# Interaction Trees

## A denotational semantics and its equational theorems

Li-yao Xia et al. @ POPL' 20

Presented by Yanning Chen @ PL Lunch

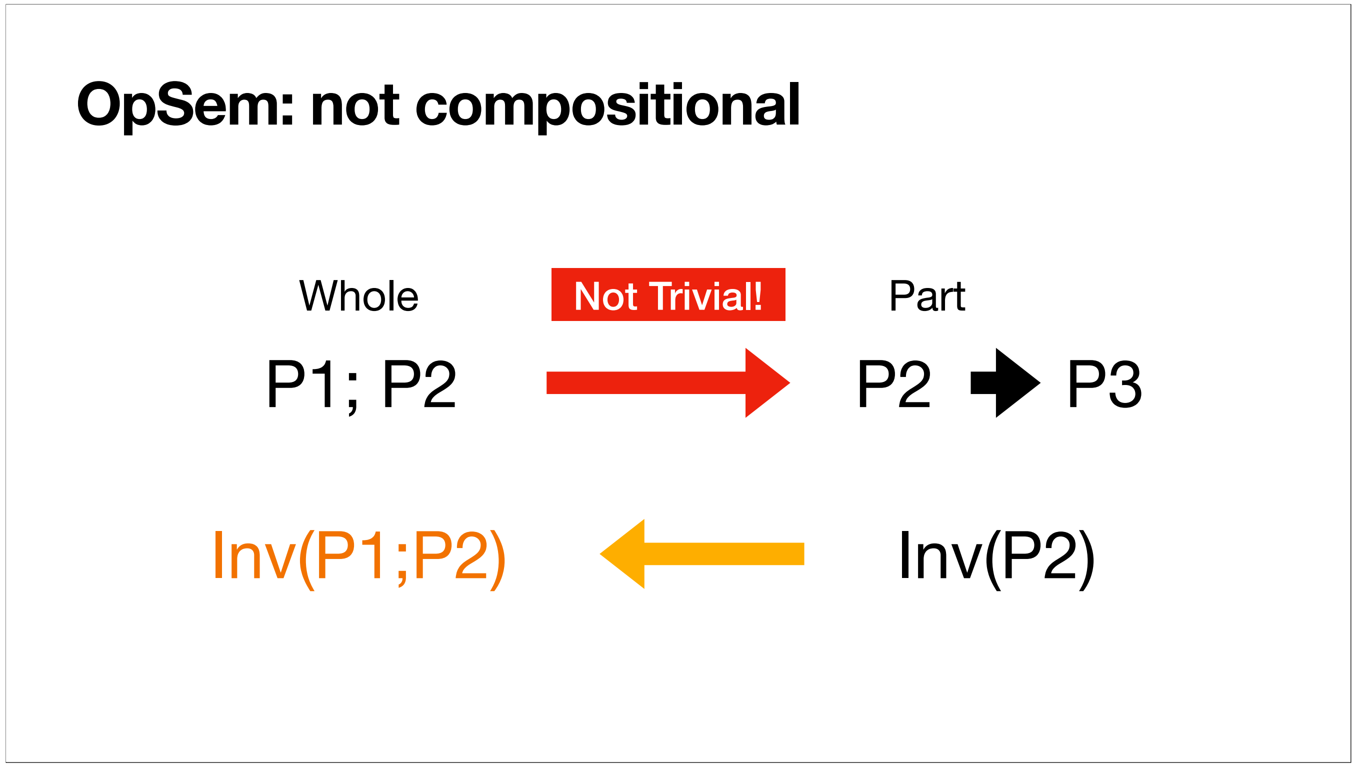
# Formal Semantics

# Operational semantics

**e.g. big step/small step**

- Semantics: execution (transition system + trace)
- S1 —event—> S2
- Intuitive & Expressive
- Inductive reasoning

1. opsem: e.g. ssos/bsos, semantics is its execution (often modelled by TRS), expressive (nearly any feature can be modeled by transition systems & traces), reason by inductive principles supported by most provers,




BUT not compositional (relate the meaning of the whole program to the meaning of its parts)

# OpSem: syntax clutter

$$\begin{array}{rl} E \vdash & \\ F \vdash & S1, K1 \\ LE \vdash & \\ M \vdash & \end{array} \xrightarrow{\text{trace}} \begin{array}{rl} E \vdash & \\ F \vdash & S2, K2 \\ LE \vdash & \\ M \vdash & \end{array}$$

syntax clutter (PC, subst, eval ctx) making proofs hard to write

## Axiom Semantics
### e.g. Hoare logic

- Program: logic formulas that describe it
- Semantics: what can be proven about it
- Higher abstraction
- (Mostly) compositional
- Can be automated (SMT solvers)
- Details are lost

axiom sem: e.g. hoare logic, a program is logic formulas that describe it, and its semantics is what can be proven about it. higher abstraction, more aligned with goals (assertions), often compositional, can be automated (SMT solvers), but many details are lost

# Denotational semantics

- Semantics: what a program denotes trivially
- e.g. Lang := _ + _ | _ - _ | $\mathbb{N} \Rightarrow \mathbb{N}$
- Math: domain theory ; PL: host language
- Reuse host language features -> no more syntax clutters!
- Can be executed/extracted
- Practical languages -?-> Proof assistant languages

Effects, non-terminating Pure, total

meaning of a program is what it denotes trivially. e.g. Lang denotes to nat.
In CS: denote to host language.
shallow representation: abstract away syntax clutters and reuse host language features
can be executed/extracted.
Problem: practical languages with effects and non-termination -?-> pure & terminating proof languages? Introduce to ITrees!

