From Representing Recursive and Impure Programs in Coq to a Modular Formal Semantics of LLVM IR

Yannick Zakowski

Introduction:
The DeepSpec NSF Expedition

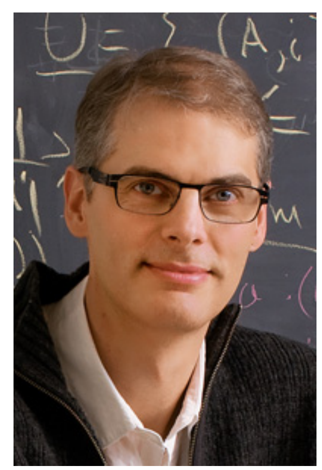
A Cross-Institutions Enterprise…

Massachusetts Institute of Technology

PRINCETON UNIVERSITY

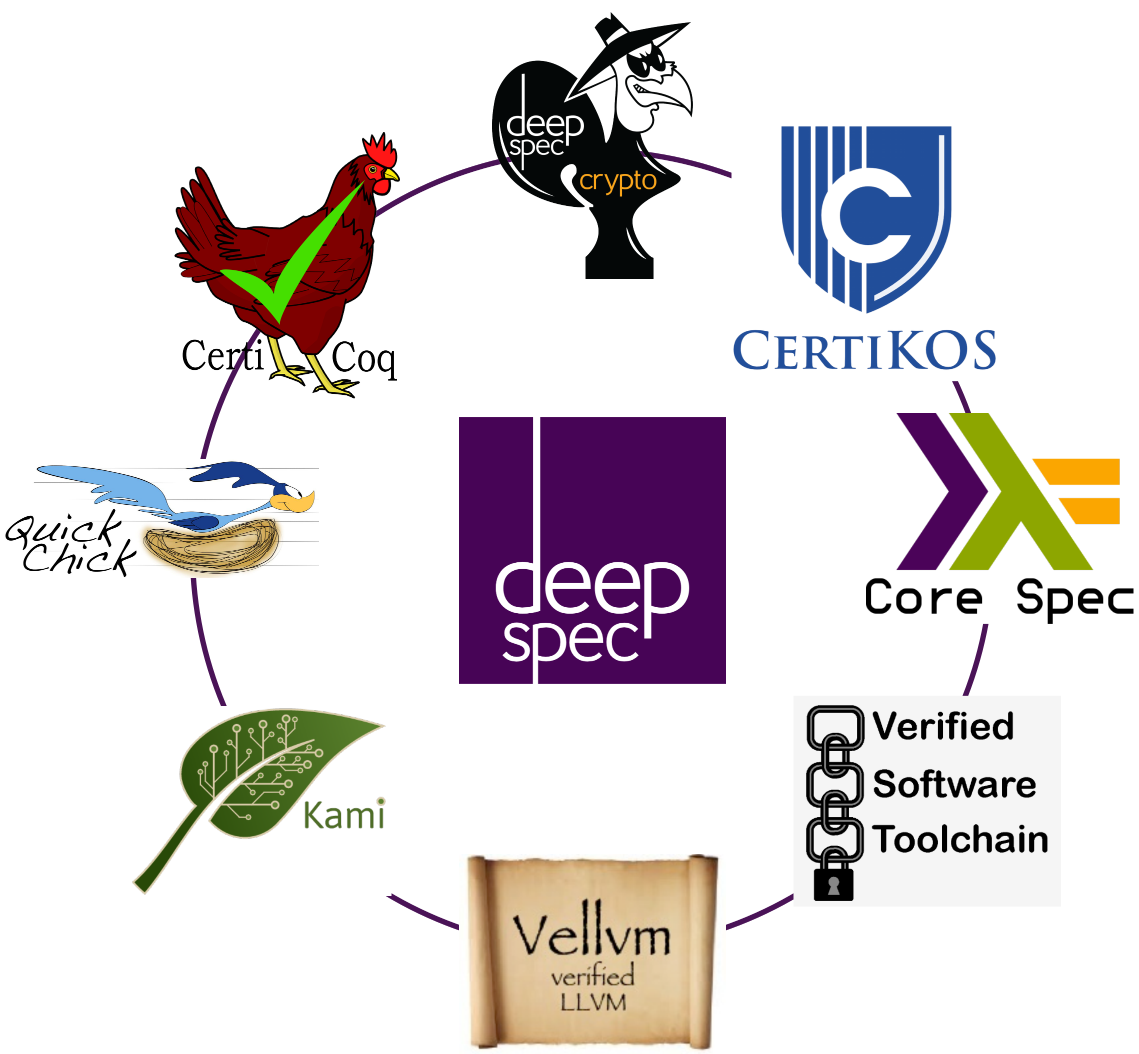
Penn University of Pennsylvania

Yale



Chlipala **Appel** **Berenger** **Pierce** **Wierich** **Zdancewic** **Shao**

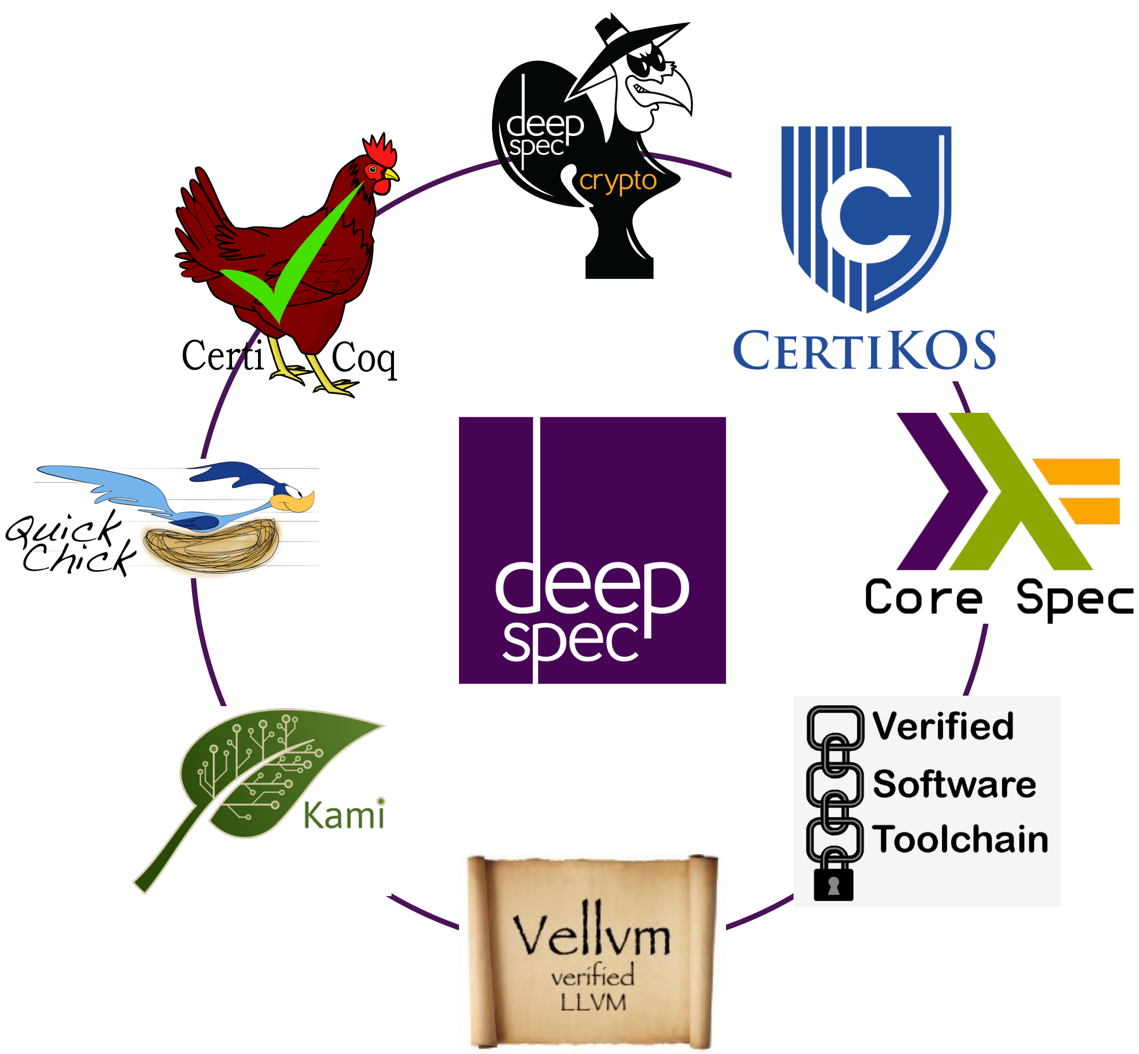
… Encompassing a Variety of Projects!



Specifications with a shared philosophy

- *Rich*
  More than functional specification
- *Live*
  Connected to executable artifacts
- *Formal*
  Ideally, machine-checked
- *Two-sided*
  Interfaced to both client and implementation

Ambition: Full Stack Verified Artifacts



Beyond a shared philosophy: combining these efforts

Kami + CertiKOS + VST + QuickChick
=
Verified Web Server?

deep spec server

Ambition: Full Stack Verified Artifacts

Property-based testing in Coq

Decidable Gallina functions

Toolchain to prove properties of compiled C programs

Separation Logic + CompCert

Verified OS Kernel

Certified Abstraction Layers

Framework for verified Blue-Spec-style components

Labelled Transition Systems

Ambition: Full Stack Verified Artifacts

Property-based testing in Coq

Decidable Gallina functions

Toolchain to prove properties of compiled C programs

Separation Logic + CompCert

Verified OS Kernel

Certified Abstraction Layers

Framework for verified Blue-Spec-style components

Labelled Transition Systems

Swap Server (now)

HTTP Server (ongoing)

Ambition: Full Stack Verified Artifacts

Property-based testing in Coq

Decidable Gallina functions

Toolchain to prove properties of compiled C programs

Separation Logic + CompCert

Verified OS Kernel

Certified Abstraction Layers

Framework for verified Blue-Spec-style components

Labelled Transition Systems

Swap Server (now)

HTTP Server (ongoing)

Specification could use a Franca Lingua!

