (2)

FailH

$$\text{wpi}(\text{Vis}_\emptyset\ \text{Fail}\ k)\{\Phi\} := \text{FailH}_\emptyset(\text{Fail}, (\lambda a.\ \text{wpi}\ (k\ a)\ \{\Phi\}))$$

Cannonical form:

$$\text{FailH}_\emptyset(\text{Fail}, \Psi) = P * (\forall a : A. Q\ a \mathrel{-\!\!*} \Psi\ a)$$

- What are P and Q?

Effect: Failure (2)

FailH

$$\text{wpi}(\text{Vis}_{\emptyset}\ \text{Fail}\ k)\{\Phi\} := \text{FailH}_{\emptyset}(\text{Fail}, (\lambda a.\ \text{wpi}\ (k\ a)\ \{\Phi\}))$$

Cannonical form:

$$\text{FailH}_{\emptyset}(\text{Fail}, \Psi) = P * (\forall a : A. Q\ a \mathrel{-\!\!*} \Psi\ a)$$

- What are P and Q?
- Precondition: we never want our program to fail

Effect: Failure (2)

FailH

$$\text{wpi}(\text{Vis}_\emptyset\ \text{Fail}\ k)\{\Phi\} := \text{FailH}_\emptyset(\text{Fail}, (\lambda a.\ \text{wpi}\ (k\ a)\ \{\Phi\}))$$

Cannonical form:

$$\text{FailH}_\emptyset(\text{Fail}, \Psi) = \bot * (\forall a : A. Q\ a \mathrel{-\!\!*} \Psi\ a)$$

- What are P and Q?
- Precondition: we never want our program to fail

Effect: Failure (2)

FailH

$$\mathsf{wpi}(\mathsf{Vis}_{\emptyset}\ \mathsf{Fail}\ k)\{\Phi\} := \mathsf{FailH}_{\emptyset}(\mathsf{Fail}, (\lambda a.\ \mathsf{wpi}\ (k\ a)\ \{\Phi\}))$$

Cannonical form:

$$\mathsf{FailH}_{\emptyset}(\mathsf{Fail}, \Psi) = \bot$$

- What are P and Q?
- Precondition: we never want our program to fail

Effect: Failure (3)

Back to λ_Z

- λ_Z up to now
  - wp $e\ \{\Phi\} := \mathsf{wpi}_{\mathbf{LangH}_Z}\ [\![e]\!]\ \{\Phi\}$
  - $\mathbf{LangE}_Z :=$ FailE
  - $\mathbf{LangH}_Z :=$ FailH

Effect: Failure (3)

Back to λ_Z

- λ_Z up to now
  - wp $e\ \{\Phi\} := \mathsf{wpi}_{\mathbf{LangH}_Z}\ [\![e]\!]\ \{\Phi\}$
  - $\mathbf{LangE}_Z :=$ FailE
  - $\mathbf{LangH}_Z :=$ FailH
- We can easily add fragments
  - Extend language
  - Give ITree denotation
  - Give wp and wpi rules
  - Give logical effect handlers

Effect: Failure (3)

Back to λ_Z

- λ_Z up to now
  - wp e $\{\Phi\} := \mathsf{wpi}_{\mathbf{LangH}_Z}$ $[\![e]\!]$ $\{\Phi\}$
  - $\mathbf{LangE}_Z :=$ FailE
  - $\mathbf{LangH}_Z :=$ FailH
- We can easily add fragments
  - Extend language
  - Give ITree denotation
  - Give wp and wpi rules
  - Give logical effect handlers
- Many fragments are already implemented

Summary

In this presentation

- Interaction trees
  - Denotation of programs
  - Program Equivalence
  - Interpretation
  - Combinators

Summary

In this presentation

- Interaction trees
  - Denotation of programs
  - Program Equivalence
  - Interpretation
  - Combinators
- Program logics
  - Program logic for ITrees
  - Failure program logic fragment