# Interaction Tree

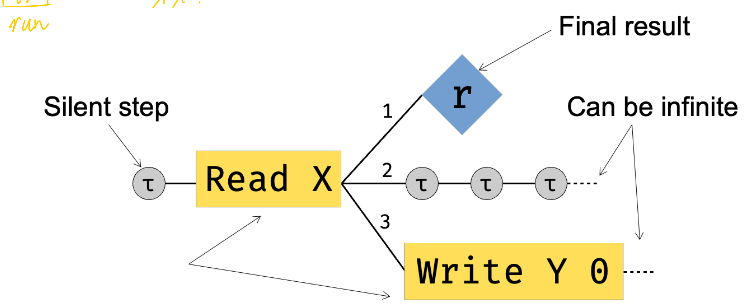
## A shallow representation of (delimited) computations

output ty effect value ty
Value

```
CoInductive itree (E: Type → Type) (R: Type): Type :=
| Ret (r: R) (* computation terminating with value r *)
| Tau (t: itree E R) (* "silent" tau transition with child t *)
| Vis {A: Type} (e : E A) (k : A → itree E R). (* visible event e yielding an answer in A *)
```

Crucial to non-terminating structure

eff output continuation λx. . . .
run




1. show the definition, explain what E and R represents
2. explain what does each variant do
   1. Ret: bare value
   2. Tau: do nothing, silent, crucial to non-terminating structure
   3. Vis: visible effects, kont (coq function, shallow)
3. delimited shallow computation split by Tau and Vis, can represent non-terminating computation & effect

# Interaction Tree

## A shallow representation of (delimited) computations

Events may carry data

And they expect an answer

```
Inductive storeE : Type → Type :=
| Read  (v : variable)           : storeE nat
| Write (v : variable) (n : nat) : storeE unit
```

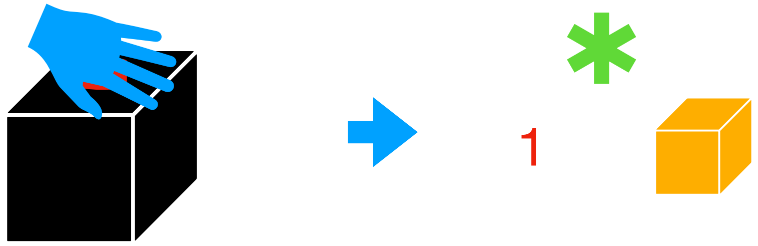
Example:

```
Vis (Read X) (λ(n : nat) =>
Vis (Write Y (n+1)) (λ(_ : unit) =>
Ret 0))
```

A taste of effects, will come back to it later.
Let’s first talk about coinductive types

# What is coinduction?

- Inductive type: What’s inside the box?
- Coinductive type: What can we do about this box?



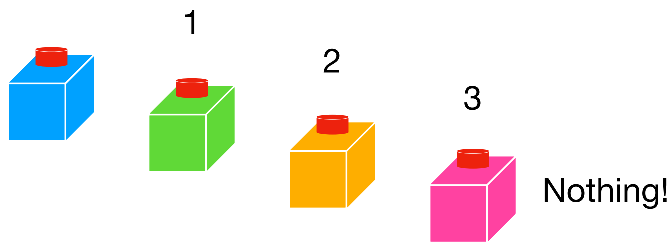

1. induction type: construct by saying what’s inside it, i.e. defined by introduction rule.
2. coinductive type: construct by what can be done about it, i.e. defined by elimination rule.
3. coinductive type is like a black box with a button on it. defined by saying what will pop out after you push the button.

```
Inductive list : Type :=
  | nil : list
  | cons : A -> list -> list.
```

```
Variant CoListF {this : Type} :=
| CoCons (hd: nat) (tl: this)
| CoNil.

CoInductive CoList : Type :=
  Press { emit : @CoListF CoList }.

Notation cocons x y := (Press (CoCons x y)).
Notation conil := (Press CoNil).
```

```
Definition l: list nat :=
  cons 1 (cons 2 (cons 3 nil)).
```

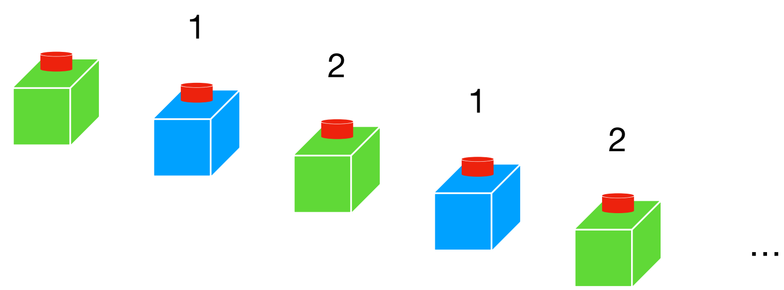
```
Definition cl: CoList :=
  cocons 1 (cocons 2 (cocons 3 conil)).
```