Interaction Trees: Representing Recursive and Impure Programs in Coq

Cahiers de Doléances

Able to model very diverse impure specifications

A C-implementation of a web-server

The interface exposed by CertiKOS

Easily linked to executable implementation

Testing specifications

Verified *executed* web-server

Convenient source of definitional interpreters

Cahiers de Doléances

Formalised in the Coq Proof Assistant

Strongly normalizing: how to represent divergence?

Pure: how to represent effects?

Amenable to large scale proofs

Modular specification

Equational reasoning

Practical library

Cahiers de Doléances

Specification of impure computations in the Coq proof assistant supporting extraction and modular reasoning

Interaction Trees

Representing Recursive and Impure Programs in Coq

LI-YAO XIA, University of Pennsylvania, USA
YANNICK ZAKOWSKI, University of Pennsylvania, USA
PAUL HE, University of Pennsylvania, USA
CHUNG-KIL HUR, Seoul National University, Republic of Korea
GREGORY MALECHA, BedRock Systems, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA
STEVE ZDANCEWIC, University of Pennsylvania, USA

Cahiers de Doléances

Specification of impure computations in the Coq proof assistant supporting extraction and modular reasoning

Interaction Trees

DISTINGUISHED PAPER

Representing Recursive and Impure Programs in Coq

LI-YAO XIA, University of Pennsylvania, USA
YANNICK ZAKOWSKI, University of Pennsylvania, USA
PAUL HE, University of Pennsylvania, USA
CHUNG-KIL HUR, Seoul National University, Republic of Korea
GREGORY MALECHA, BedRock Systems, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA
STEVE ZDANCEWIC, University of Pennsylvania, USA

Interaction Trees

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

A value of the datatype (itree E R) represents:

Interaction Trees

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

A value of the datatype (itree E R) represents:

Interaction Trees

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

A value of the datatype (itree E R) represents:

- a potentially diverging computation,

Interaction Trees

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

A value of the datatype (itree E R) represents:

- a potentially diverging computation,
- which may return a value of type R,

Interaction Trees

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

A value of the datatype (itree E R) represents:

- a potentially diverging computation,
- which may return a value of type R,
- while emitting during its execution events from the interface E.

Interaction Trees

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

A value of the datatype (itree E R) represents:

- a potentially diverging computation,
- which may return a value of type R,
- while emitting during its execution events from the interface E.

Interaction Trees

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

A value of the datatype (itree E R) represents:

- a potentially diverging computation,
- which may return a value of type R,
- while emitting during its execution events from the interface E.

Interaction Trees

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

A value of the datatype (itree E R) represents:

- a potentially diverging computation,
- which may return a value of type R,
- while emitting during its execution events from the interface E.

Relates to many existing works in the litterature:
* Composible effects: Kiselyov & Ishii's Freer monad
* Partial function in type theory: Capretta's Delay monad
* Effectful computations in Type Theory: Hancock, McBride's general monad
* Effectful Programs in Coq: Letan & Gianas's FreeSpec

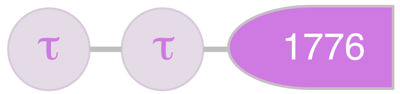
ITrees Come in All Shapes and Forms

ITrees Come in All Shapes and Forms

Pure computations

Ret 1789

Tau (Tau (Ret 1776))

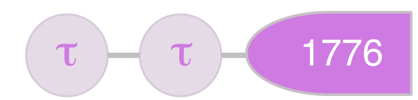


ITrees Come in All Shapes and Forms

Pure computations

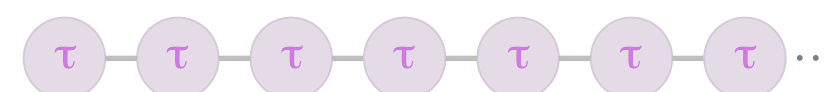
`Ret 1789`

`Tau (Tau (Ret 1776))`



Silent divergence

`CoFixpoint spin := Tau spin`

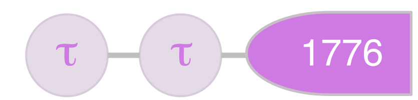


ITrees Come in All Shapes and Forms

Pure computations

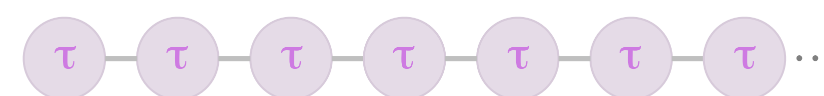
```
Ret 1789
```


```
Tau (Tau (Ret 1776))
```



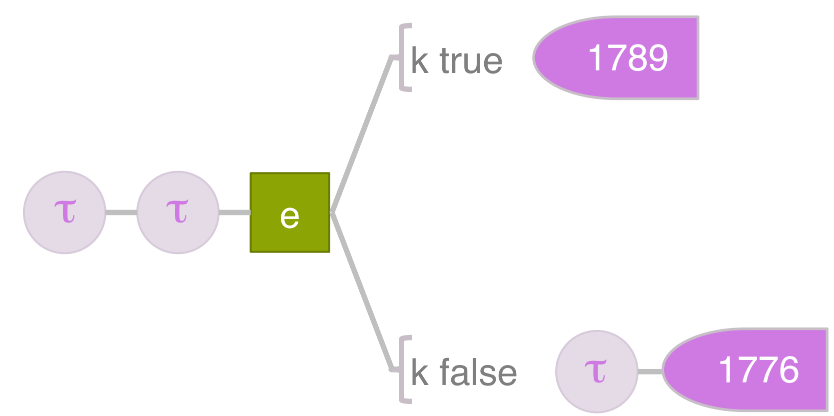
Silent divergence

```
CoFixpoint spin := Tau spin
```

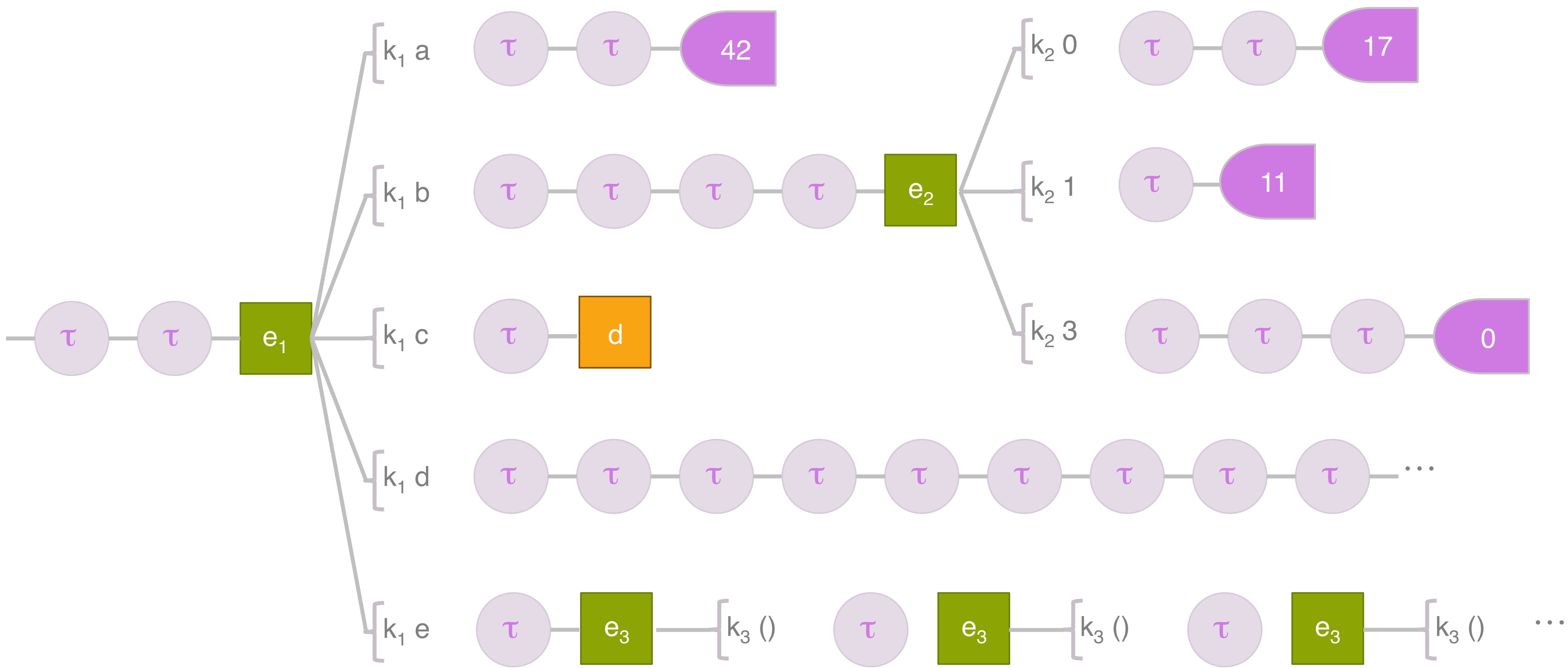


Effectful computation

```
Tau (Tau (Vis e
  (fun b => match b with
       | true => Ret 1789
       | false => Tau (Ret 1776)
   end)))
```