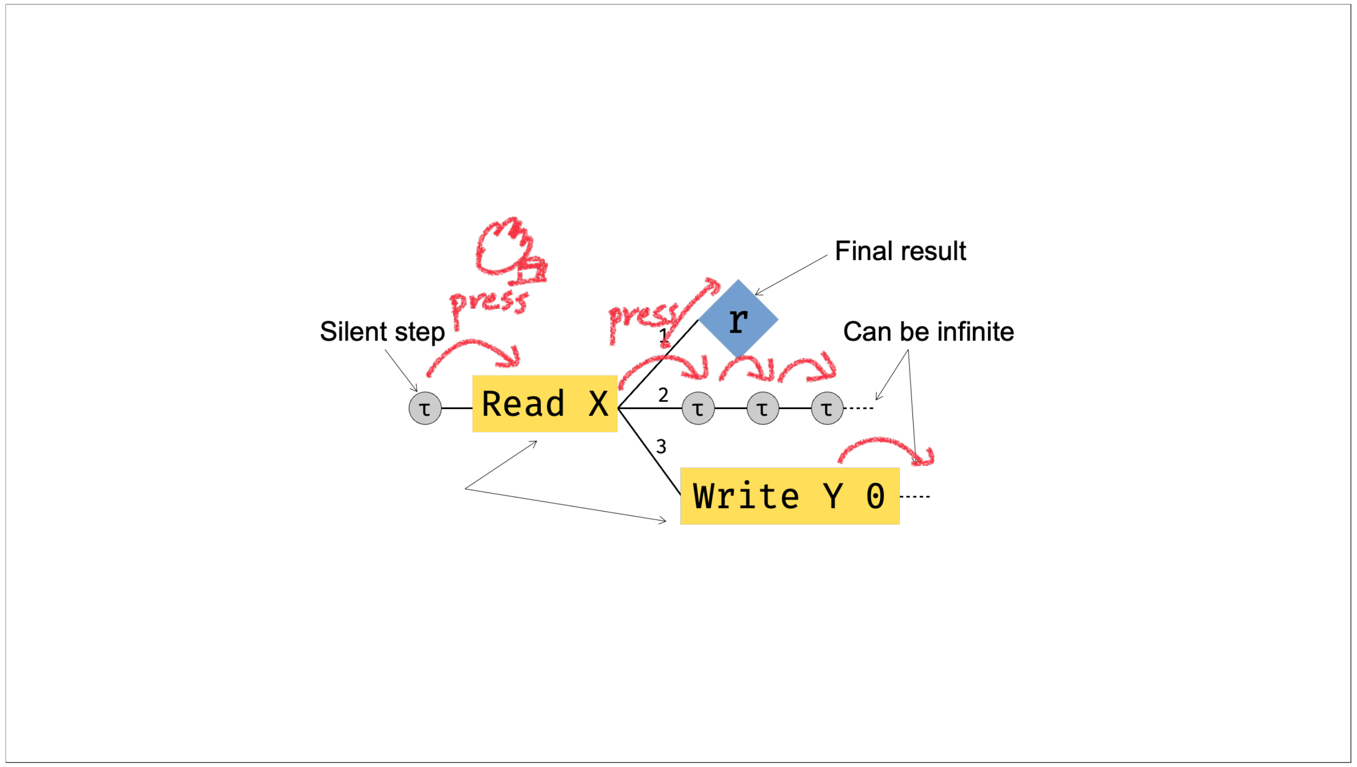

Example: list <-> colist. list: 1; 2 <-> colist: 1; 2. list: there’s 1, 2 inside the box. colist: when press the button, it emits 1, another box, then press the button on the new box, it outputs 2 and nothing.

```
Variant CoListF {this : Type} :=
| CoCons (hd: nat) (tl: this)
| CoNil.

CoInductive CoList : Type :=
  Press { emit : @CoListF CoList }.

Notation cocons x y := (Press (CoCons x y)).
Notation conil := (Press CoNil).
```

```
CoFixpoint flip_flop: CoList := cocons 1 (cocons 2 flip_flop).
```




flipflop: (show code), press once it outputs 1 and another box, press the button on the new box it outputs 0, and the first box. output seq: 1;0;1;0;… Well typed, pure, total, but infinite, because it doesn't generate value unless you press the button.

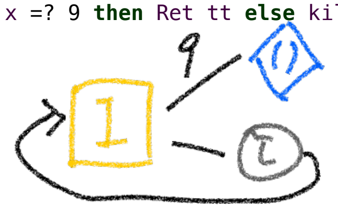



5. Tau: by expanding all taus, you got infinite computation trace. but if you don’t press it, it does nothing, i.e. terminating. Coq won’t complain about this!

# Examples of Trees

```
CoFixpoint echo : itree IO void :=
Vis Input (fun x ⇒ Vis (Output x) (fun _ ⇒ echo)).
```


```
CoFixpoint kill9 : itree IO unit :=
Vis Input (fun x ⇒ if x =? 9 then Ret tt else kill9).
```



# Equivalent relations

- Strong bisimulation: t1 ≅ t2 => exactly the same shape
- Weak bisimulation: tau t ≈ t
- Heterogeneous bisimulation:

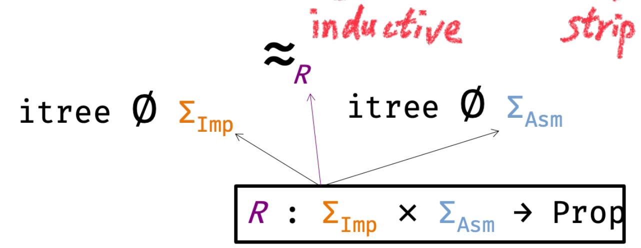
aRb  Ret a ≈ Ret b  match!

```
Inductive euttF (sim : itree E A →itree E B →Prop) : itree E A →itree E B →Prop :=
| EqRet a b (REL: r a b) : euttF sim (Ret a) (Ret b)
| EqVis {R} (e : E R) k1 k2 (REL: ∀v, sim (k1 v) (k2 v)) : euttF sim (Vis e k1) (Vis e k2)
| EqTau t1 t2 (REL: sim t1 t2) : euttF sim (Tau t1) (Tau t2)
| EqTauL t1 ot2 (REL: euttF sim t1 ot2) : euttF sim (Tau t1) ot2
| EqTauR ot1 t2 (REL: euttF sim ot1 t2) : euttF sim ot1 (Tau t2).
```

coinductive

$\tau\tau\tau \cdots t_1 \approx \tau\tau\tau \cdots t_2$

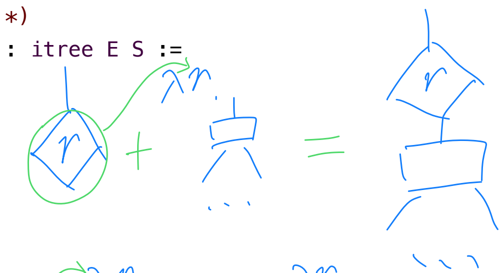
inductive  strip finite $\tau$'s



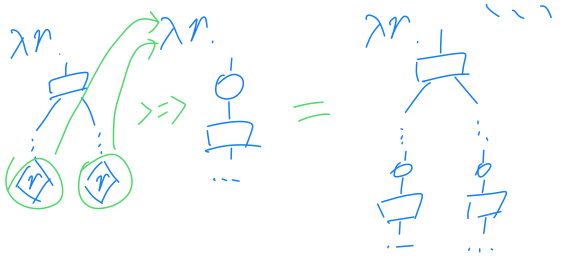

1. strong bisim: t1 ~== t2 when t1 and t2 have exactly the same shape
2. weak bisim: observe: tau t evaluates to the same value as t, so we want Equivalence Up To Tau. (give def on slides) define weak bisim t1 ~~ t2 with tau t = t, ONLY when removing finite number of taus (EqTauL & EqTauR are inductively defined, so they can only apply finite times). When it comes to inf taus, both ends should have inf taus. => weak bisim is termination sensitive.
3. heterogeneous bisim: compiler compiles a language of return type A to a language of return type B. How to reason about them? Given a relation to match A and B, define eutt r (equivalence up to tau modulo r), in which Ret a ~~R Ret b iff a R b. Theorem: If R is equiv rel, then ~~R is equiv rel. eutt is a special case of eutt mod r with R := leibniz equality.

# ITrees are compositional

```
(* Apply the continuation k to the Ret nodes of the itree t *)
Definition bind {E R S} (t : itree E R) (k : R → itree E S) : itree E S :=
(cofix bind_ u := match u with
| Ret r ⇒ k r
| Tau t ⇒ Tau (bind_ t)
| Vis e k ⇒ Vis e (fun x ⇒ bind_ (k x)) end) t.
Notation "x t1 ;; t2" := (bind t1 (fun x ⇒ t2)).
```




```
  (* Composition of KTrees *)
Definition cat {E} {A B C : Type}
: ktree E A B → ktree E B C → ktree E A C :=
fun h k ⇒ (fun a ⇒ bind (h a) k).
Infix ">>>" := cat
```



# ITrees are compositional

Monad Laws

```
       (x ← ret v ;; k x) ≅ (k v)
          (x ← t ;; ret x) ≅ t
(x ← (y ← s ;; t) ;; u) ≅ (y ← s ;; x ← t ;; u)
```

ret >=> k ≅ k left id

t >=> ret ≅ t right id

assoc

(s >=> t) >=> u ≅ s >=> t >=> u

Structural Laws

```
                   (Tau t) ≈ t
      (x ← (Tau t) ;; k) ≈ Tau (x ← t ;; k)
  (x ← (Vis e k1) ;; k2) ≈
          (Vis e (fun y ⇒ (k1 y) ;; k2))
```

eutt

lift tau

lift vis

Congruences

```
              t1 ≅ t2 →      Tau t1 ≅ Tau t2
              k1 ≈̂ k2 →    Vis e k1 ≈ Vis e k2
t1 ≈ t2  ∧  k1 ≈̂ k2 → bind t1 k1 ≈ bind t2 k2
```