ITrees Come in All Shapes and Forms



d **Failure: event of return type void**

Composing Computations: the ITree Monad

Monadic structure

```
Definition ret {X: Type} (x: X): itree E X := Ret x

CoFixpoint bind {R S} (t: itree E R) (k: R -> itree E S): itree E S :=
```

Composing Computations: the ITree Monad

Monadic structure

```
Definition ret {X: Type} (x: X): itree E X := Ret x

CoFixpoint bind {R S} (t: itree E R) (k: R -> itree E S): itree E S :=
  match t with
   | Ret r    => k r
   | Tau t    => Tau (bind t k)
   | Vis e h => Vis e (fun x => bind (h x) k)
  end.
```

Composing Computations: the ITree Monad

Monadic structure

```
Definition ret {X: Type} (x: X): itree E X := Ret x

CoFixpoint bind {R S} (t: itree E R) (k: R -> itree E S): itree E S :=
  match t with
   | Ret r    => k r
   | Tau t    => Tau (bind t k)
   | Vis e h => Vis e (fun x => bind (h x) k)
  end.
```

Notation:
x <- s ;; k
≜
bind s (fun x => k)

Composing Computations: the ITree Monad

Monadic structure

```
Definition ret {X: Type} (x: X): itree E X := Ret x

CoFixpoint bind {R S} (t: itree E R) (k: R -> itree E S): itree E S :=
   match t with
    | Ret r    => k r
    | Tau t    => Tau (bind t k)
    | Vis e h => Vis e (fun x => bind (h x) k)
   end.
```

Notation:

x <- s ;; k

$\triangleq$

bind s (fun x => k)

Monad laws:

ret_bind:	x <- ret v ;; k x	$\approx$	k v
bind_ret:	x <- t ;; ret x	$\approx$	t
bind_bind:	x <- (y <- s ;; t) ;; u	$\approx$	y <- s ;; x <- t ;; u

Composing Computations: the ITree Monad

Monadic structure

```
Definition ret {X: Type} (x: X): itree E X := Ret x

CoFixpoint bind {R S} (t: itree E R) (k: R -> itree E S): itree E S :=
  match t with
   | Ret r   => k r
   | Tau t   => Tau (bind t k)
   | Vis e h => Vis e (fun x => bind (h x) k)
  end.
```

Notation:

x <- s ;; k

$\triangleq$

bind s (fun x => k)

Monad laws:

ret_bind:	x <- ret v ;; k x	$\approx$	k v
bind_ret:	x <- t ;; ret x	$\approx$	t
bind_bind:	x <- (y <- s ;; t) ;; u	$\approx$	y <- s ;; x <- t ;; u

ITree equivalence?

ITree Equivalence

Option 1: Coq's propositional equality?

$$t \approx s \triangleq t = s$$

```
Inductive eq {X: Type}: Prop :=
  | eq_refl: forall (x: X), eq x x.
```

ITree Equivalence

Option 1: Coq's propositional equality?

$$t \approx s \triangleq t = s$$

```
Inductive eq {X: Type}: Prop :=
  | eq_refl: forall (x: X), eq x x.
```

$\nvdash$ spin = Tau spin

ITree Equivalence

Option 2: Strong bisimulation?

ITree Equivalence

Option 2: Strong bisimulation?

$$t \approx s \triangleq \text{bisim } t\ s$$

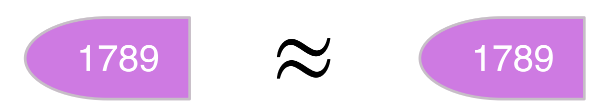
```
Inductive bisimF (sim: relation (itree E R)): relation (itree E R) :=

  | EqRet: bisimF (Ret v) (Ret v)

  | EqTau: sim t s  -> bisimF sim (Tau t) (Tau s)

  | EqVis (e: E X): (forall (v: X), sim (k1 v) (k2 v))
                    -> bisimF sim (Vis e k1) (Vis e k2)
```

ITree Equivalence

Option 2: Strong bisimulation?

$$t \approx s \triangleq \text{bisim } t\ s$$

```
Inductive bisimF (sim: relation (itree E R)): relation (itree E R) :=

  | EqRet: bisimF (Ret v) (Ret v)

  | EqTau: sim t s  -> bisimF sim (Tau t) (Tau s)

  | EqVis (e: E X): (forall (v: X), sim (k1 v) (k2 v))
                        -> bisimF sim (Vis e k1) (Vis e k2)
```

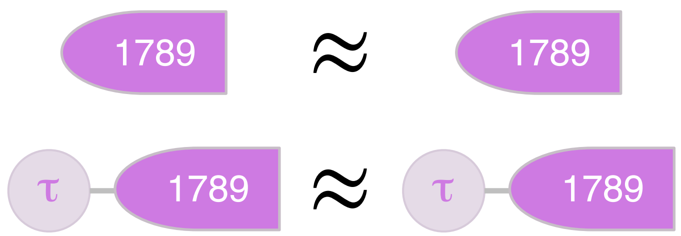


ITree Equivalence

Option 2: Strong bisimulation?

t ≈ s ≜ bisim t s

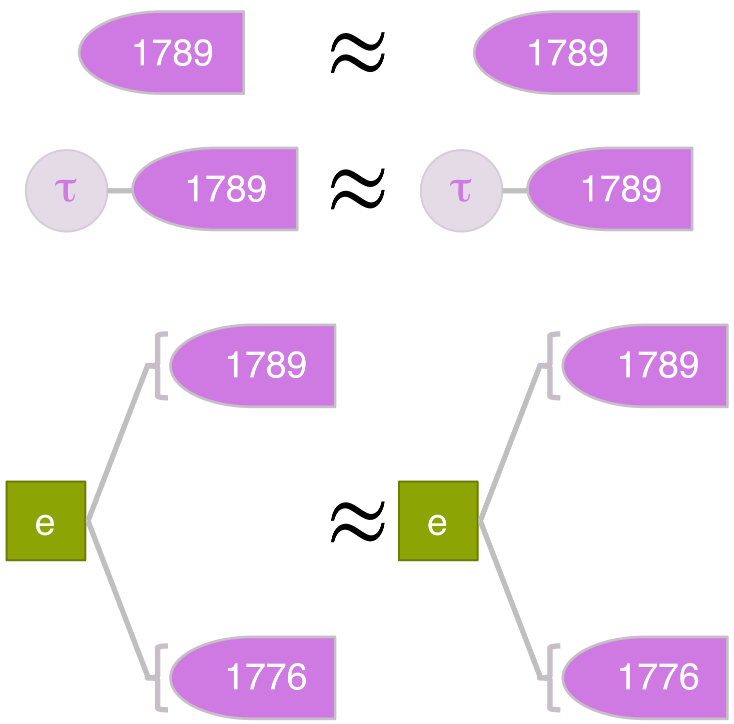
```
Inductive bisimF (sim: relation (itree E R)): relation (itree E R) :=

  | EqRet: bisimF (Ret v) (Ret v)

  | EqTau: sim t s  -> bisimF sim (Tau t) (Tau s)

  | EqVis (e: E X): (forall (v: X), sim (k1 v) (k2 v))
                          -> bisimF sim (Vis e k1) (Vis e k2)
```



ITree Equivalence

Option 2: Strong bisimulation?

```
t ≈ s ≜ bisim t s

Inductive bisimF (sim: relation (itree E R)): relation (itree E R) :=

  | EqRet: bisimF (Ret v) (Ret v)

  | EqTau: sim t s -> bisimF sim (Tau t) (Tau s)

  | EqVis (e: E X): (forall (v: X), sim (k1 v) (k2 v))
                     -> bisimF sim (Vis e k1) (Vis e k2)
```

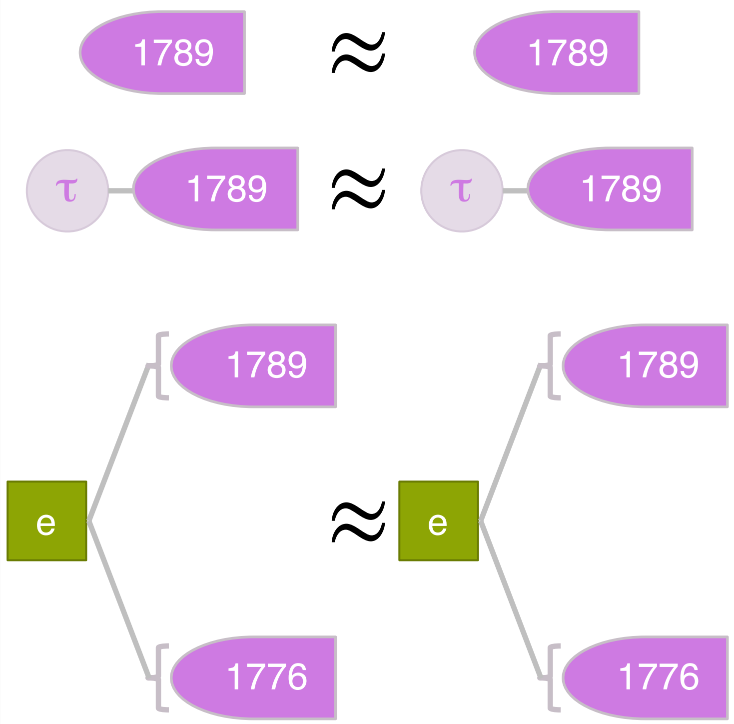


ITree Equivalence

Option 2: Strong bisimulation?

t ≈ s ≜ bisim t s

```
Inductive bisimF (sim: relation (itree E R)): relation (itree E R) :=

  | EqRet: bisimF (Ret v) (Ret v)

  | EqTau: sim t s  -> bisimF sim (Tau t) (Tau s)

  | EqVis (e: E X): (forall (v: X), sim (k1 v) (k2 v))
                          -> bisimF sim (Vis e k1) (Vis e k2)
```



ITree Equivalence

Option 2: Strong bisimulation?