(str

# ITrees are compositional

```
id_   : A → itree E A
cat   : (B → itree E C) →
        (A → itree E B) → (A → itree E C)
case_ : (A → tree E C) →
        (B → itree E C) → (A + B → itree E C)
inl_  : A → itree E (A + B)
inr_  : B → itree E (A + B)
pure  : (A → B) → (A → itree E B)
```

```
          id_ >>> k ≈̂ k
          k >>> id_ ≈̂ k
   (i >>> j) >>> k ≈̂ i >>> (j >>> k)
 pure f >>> pure g ≈̂ pure (f ∘ g)
inl_ >>> case_ h k ≈̂ h
inr_ >>> case_ h k ≈̂ k
(inl_ >>> f) ≈̂ h ∧ (inr_ >>> f) ≈̂ k →
                  f ≈̂ case_ h k
```

Proof of correctness needs coinductive reasoning, but done by ITree authors. Users just rewrite use thms.

# Recap: State

| | |
|---|---|
| | (m, v) |
| x <- 1; | (_, 1) |
| set x; | (1, ()) |
| x <- get; | (1, 1) |
| x + 1 | (1, 2) |

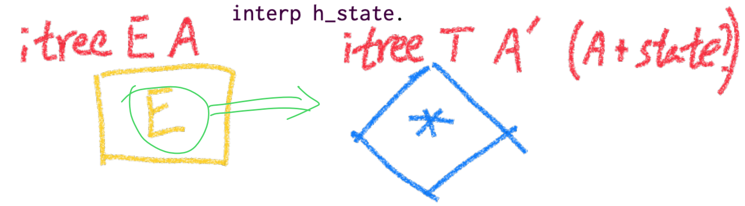
# State effect handler

```
(* The type of
     state events *)
Variant stateE (S : Type)
  : Type → Type :=
| Get : stateE S S
| Put : S → stateE S unit.
```

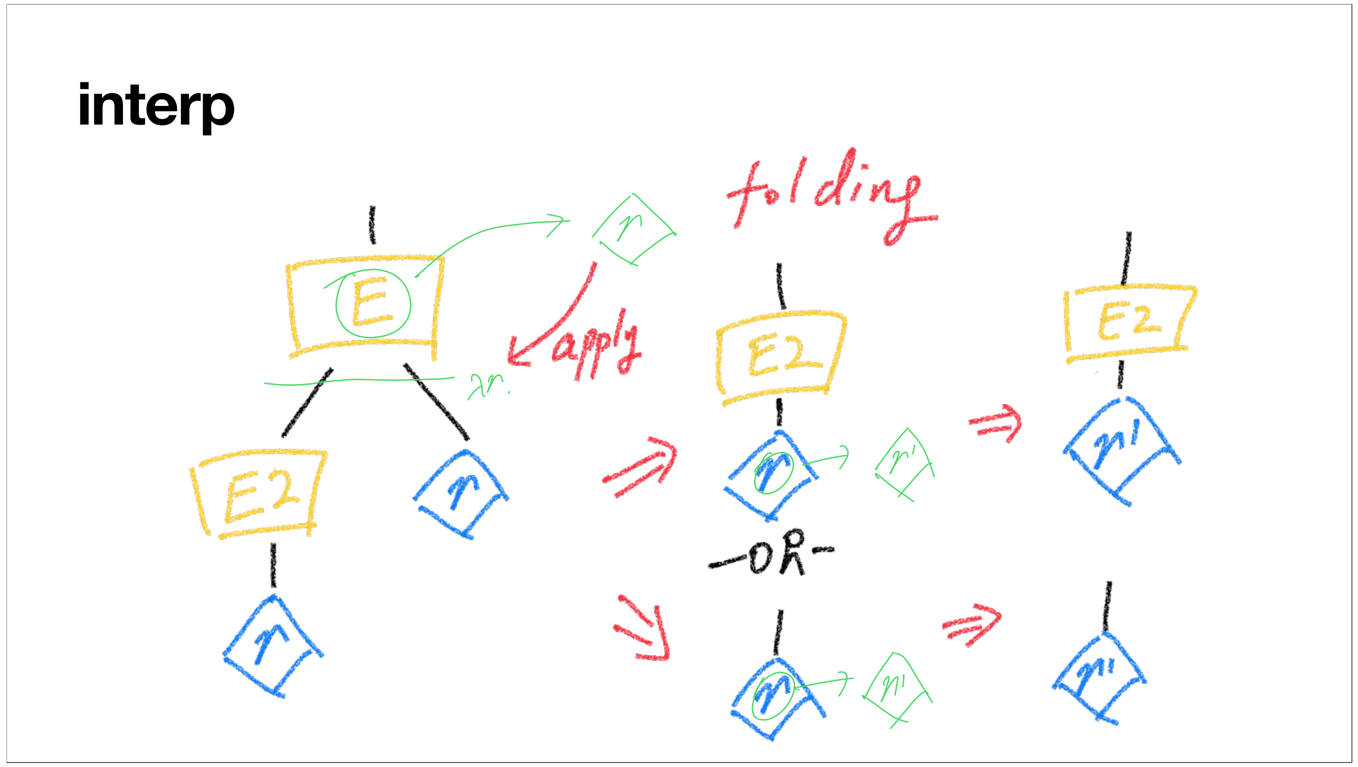
```
(* State monad transformer *)
Definition stateT (S:Type) (M:Type → Type) (R:Type) : Type :=
  S → M (S * R).
Definition getT (S:Type) : stateT S M S := fun s ⇒ ret (s, s).
Definition putT (S:Type) : S → stateT S M unit :=
  fun s' s ⇒ ret (s', tt).
```

```
(* Handler for state events *)
Definition h_state (S:Type) {E}
  : (stateE S) ↝ stateT S (itree E) :=
  fun _ e ⇒ match e with
            | Get   ⇒ getT S
            | Put s ⇒ putT S s
            end.
```

```
(* Interpreter for state events *)
Definition interp_state {E S}
  : itree (stateE S) ↝ stateT S (itree E) :=
  interp h_state.
```




effect handler: convert a itree with effects into one with no effect and modified value (state, value)
(slide: show tree example)




3. interp function: take eff, take ITree E A, output ITree TT A'. (slide: graph repr of what the function does) interp is folding the tree, transforming all nodes into new ret type, and replace Vis with handler call & bind.

# How to “fold” an ITree?

- Define iter := (A → M (A + B)) → A → M B
- A: continue loop | B: break

```
Definition interp {E M : Type → Type} `{MonadIter M} {R : Type} (handler : E ⤳ M)
  : itree E R → M R := iter (fun t : itree E R ⇒
                               match t with
                               | Ret r   ⇒ ret (inr r)
                               | Tau t   ⇒ ret (inl t)
                               | Vis e k ⇒ bind (handler _ e) (fun a ⇒ ret (inl (k a)))
                               end).
```

pure value: no effect left, break

τ: t still need to be transformed, continue

Vis: still need to transform kont, continue

How to represent the “fold” concept? introduce iter, show its signature. return to this later.

# Effect combinators & properties

```
id_   : E ↝ itree E          (* trigger *)
cat   : (F ↝ itree G) →      (* interp *)
        (E ↝ itree F) → (E ↝ itree G)
case_ : (E ↝ itree G) →
inl_
inr_
```

```
Definition case_ {E F M} : (E ↝ M) → (F ↝ M) → (E +' F) ↝ M
  := fun f g _ e ⇒ match e with
                   | inl1 e1 ⇒ f _ e1
                   | inr1 e2 ⇒ g _ e2
                   end.

Definition cat {E F G} : (E ↝ itree F) → (F ↝ itree G) → (E ↝ itree G)
  := fun f g _ e ⇒ interp g (f _ e).
```

```
interp_state (x ← get ;; y ← get ;; k x y) s ≈ interp_state (x ← get ;; k x x) s
```

preserve structure

```
interp h (trigger e)    ≅ h _ e
interp h (Ret r)        ≅ ret r
interp h (x ← t;; k x) ≅
  x ← (interp h t);; interp h (k x)
```

4. there are many interp combinators, and they still have good properties. (slide: show combinators & props) You can reason about non-trivial things with them like a poor version of useless load elimination

Next: non-terminating structure

# Iteration

- Define iter := (A → M (A + B)) → A → M B
- A: continue loop | B: break

```
CoFixpoint iter (body : A → itree E (A + B))
  : A → itree E B :=
  fun a ⇒ ab ← body a ;;
           match ab with
           | inl a ⇒ Tau (iter body a)
           | inr b ⇒ Ret b
           end.
```

reminder: coinductive dt, no press, no expand, so terminating

# Iteration

```
CoFixpoint iter (body : A → itree E (A + B))
  : A → itree E B :=
  fun a ⇒ ab ← body a ;;
           match ab with
           | inl a ⇒ Tau (iter body a)
           | inr b ⇒ Ret b
           end.
```

- Does not rely on shape of the body
- No guardedness check

does not rely on body shape, no guard check

# Properties of Iteration

| | | | |
|---|---|---|---|
| `iter f` | $\hat{\approx}$ | `f >>> case_ (iter f) id_` | *(fixed point)* |
| `iter f >>> g` | $\hat{\approx}$ | `iter (f >>> bimap id_ g)` | *(parameter)* |
| `iter (f >>> case_ g inr_)` | $\hat{\approx}$ | `f >>> case_ (iter (g >>> case_ f inr_)) id_` | *(composition)* |
| `iter (iter f)` | $\hat{\approx}$ | `iter (f >>> case_ inl_ id_)` | *(codiagonal)* |

$$\text{iter } f \mathrel{\hat{\approx}} f ; \text{iter } f$$

$$\text{iter } f ; g \mathrel{\hat{\approx}} \text{iter } f ; f ; g$$

$$\text{iter } (f ; g) \mathrel{\hat{\approx}} f ; \text{iter } (g ; f)$$

$$\text{iter } (\text{iter } f) \mathrel{\hat{\approx}} \text{iter } f$$

many good properties

# Recursion

## A special kind of *effect*

```
Inductive ackermannE : Type → Type :=
| Ackermann : nat → nat → ackermannE nat.
```

still present ?!

```
Definition h_ackermann : ackermannE ~> itree (ackermannE +' emptyE) :=
  fun _ '(Ackermann m n) ⇒ if m =? 0 then Ret (n + 1)
                          else if n =? 0 then trigger (inl1 (Ackermann (m-1) 1))
                          else (ack ← trigger (inl1 (Ackermann m (n-1))) ;;
                                trigger (inl1 (Ackermann (m-1) ack))).
```