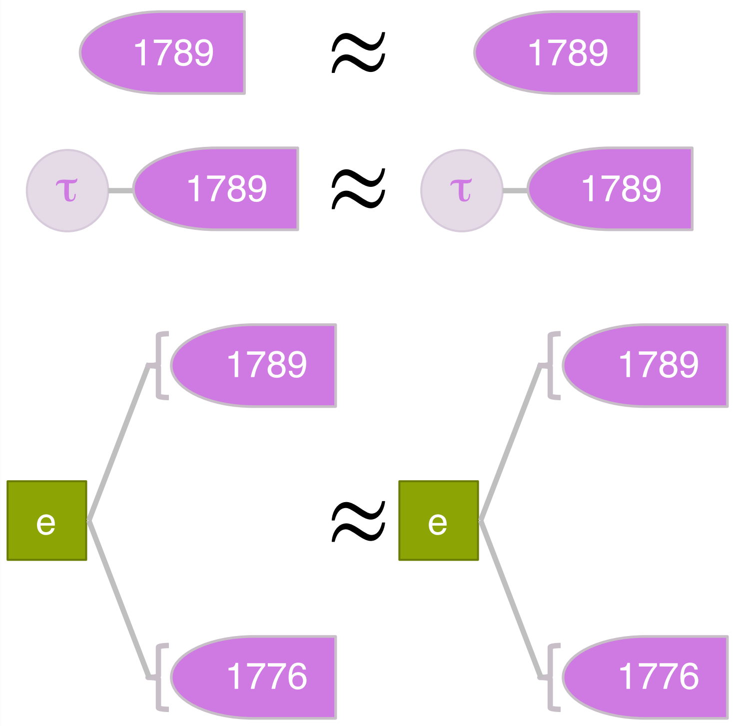
t ≈ s ≜ bisim t s

Inductive bisimF (sim: relation (itree E R)): relation (itree E R) :=

| EqRet: bisimF (Ret v) (Ret v)

| EqTau: sim t s -> bisimF sim (Tau t) (Tau s)

| EqVis (e: E X): (forall (v: X), sim (k1 v) (k2 v))
-> bisimF sim (Vis e k1) (Vis e k2)



bisim t s ≜ paco bisimF bot

https://github.com/snu-sf/paco

ITree Equivalence

Option 2: Strong bisimulation?

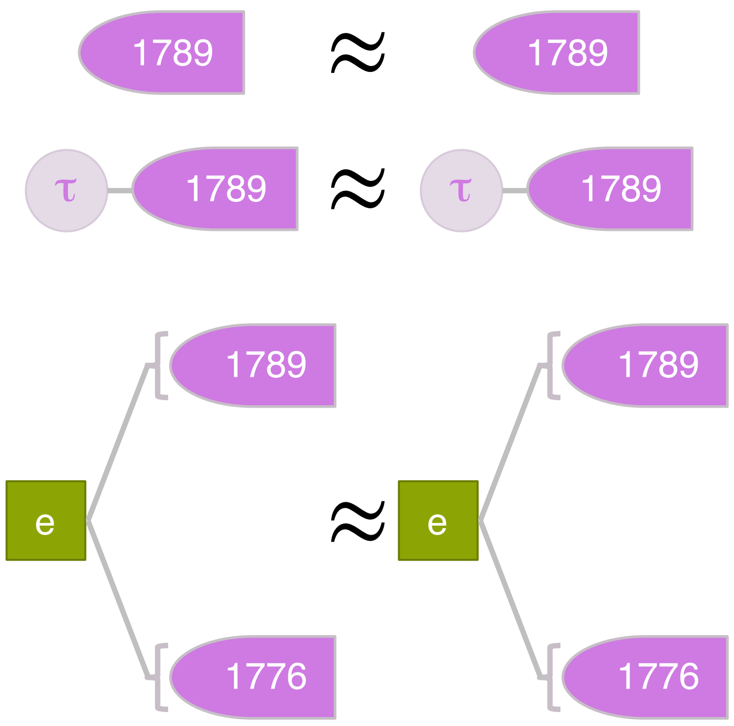
t ≈ s ≜ bisim t s

```
Inductive bisimF (sim: relation (itree E R)): relation (itree E R) :=
  | EqRet: bisimF (Ret v) (Ret v)
  | EqTau: sim t s  -> bisimF sim (Tau t) (Tau s)
  | EqVis (e: E X): (forall (v: X), sim (k1 v) (k2 v))
                    -> bisimF sim (Vis e k1) (Vis e k2)
```

⊢ Tau spin ≈ spin

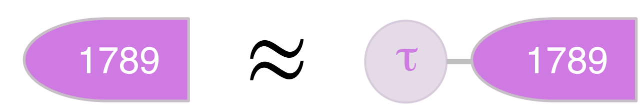
bisim t s ≜ paco bisimF bot

https://github.com/snu-sf/paco



ITree Equivalence

Equivalence Up-To Tau



ITree Equivalence

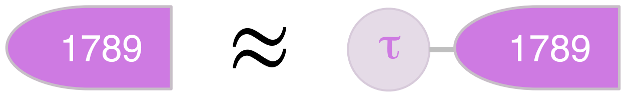
Equivalence Up-To Tau

$t \approx s \triangleq$ eutt t s

```
Inductive euttF (sim: relation (itree E R)): relation itree E R :=

  | EqRet: euttF (Ret v) (Ret v)

  | EqTau: sim t s  -> euttF sim (Tau t) (Tau s)

  | EqVis (e: E X): (forall (v: X), sim (k1 v) (k2 v))
                     -> euttF sim (Vis e k1) (Vis e k2)

  | EqTauL: euttF sim t s -> euttF sim (Tau t) s

  | EqTauR: euttF sim t s -> euttF sim t (Tau s)
```

eutt t s $\triangleq$ paco euttF bot2

https://github.com/snu-sf/paco



ITree Equivalence

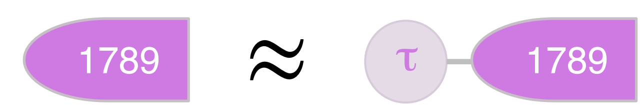
Equivalence Up-To Tau

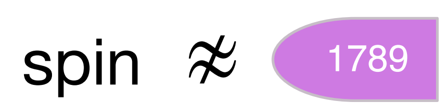
t ≈ s ≜ eutt t s

```
Inductive euttF (sim: relation (itree E R)): relation itree E R :=

  | EqRet: euttF (Ret v) (Ret v)

  | EqTau: sim t s  -> euttF sim (Tau t) (Tau s)

  | EqVis (e: E X): (forall (v: X), sim (k1 v) (k2 v))
                        -> euttF sim (Vis e k1) (Vis e k2)

  | EqTauL: euttF sim t s -> euttF sim (Tau t) s

  | EqTauR: euttF sim t s -> euttF sim t (Tau s)
```

eutt t s ≜ paco euttF bot2

https://github.com/snu-sf/paco





ITrees so Far

A coinductive datastructure representing computations;

Which forms a monad;

Whose notion of equivalence is bisimilarity up-to Tau.

ITrees so Far

A coinductive datastructure representing computations;

Which forms a monad;

Whose notion of equivalence is bisimilarity up-to Tau.

Let’s try using them!

Everyone's Favorite Case Study: Imp

```
Inductive imp : Type :=
  | Skip
  | Assign (x: var) (e: exp)
  | Seq (c1 c2: imp)
  | If (b: exp) (t e: imp)
  | While (b: exp) (c: imp).
```

Our objective:

- Give a denotation to imp
- That is executable
- Suitable to verify a compiler

Everyone's Favorite Case Study: Imp

```
Inductive imp : Type :=
  | Skip
  | Assign (x: var) (e: exp)
  | Seq (c1 c2: imp)
  | If (b: exp) (t e: imp)
  | While (b: exp) (c: imp).
```

Our objective:

- Give a denotation to imp
- That is executable
- Suitable to verify a compiler

Proceeds in two steps

1. Syntax is denoted in terms of itrees;
2. Events contained in the trees are given a semantics into a monad.

Imp Programs as ITrees

Denotation of imp in term of itrees:

```
Fixpoint den_imp (c: imp): itree E_imp unit :=
 match c with
 | Skip         => ret tt
 | Assign x e => v <- den_exp e ;; trigger (EWrite x v)
 | Seq c1 c2  => den_imp c1 ;; den_imp c2
 | If b t e       => v <- den_exp b ;;
                      if is_true v then den_imp t else den_imp e
 | While b c   => ???
 end.
```

Imp Programs as ITrees

Denotation of imp in term of itrees:

```
Fixpoint den_imp (c: imp): itree E_imp unit :=
  match c with
  | Skip         => ret tt
  | Assign x e => v <- den_exp e ;; trigger (EWrite x v)
  | Seq c1 c2  => den_imp c1 ;; den_imp c2
  | If b t e      => v <- den_exp b ;;
                         if is_true v then den_imp t else den_imp e
  | While b c   => ???
  end.
```