Recursion effects: D-> itree (D + ‘E) **Can make recursive calls**

Normal effects: D -> itree ‘E

represented by eff
rec effect vs normal eff: rec effect D -> itree (D + ‘E), it returns an ITree with itself present so can make recursive calls, while normal eff looks like D -> itree ‘E, no recursive calls

# mrec

- **mrec** is to recursive effects what **interp** is to normal effects

gone

```
(* Interpret an itree in the context of a mutually recursive definition (rh) *)
Definition mrec {D E} (rh : D ~> itree (D +' E)) : D ~> itree E :=
  fun R d => iter (fun t : itree (D +' E) R =>
              match t with
              | Ret r => Ret (inr r)
              | Tau t => Ret (inl t)
              | Vis (inl1 d) k => Ret (inl (bind (rh _ d) k))      recursive eff
              | Vis (inr1 e) k => bind (trigger e) (fun x => Ret (inl (k x)))
              end) (rh _ d).                                       normal eff

Definition ackermann : nat -> nat -> itree emptyE nat :=
  fun m n => mrec h_ackermann (Ackermann m n).
```

mrec as to recursion effect is interp as to normal effs

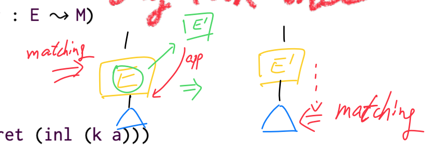
# mrec vs interp

```
(* Interpret an itree in the context of a mutually recursive definition (rh) *)
Definition mrec {D E} (rh : D ⤳ itree (D +' E)) : D ⤳ itree E :=
  fun R d ⇒ iter (fun t : itree (D +' E) R ⇒
              match t with
              | Ret r ⇒ Ret (inr r)
              | Tau t ⇒ Ret (inl t)
              | Vis (inl1 d) k ⇒ Ret (inl (bind (rh _ d) k))
              | Vis (inr1 e) k ⇒ bind (trigger e) (fun x ⇒ Ret (inl (k x)))
              end) (rh _ d).
```

Stay on the same node


```
Definition interp {E M : Type → Type} `{MonadIter M} {R : Type} (handler : E ⤳ M)
  : itree E R → M R := iter (fun t : itree E R ⇒
                              match t with
                              | Ret r   ⇒ ret (inr r)
                              | Tau t   ⇒ ret (inl t)
                              | Vis e k ⇒ bind (handler _ e) (fun a ⇒ ret (inl (k a)))
                              end).
```

Only look once



## mrec is a fixpoint

### … by an unfolding equation

```
mrec rh ≈ interp (mrec rh) rh
```

mrec is a fixpoint by an unfolding equation

# What does ITree enable us to do?

## Compiler correctness

$$
\begin{tikzcd}
\textsc{Imp} \arrow[r, "\text{compile}"] \arrow[d, "\text{denote}"'] & \textsc{Asm} \arrow[d] \\
\texttt{itree}\ E_{\texttt{Imp}}\ \texttt{unit} \arrow[d, "\text{interpret}"'] & \texttt{itree}\ E_{\texttt{Asm}}\ \texttt{unit} \arrow[d] \\
\texttt{itree}\ \emptyset\ \Sigma_{\texttt{Imp}} \arrow[r, phantom, "\approx_R"] & \texttt{itree}\ \emptyset\ \Sigma_{\texttt{Asm}}
\end{tikzcd}
$$

1. pre: define two languages, define compiler function
2. define their semantics by (syntax-directed) denote: denote imp -> ITree ImpMemE (), asm -> ITree (AsmRegE + AsmMemE) ()
3. given eh of ImpMemE, AsmRegE, AsmMemE, we can define interp_imp, interp_asm by using interp combinators
5. define match relation between imp state * value and asm state * value, now we have weak bisim. compiler correctness thm defined

# What does ITree enable us to do?

## Compiler correctness

$$
\begin{tikzcd}
\textsc{Imp} \arrow[r, "\text{compile}"] \arrow[d, "\text{denote}"'] & \textsc{Asm} \arrow[d] \\
\texttt{itree}\ E_{\texttt{Imp}}\ \texttt{unit} \arrow[d, "\text{interpret}"'] & \texttt{itree}\ E_{\texttt{Asm}}\ \texttt{unit} \arrow[d] \\
\texttt{itree}\ \emptyset\ \Sigma_{\texttt{Imp}} \arrow[r, phantom, "\approx_R"] & \texttt{itree}\ \emptyset\ \Sigma_{\texttt{Asm}}
\end{tikzcd}
$$

- Proof by equiv rewrites
- ~5k lines (2k loc w/ docs)
- Can be automated

proof by equiv rewrites. might be automated by equality saturation, just like peephole optim. hand-written version 5k lines, (including def & semantics def, should be 2k lines without comments)