Imp Programs as ITrees

Denotation of imp in term of itrees:

```
Fixpoint den_imp (c: imp): itree E_imp unit :=
  match c with
  | Skip        => ret tt
  | Assign x e => v <- den_exp e ;; trigger (EWrite x v)
  | Seq c1 c2  => den_imp c1 ;; den_imp c2
  | If b t e      => v <- den_exp b ;;
                         if is_true v then den_imp t else den_imp e
  | While b c   => ???
  end.
```

Effect interface of Imp:

```
Inductive E_imp : Type -> Type :=
 | ERead (x: var)                : E_imp value
 | EWrite (x: var) (v: value): E_imp unit
```

Imp Programs as ITrees

Denotation of imp in term of itrees:

```
Fixpoint den_imp (c: imp): itree E_imp unit :=
  match c with
  | Skip         => ret tt
  | Assign x e => v <- den_exp e ;; trigger (EWrite x v)
  | Seq c1 c2  => den_imp c1 ;; den_imp c2
  | If b t e       => v <- den_exp b ;;
                        if is_true v then den_imp t else den_imp e
  | While b c   => ???
  end.
```

Effect interface of Imp:

```
Inductive E_imp : Type -> Type :=
  | ERead (x: var)              : E_imp value
  | EWrite (x: var) (v: value): E_imp unit
```

Imp Programs as ITrees

Denotation of imp in term of itrees:

```
Fixpoint den_imp (c: imp): itree E_imp unit :=
 match c with
 | Skip        => ret tt
 | Assign x e => v <- den_exp e ;; trigger (EWrite x v)
 | Seq c1 c2  => den_imp c1 ;; den_imp c2
 | If b t e       => v <- den_exp b ;;
                      if is_true v then den_imp t else den_imp e
 | While b c   => ???
 end.
```

Effect interface of Imp:

```
Inductive E_imp : Type -> Type :=
 | ERead (x: var)                : E_imp value
 | EWrite (x: var) (v: value): E_imp unit
```

Imp Programs as ITrees

Denotation of imp in term of itrees:

```
Fixpoint den_imp (c: imp): itree E_imp unit :=
  match c with
  | Skip        => ret tt
  | Assign x e => v <- den_exp e ;; trigger (EWrite x v)
  | Seq c1 c2  => den_imp c1 ;; den_imp c2
  | If b t e     => v <- den_exp b ;;
                     if is_true v then den_imp t else den_imp e
  | While b c   => ???
  end.
```

Effect interface of Imp:

```
Inductive E_imp : Type -> Type :=
  | ERead (x: var)               : E_imp value
  | EWrite (x: var) (v: value): E_imp unit
```

Imp Programs as ITrees

Denotation of imp in term of itrees:

```
Fixpoint den_imp (c: imp): itree E_imp unit :=
  match c with
  | Skip        => ret tt
  | Assign x e => v <- den_exp e ;; trigger (EWrite x v)
  | Seq c1 c2  => den_imp c1 ;; den_imp c2
  | If b t e      => v <- den_exp b ;;
                       if is_true v then den_imp t else den_imp e
  | While b c   => ???
  end.
```

Effect interface of Imp:

```
Inductive E_imp : Type -> Type :=
 | ERead (x: var)                  : E_imp value
 | EWrite (x: var) (v: value): E_imp unit
```

Imp Programs as ITrees

Denotation of imp in term of itrees:

```
Fixpoint den_imp (c: imp): itree E_imp unit :=
  match c with
  | Skip        => ret tt
  | Assign x e => v <- den_exp e ;; trigger (EWrite x v)
  | Seq c1 c2  => den_imp c1 ;; den_imp c2
  | If b t e      => v <- den_exp b ;;
                        if is_true v then den_imp t else den_imp e
  | While b c   => ???
  end.
```

Effect interface of Imp:

```
Inductive E_imp : Type -> Type :=
  | ERead (x: var)                : E_imp value
  | EWrite (x: var) (v: value): E_imp unit
```

Imp Programs as ITrees

Denotation of imp in term of itrees:

```
Fixpoint den_imp (c: imp): itree E_imp unit :=
  match c with
  | Skip        => ret tt
  | Assign x e => v <- den_exp e ;; trigger (EWrite x v)
  | Seq c1 c2  => den_imp c1 ;; den_imp c2
  | If b t e      => v <- den_exp b ;;
                       if is_true v then den_imp t else den_imp e
  | While b c   => ???
  end.
```

Effect interface of Imp:

```
Inductive E_imp : Type -> Type :=
 | ERead (x: var)                : E_imp value
 | EWrite (x: var) (v: value): E_imp unit
```

Minimal effectful computation:

```
Definition trigger {E X}(e: E X): itree E X :=
  Vis e (fun x => Ret x)
```

Imp Programs as ITrees

Denotation of imp in term of itrees:

```
Fixpoint den_imp (c: imp): itree E_imp unit :=
  match c with
  | Skip        => ret tt
  | Assign x e => v <- den_exp e ;; trigger (EWrite x v)
  | Seq c1 c2  => den_imp c1 ;; den_imp c2
  | If b t e      => v <- den_exp b ;;
                      if is_true v then den_imp t else den_imp e
  | While b c   => ???
  end.
```

Effect interface of Imp:

```
Inductive E_imp : Type -> Type :=
  | ERead (x: var)                  : E_imp value
  | EWrite (x: var) (v: value): E_imp unit
```

Minimal effectful computation:

```
Definition trigger {E X}(e: E X): itree E X :=
  Vis e (fun x => Ret x)
```

Imp Programs as ITrees

Denotation of imp in term of itrees:

```
Fixpoint den_imp (c: imp): itree E_imp unit :=
  match c with
  | Skip         => ret tt
  | Assign x e => v <- den_exp e ;; trigger (EWrite x v)
  | Seq c1 c2  => den_imp c1 ;; den_imp c2
  | If b t e       => v <- den_exp b ;;
                          if is_true v then den_imp t else den_imp e
  | While b c   => ???
  end.
```

Effect interface of Imp:

```
Inductive E_imp : Type -> Type :=
 | ERead (x: var)                 : E_imp value
 | EWrite (x: var) (v: value): E_imp unit
```

Minimal effectful computation:

```
Definition trigger {E X}(e: E X): itree E X :=
  Vis e (fun x => Ret x)
```

Imp Programs as ITrees

Denotation of imp in term of itrees:

```
Fixpoint den_imp (c: imp): itree E_imp unit :=
  match c with
  | Skip         => ret tt
  | Assign x e => v <- den_exp e ;; trigger (EWrite x v)
  | Seq c1 c2  => den_imp c1 ;; den_imp c2
  | If b t e       => v <- den_exp b ;;
                          if is_true v then den_imp t else den_imp e
  | While b c   => ???
  end.
```

Effect interface of Imp:

```
Inductive E_imp : Type -> Type :=
 | ERead (x: var)                  : E_imp value
 | EWrite (x: var) (v: value): E_imp unit
```

Minimal effectful computation:

```
Definition trigger {E X}(e: E X): itree E X :=
  Vis e (fun x => Ret x)
```

An Iteration Combinator

One would like to write:

```
den_imp (while b do c) =?
   v <- den_exp b ;;
   if is_true v
   then den_imp c ;; den_imp (while b do c)
   else ret tt
```

An Iteration Combinator

One would like to write:

```
den_imp (while b do c) =?
   v <- den_exp b ;;
   if is_true v
   then den_imp c ;; den_imp (while b do c)
   else ret tt
```

Continuation trees:

```
Definition ktree E A B := A -> itree E B.
```

Continuation trees have a nice structure:

An Iteration Combinator

One would like to write:

```
den_imp (while b do c) =?
   v <- den_exp b ;;
   if is_true v
   then den_imp c ;; den_imp (while b do c)
   else ret tt
```

Continuation trees:

```
Definition ktree E A B := A -> itree E B.
```

Continuation trees have a nice structure:

- They can be composed; k1 >>> k2

An Iteration Combinator

One would like to write:

```
den_imp (while b do c) =?
   v <- den_exp b ;;
   if is_true v
   then den_imp c ;; den_imp (while b do c)
   else ret tt
```

Continuation trees:

```
Definition ktree E A B := A -> itree E B.
```

Continuation trees have a nice structure:

- They can be composed; k1 >>> k2
- They support case analysis; case k1 k2

An Iteration Combinator

One would like to write:

```
den_imp (while b do c) =?
   v <- den_exp b ;;
   if is_true v
   then den_imp c ;; den_imp (while b do c)
   else ret tt
```

Continuation trees:

```
Definition ktree E A B := A -> itree E B.
```

Continuation trees have a nice structure:

- They can be composed; k1 >>> k2
- They support case analysis; case k1 k2
- They can be iterated over! iter k

An Iteration Combinator

Continuation trees:

```
Definition ktree E A B := A -> itree E B.
```

Iteration combinator:

```
CoFixpoint iter (body: ktree E A (A + B)): ktree E A B :=
  fun a => ab <- body a ;;
           match ab with
           | inl a => Tau (iter body a)
           | inr b => Ret b
           end.
```

An Iteration Combinator

Continuation trees:

```
Definition ktree E A B := A -> itree E B.
```

Iteration combinator:

```
CoFixpoint iter (body: ktree E A (A + B)): ktree E A B :=
  fun a => ab <- body a ;;
           match ab with
           | inl a => Tau (iter body a)
           | inr b => Ret b
           end.
```

Termination

An Iteration Combinator

Continuation trees:

```
Definition ktree E A B := A -> itree E B.
```

Iteration combinator:

```
CoFixpoint iter (body: ktree E A (A + B)): ktree E A B :=
  fun a => ab <- body a ;;
           match ab with
           | inl a => Tau (iter body a)
           | inr b => Ret b
           end.
```

New iteration (guarded)
Termination

An Iteration Combinator

Continuation trees:

```
Definition ktree E A B := A -> itree E B.
```

Iteration combinator:

```
CoFixpoint iter (body: ktree E A (A + B)): ktree E A B :=
  fun a => ab <- body a ;;
           match ab with
           | inl a => Tau (iter body a)
           | inr b => Ret b
           end.
```

New iteration (guarded)
Termination

One would like to write:

```
den_imp (while b do c) =?
   v <- den_exp b ;;
   if is_true v
   then den_imp c ;; den_imp (while b do c)
   else ret tt
```

One can write:

```
den_imp (while b do c) = iter
   (fun _ => v <- den_exp b ;;
             if is_true v
             then den_imp c ;; ret (inl tt)
             else ret (inr tt))
```

Imp Programs as ITrees

Denotation of imp in term of itrees:

```
Fixpoint den_imp (c: imp): itree E_imp unit :=
  match c with
  | Skip         => ret tt
  | Assign x e => v <- den_exp e ;; trigger (GetVar v)
  | Seq c1 c2  => den_imp c1 ;; den_imp c2
  | If b t e       => v <- den_exp b ;;
                       if is_true v then den_imp t else den_imp e
  | While b c   => iter (fun _ => v <- den_exp b ;;
                                    if is_true v
                                    then den_imp c ;; ret (inl tt)
                                    else ret (inr tt))
```

Are we done?