# What does ITree enable us to do?

## Extract to OCaml

```
CoFixpoint echo : itree IO void :=
Vis Input (fun x ⇒ Vis (Output x) (fun _ ⇒ echo)).
```

```
let rec echo =
  lazy (Vis (Input, (fun x -> lazy (Vis ((Output (Obj.magic x)), (fun _ ->
    echo))))))
(* OCaml handler -----(not extracted) ------------------------------------------ *)
let handle_io e k = match e with
  | Input -> k (Obj.magic (read_int ()))
  | Output x -> print_int x ; k (Obj.magic ())
let rec run t =
  match observe t with
  | Ret r -> r
  | Tau t -> run t
  | Vis (e, k) -> handle_io e (fun x -> run (k x))
```

10. Example: extract itree to ocaml, get reference intepreter for free

# What does ITree enable us to do?

## Extract to OCaml

- Reference interpreter for free
- Support side effects not implementable in Coq (network IO, etc)
- Fuzzing

1. implement eh to do side effects not possible in coq (network IO)
2. fuzzer, fuzz your semantics before proof (next slide: avoid retakes)

# What does ITree enable us to do?

## Extract to OCaml

- Add new feature to semantics
- Try to prove (took months)
- Oops! Feature unsound!!
- Rework the semantics…
- Retake the proof (took months)
- Oops! Still unsound!!
- Months wasted…🤯

VS

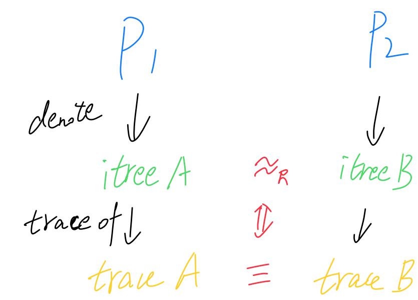
- Add new feature to semantics
- Extract & fuzz the interpreter (in one day)
- Oops! Unexpected output!
- Rework the semantics…
- Extract & fuzz the interpreter (in one day)
- Oops! Unexpected output!
- …
- We believe this semantics should be right!
- Try to prove (took months)
- Done! 🎉

# What does ITree enable us to do?

## Trace semantics

Trace: a sequence of events emitted by the execution of a program

```
Inductive trace (E : Type → Type) (R : Type) : Type :=
| TEnd : trace E R
| TRet : R → trace E R
| TEventEnd : ∀{X}, E X → trace E R
| TEventResponse : ∀{X}, E X → X → trace E R → trace E R.

Inductive is_trace_of {E : Type → Type} {R : Type} :
  itree E R → trace E R → Prop :=
| TraceEmpty : ∀t, is_trace_of t TEnd
| TraceRet : ∀r, is_trace_of (Ret r) (TRet r)
| TraceTau : ∀t tr, is_trace_of t tr → is_trace_of (Tau t) tr
| TraceVisEnd : ∀X (e : E X) k, is_trace_of (Vis e k) (TEventEnd e)
| TraceVisContinue : ∀X (e : E X) (x : X) k tr,
    is_trace_of (k x) tr → is_trace_of (Vis e k) (TEventResponse e x tr).
```




*Definition 1 (Trace Refinement).* $t \sqsubseteq u$ iff $\forall$ tr, is_trace_of t tr → is_trace_of u tr.

*Definition 2 (Trace Equivalence).* $t \equiv u$ iff $t \sqsubseteq u$ and $u \sqsubseteq t$.

Using these definitions, we can show that trace equivalence coincides with weak bisimulation, *i.e.*, that $t1 \approx t2 \iff t1 \equiv t2$.

11. relation with good old trace semantics
    1. compcert verify programs by step: st -> ev -> st -> Prop, execute program got a trace
    2. can also extract trace from itree, good property: weak sim <-> trace reequiv
    3. able to reason nondeterministic behavior (next slide)

# What does ITree enable us to do?

## Trace semantics

Trace: a sequence of events emitted by the execution of a program

*Definition 1 (Trace Refinement).* $\mathtt{t} \sqsubseteq \mathtt{u}$ iff $\forall$ $\mathtt{tr}$, $\mathtt{is\_trace\_of\ t\ tr} \rightarrow \mathtt{is\_trace\_of\ u\ tr}$.

*Definition 2 (Trace Equivalence).* $\mathtt{t} \equiv \mathtt{u}$ iff $\mathtt{t} \sqsubseteq \mathtt{u}$ and $\mathtt{u} \sqsubseteq \mathtt{t}$.

Using these definitions, we can show that trace equivalence coincides with weak bisimulation, *i.e.*, that $\mathtt{t1} \approx \mathtt{t2} \iff \mathtt{t1} \equiv \mathtt{t2}$.

- Reason about non-deterministic side effects

# Conclusion

- ITree: a foundation for program semantics and an equational theory
- A shallow representation of non-terminating effectful languages, leveraging the nature of coinductive types
- Leverage existing power of meta language (proof assistants), simplifying proof engineering
- Proof by eq rewrite, room for automatic reasoning
- Extract to executable programs, enable “swift” development of formal semantics

# Future work

- Non-determinism & concurrency (multiple followups)
- Relate its theorem with domain theories, operational semantics, and game semantics
- Does not work when the state match relations is not one-to-one (impossible to formalize most practical languages, e.g. Clight)

# Interaction Trees

## A denotational semantics and its equational theorems