Imp Programs as ITrees

Denotation of imp in term of itrees:

```
Fixpoint den_imp (c: imp): itree E_imp unit :=
  match c with
  | Skip        => ret tt
  | Assign x e => v <- den_exp e ;; trigger (GetVar v)
  | Seq c1 c2  => den_imp c1 ;; den_imp c2
  | If b t e     => v <- den_exp b ;;
                     if is_true v then den_imp t else den_imp e
  | While b c   => iter (fun _ => v <- den_exp b ;;
                                    if is_true v
                                    then den_imp c ;; ret (inl tt)
                                    else ret (inr tt))
```

Are we done?

Let's add some semantic to the mix

Giving Meaning to Events: Handlers

```
Inductive E_imp : Type -> Type :=
 | ERead (x: var)               : E_imp value
 | EWrite (x: var) (v: value): E_imp unit
```

Giving Meaning to Events: Handlers

```
Inductive E_imp : Type -> Type :=
  | ERead (x: var)              : E_imp value
  | EWrite (x: var) (v: value): E_imp unit
```

Events are given *meaning* by handling them into monads:

```
Definition handler (E M: Type -> Type) := E ~> M.
```

Giving Meaning to Events: Handlers

```
Inductive E_imp : Type -> Type :=
  | ERead (x: var)              : E_imp value
  | EWrite (x: var) (v: value): E_imp unit
```

Events are given *meaning* by handling them into monads:

```
Definition handler (E M: Type -> Type) := E ~> M.
```

Notation:
E ~> M ≜ forall X, E X -> M X

Giving Meaning to Events: Handlers

```
Inductive E_imp : Type -> Type :=
  | ERead (x: var)              : E_imp value
  | EWrite (x: var) (v: value): E_imp unit
```

Events are given *meaning* by handling them into monads:

```
Definition handler (E M: Type -> Type) := E ~> M.
```

Notation:
E ~> M ≜ forall X, E X -> M X

Let's handle E_imp into the state monad.

```
Definition h_imp : E_imp ~> stateT (itree voidE) :=
  fun X e s => match e with
              | ERead x     => Ret (s          , s[x])
              | EWrite x v => Ret (s[x <- v], tt    )
              end
```

Lifting Meaning to ITrees: Interpreters

The library provides an interpretation function:

```
interp (h: E ~> M):  itree E R ~> M R
```

Assuming that the monad M supports a notion of iteration:

```
Class MonadIter (M : Type -> Type) : Type :=
  iter : forall {R A: Type}
                   (body: A -> M (A + R)),
                    A -> M R.
```

Lifting Meaning to ITrees: Interpreters

The library provides an interpretation function:

```
interp (h: E ~> M):  itree E R ~> M R
```

Assuming that the monad M supports a notion of iteration:

```
Class MonadIter (M : Type -> Type) : Type :=
  iter : forall {R A: Type}
                  (body: A -> M (A + R)),
                   A -> M R.
```

- itree E
- Prop

supports it

- stateT M
- readerT M
- optionT M
- eitherT M

preserves it

Denotational, Yet Executable

ITrees are coinductive: they can therefore be extracted to an OCaml lazy structure!

Denotational, Yet Executable

ITrees are coinductive: they can therefore be extracted to an OCaml lazy structure!

Simply requires a minimal driver in OCaml:

```
let rec run t =
  match t with
  | Ret r      -> r
  | Tau t      -> run t
  | Vis (e,k) -> handle e (fun x -> run (k x))
```

Denotational, Yet Executable

ITrees are coinductive: they can therefore be extracted to an OCaml lazy structure!

Simply requires a minimal driver in OCaml:

```
let rec run t =
  match t with
  | Ret r      -> r
  | Tau t      -> run t
  | Vis (e,k) -> handle e (fun x -> run (k x))
```

- Nothing to do in the case of our Imp language: all events are interpreted in Coq
- In general, leaves the leisure to write unverified handlers in OCaml

What About Reasoning?

Rich equational reasoning over eutt (excerpt)

What About Reasoning?

Rich equational reasoning over eutt (excerpt)

- Monad Laws: $(x \leftarrow t \;;;\; x) \approx t$

What About Reasoning?

Rich equational reasoning over eutt (excerpt)

- Monad Laws: $(x \leftarrow t \;;;\; x) \approx t$
- Structural Laws: $(\text{Tau } t) \approx t$

What About Reasoning?

Rich equational reasoning over eutt (excerpt)

- Monad Laws: $(x \leftarrow t \;;;\; x) \approx t$
- Structural Laws: $(\text{Tau } t) \approx t$
- Congruence Laws: $(t1 \approx t2 \land k1 \mathrel{\dot{\approx}} k2) \rightarrow (t1 \;;;\; k1) \approx (t2 \;;;\; k2)$

What About Reasoning?

Rich equational reasoning over eutt (excerpt)

- Monad Laws: $(x \leftarrow t \;;;\; x) \approx t$
- Structural Laws: $(\mathrm{Tau}\ t) \approx t$
- Congruence Laws: $(t1 \approx t2 \wedge k1 \doteq\!\!\approx k2) \rightarrow (t1 \;;;\; k1) \approx (t2 \;;;\; k2)$
- Monoidal Laws: $(\mathrm{inl} >>> \mathrm{case}\ h\ k) \doteq\!\!\approx h$

What About Reasoning?

Rich equational reasoning over eutt (excerpt)

- Monad Laws: $(x \leftarrow t \;;;\; x) \approx t$
- Structural Laws: $(\mathrm{Tau}\ t) \approx t$
- Congruence Laws: $(t1 \approx t2 \land k1 \mathrel{\dot{\approx}} k2) \rightarrow (t1 \;;;\; k1) \approx (t2 \;;;\; k2)$
- Monoidal Laws: $(\mathrm{inl} \ggg \mathrm{case}\ h\ k) \mathrel{\dot{\approx}} h$
- Iteration Laws: $(\mathrm{iter}\ f) \mathrel{\dot{\approx}} (f \ggg \mathrm{case}\ (\mathrm{iter}\ f)\ \mathrm{id})$

What About Reasoning?

Rich equational reasoning over eutt (excerpt)

- Monad Laws: $(x \leftarrow t\ ;;\ x) \approx t$
- Structural Laws: $(\mathrm{Tau}\ t) \approx t$
- Congruence Laws: $(t1 \approx t2 \wedge k1 \mathrel{\dot{\approx}} k2) \rightarrow (t1\ ;;\ k1) \approx (t2\ ;;\ k2)$
- Monoidal Laws: $(\mathrm{inl} >>> \mathrm{case}\ h\ k) \mathrel{\dot{\approx}} h$
- Iteration Laws: $(\mathrm{iter}\ f) \mathrel{\dot{\approx}} (f >>> \mathrm{case}\ (\mathrm{iter}\ f)\ \mathrm{id})$
- Interp Laws: $(\mathrm{interp}\ h\ (\mathrm{trigger}\ e)) \approx h\ e$

  $(\mathrm{interp}\ h\ (t\ ;;\ k)) \approx (x \leftarrow \mathrm{interp}\ h\ t\ ;;\ \mathrm{interp}\ h\ (k\ x))$

What About Reasoning?

Rich equational reasoning over eutt (excerpt)

- Monad Laws: $(x \leftarrow t \;;;\; x) \approx t$
- Structural Laws: $(\text{Tau } t) \approx t$
- Congruence Laws: $(t1 \approx t2 \wedge\; k1 \dot{\approx} k2) \rightarrow (t1 \;;;\; k1) \approx (t2 \;;;\; k2)$
- Monoidal Laws: $(\text{inl} >\!>\!> \text{case } h\ k) \dot{\approx} h$
- Iteration Laws: $(\text{iter } f) \;\dot{\approx}\; (f >\!>\!> \text{case } (\text{iter } f)\ \text{id})$
- Interp Laws: $(\text{interp } h\ (\text{trigger } e)) \approx h\ e$

  $(\text{interp } h\ (t \;;;\; k)) \approx (x \leftarrow \text{interp } h\ t \;;;\; \text{interp } h\ (k\ x))$

Support for setoid-based rewriting

~> Most proofs about itrees are purely based on rewriting

A Side Product

In the process of establishing this equational theory, we worked with Gil Hur on an extension of paco

- Richer reasoning principles
  (fixed a deficiency of paco in the presence of nested cofixed-points);
- Fully backward compatible with paco;
- An approach to up-to reasoning principles discriminating between strong and weak guards;
- Come see the talk at CPP in January for more!

An Equational Theory for Weak Bisimulation via Generalized Parameterized Coinduction

Yannick Zakowski
University of Pennsylvania
Philadelphia, PA, USA

Chung-kil Hur
Seoul National University
Seoul, Republic of Korea

Paul He
University of Pennsylvania
Philadelphia, PA, USA

Steve Zdancewic
University of Pennsylvania
Philadelphia, PA, USA

A Verified Compiler you Said?

Case study presented in the paper:

- Similar process over asm, an assembly like language;
- Compiler from imp to asm;
- Proof of correctness:
  expressed as a bisimulation up-to tau, using the eutt relation.

A Verified Compiler you Said?

Case study presented in the paper:

- Similar process over asm, an assembly like language;
- Compiler from imp to asm;
- Proof of correctness:
  expressed as a bisimulation up-to tau, using the eutt relation.

$$
\begin{tikzcd}
p \arrow[r, "den_{Imp}"] \arrow[d] & t^{E}_{Imp} \arrow[r, "interp_{Imp}"] & t^{\varnothing}_{Imp} \arrow[d, "eutt_{\mathcal{R}_{sim}}"] \\
\mathcal{C}(p) \arrow[r, "den_{Asm}"] & t^{F}_{asm} \arrow[r, "interp_{Asm}"] & t^{\varnothing}_{asm}
\end{tikzcd}
$$

A Verified Compiler you Said?

Case study presented in the paper:

- Similar process over asm, an assembly like language;
- Compiler from imp to asm;
- Proof of correctness:
  expressed as a bisimulation up-to tau, using the eutt relation.

Key characteristics of the approach:

- Correctness of the control flow proved independently;
- Termination sensitive, yet inductive proof;
- Almost entirely based on rewriting.

A Verified Compiler you Said?

Case study presented in the paper:

- Similar process over asm, an assembly like language;
- Compiler from imp to asm;
- Proof of correctness:
  expressed as a bisimulation up-to tau, using the eutt relation.

Key characteristics of the approach:

- Correctness of the control flow proved independently;
- Termination sensitive, yet inductive proof;
- Almost entirely based on rewriting.

Documented as a tutorial:
https://github.com/DeepSpec/InteractionTrees/tree/master/tutorial

ITrees Used in Projects

Core Spec

Embeds Haskell programs in Coq to verify them

deep spec server

ITrees instantiated with two different interfaces specify the server and its implementation

Verified Software Toolchain

ITrees are embedded into VST's assertions to specify C programs

Quick Chick

ITree-based specifications are used as a model generating test tracing to check again

A Modular Semantics for LLVM's IR Based on ITrees (Work In Progress)

Vellvm: a Formal Semantics for LLVM

Active participants

Steve Zdancewic

Calvin Beck

Yannick Zakowski

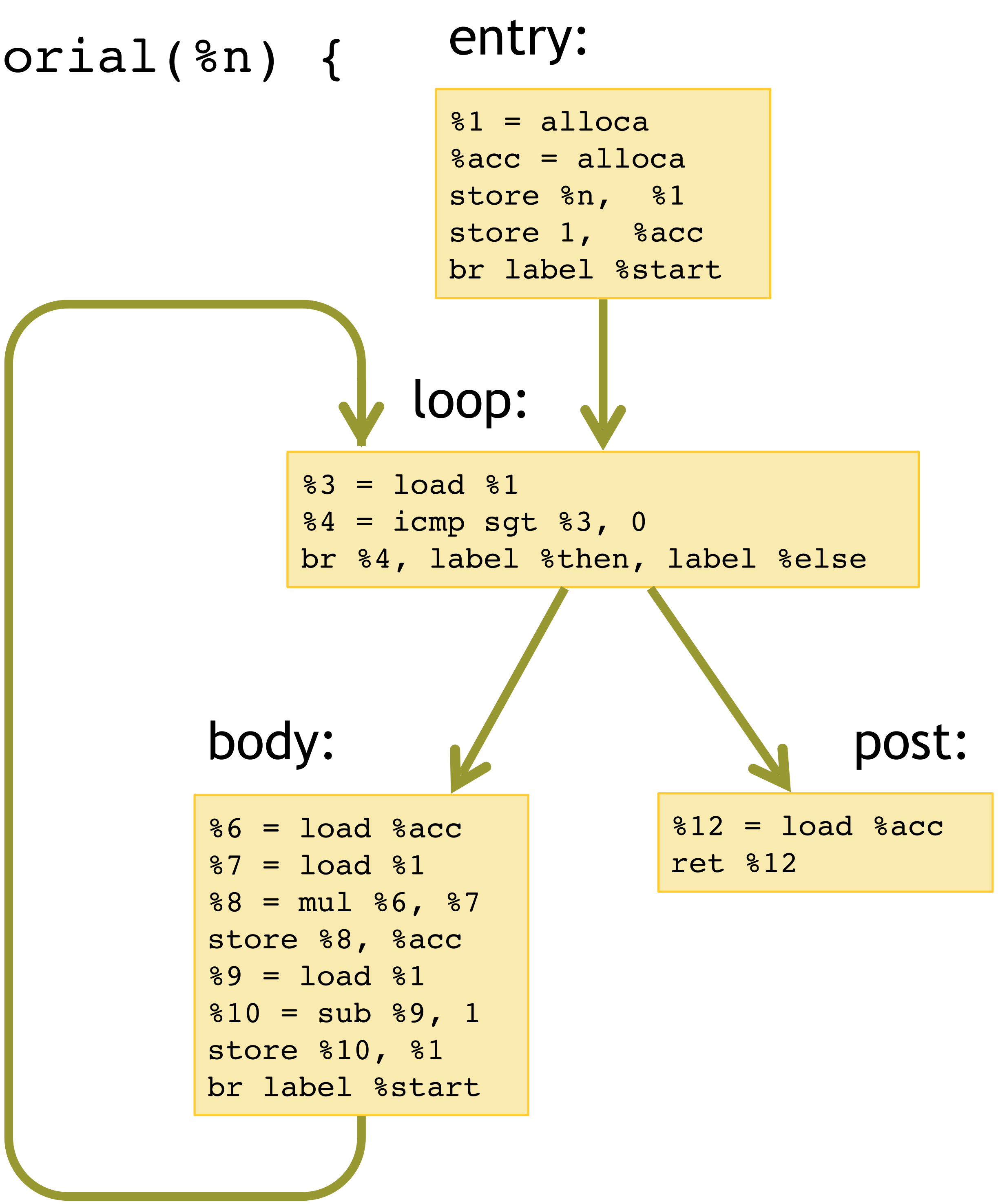
Past participants

- Jianzhou Zhao
- Milo M.K. Martin
- Santosh Nagarakatte
- Dmitri Garbuzov
- William Mansky
- Christine Rizkallah
- Olek Gierczak

- Gil Hur
- Jeehon Kang
- Viktor Vafeiadis

Example LLVM Code

```
define @factorial(%n) {
```




```
}
```

Vellvm: version 1 (2013)

A success inspired by CompCert:

- A large fragment of (sequential) LLVM covered
- A small step operational semantics
- Complex transformations proved correct (mem2reg, …)

With its limitations:

- A monolithic development
- Hard to maintain, difficult to expand
- Complex proofs involved

Can interaction trees help to develop a new semantics that enjoys more modularity?

Well… Let's Start at the Beginning!

```
Definition denote_llvm (p: llvm): itree E_llvm value := …
```

Well… Let's Start at the Beginning!

```
Definition denote_llvm (p: llvm): itree E_llvm value := …
```

What kind of events can an llvm computation trigger?

Well… Let's Start at the Beginning!

```
Definition denote_llvm (p: llvm): itree E_llvm value := …
```

What kind of events can an llvm computation trigger?

Well… Let's Start at the Beginning!

```
Definition denote_llvm (p: llvm): itree E_llvm value := …
```

What kind of events can an llvm computation trigger?

- Global state

Well… Let’s Start at the Beginning!

```
Definition denote_llvm (p: llvm): itree E_llvm value := …
```

What kind of events can an llvm computation trigger?

- Global state
- Local state

Well… Let's Start at the Beginning!

```
Definition denote_llvm (p: llvm): itree E_llvm value := …
```

What kind of events can an llvm computation trigger?

- Global state
- Local state
- Stack of local frames

Well… Let's Start at the Beginning!

Definition denote_llvm (p: llvm): itree E_llvm value := …

What kind of events can an llvm computation trigger?

- Global state
- Local state
- Stack of local frames
- Memory
  - MPush/MPop
  - Load(t,l)/Store(a,v)
  - Alloca(t)
  - GEP(t,v,vs)
  - PtoI(a)/ItoP(i)

Well… Let's Start at the Beginning!

```
Definition denote_llvm (p: llvm): itree E_llvm value := …
```

What kind of events can an llvm computation trigger?

- Global state
- Local state
- Stack of local frames
- Memory
  - MPush/MPop
  - Load(t,l)/Store(a,v)
  - Alloca(t)
  - GEP(t,v,vs)
  - PtoI(a)/ItoP(i)
- Pick

Well… Let's Start at the Beginning!

```
Definition denote_llvm (p: llvm): itree E_llvm value := …
```

What kind of events can an llvm computation trigger?

- Global state
- Local state
- Stack of local frames
- Memory
    - MPush/MPop
    - Load(t,l)/Store(a,v)
    - Alloca(t)
    - GEP(t,v,vs)
    - PtoI(a)/ItoP(i)
- Pick
- Undefined Behavior

Well… Let's Start at the Beginning!

`Definition denote_llvm (p: llvm): itree E_llvm value := …`

What kind of events can an llvm computation trigger?

- Global state
- Local state
- Stack of local frames
- Memory
  - MPush/MPop
  - Load(t,l)/Store(a,v)
  - Alloca(t)
  - GEP(t,v,vs)
  - PtoI(a)/ItoP(i)
- Pick
- Undefined Behavior
- Calls

Well… Let's Start at the Beginning!

```
Definition denote_llvm (p: llvm): itree E_llvm value := …
```

What kind of events can an llvm computation trigger?

- Global state
- Local state
- Stack of local frames
- Memory
    - MPush/MPop
    - Load(t,l)/Store(a,v)
    - Alloca(t)
    - GEP(t,v,vs)
    - PtoI(a)/ItoP(i)
- Pick
- Undefined Behavior
- Calls
- Debugging

Well… Let's Start at the Beginning!

Definition denote_llvm (p: llvm): itree E_llvm value := …

What kind of events can an llvm computation trigger?

- Global state
- Local state
- Stack of local frames
- Memory
  - MPush/MPop
  - Load(t,l)/Store(a,v)
  - Alloca(t)
  - GEP(t,v,vs)
  - PtoI(a)/ItoP(i)
- Pick
- Undefined Behavior
- Calls
- Debugging
- Failure

Well… Let's Start at the Beginning!

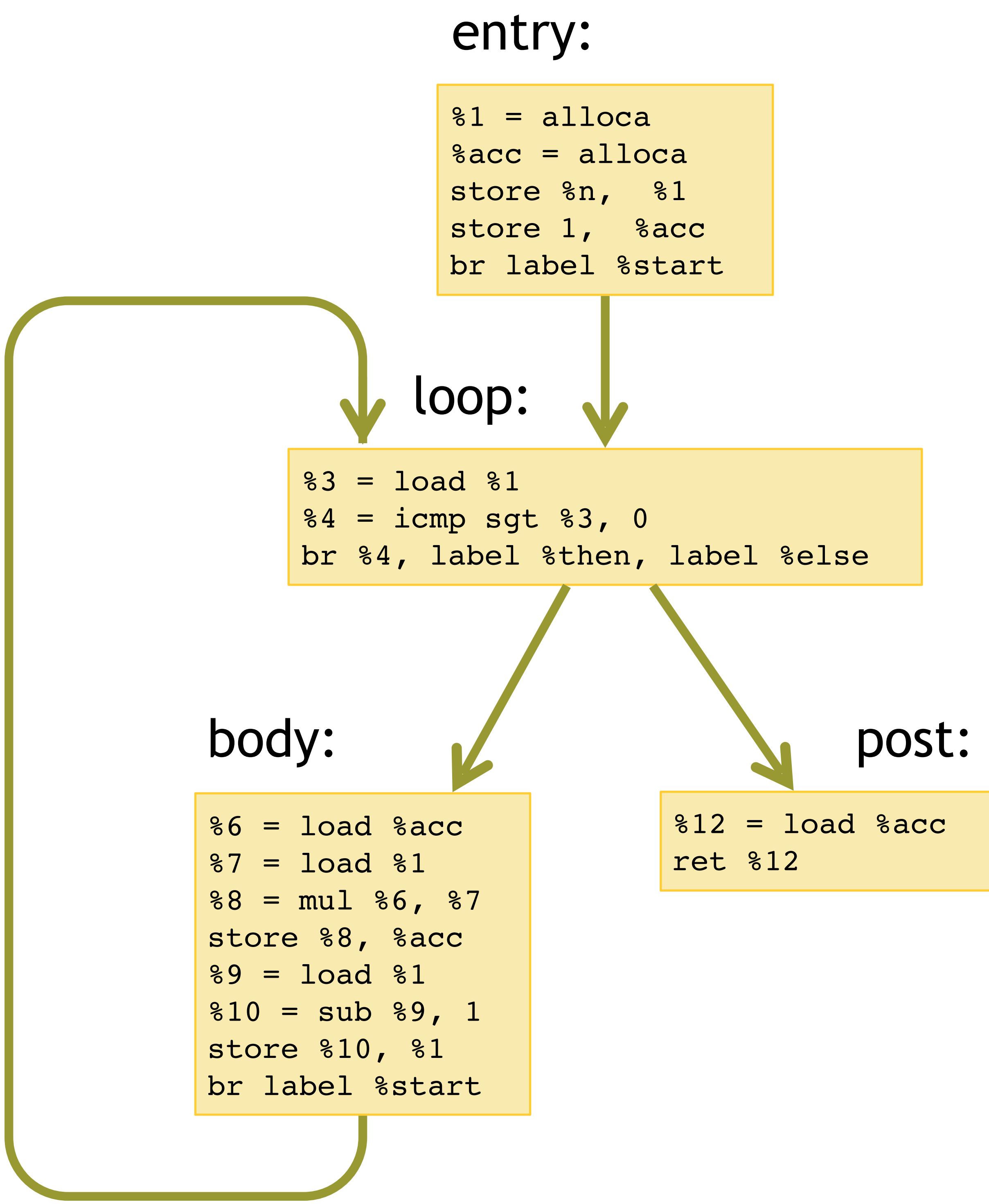
Definition denote_llvm (p: llvm): itree E_llvm value := …

What kind of events can an llvm computation trigger?

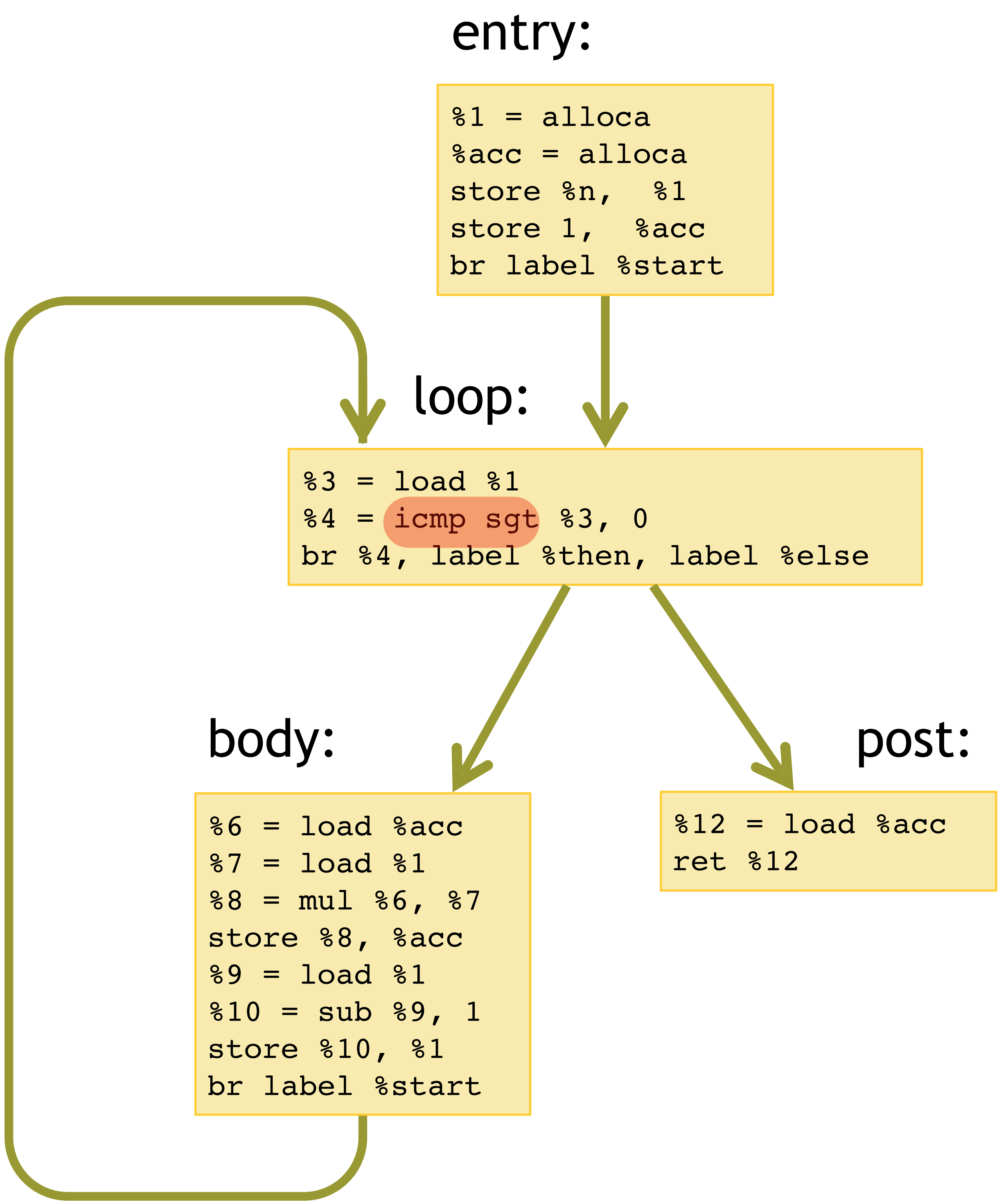
- Global state
- Local state
- Stack of local frames
- Memory
  - MPush/MPop
  - Load(t,l)/Store(a,v)
  - Alloca(t)
  - GEP(t,v,vs)
  - PtoI(a)/ItoP(i)
- Pick
- Undefined Behavior
- Calls
- Debugging
- Failure

Raises challenges to compose interfaces!

To Some Extent: Same Story on Another Scale

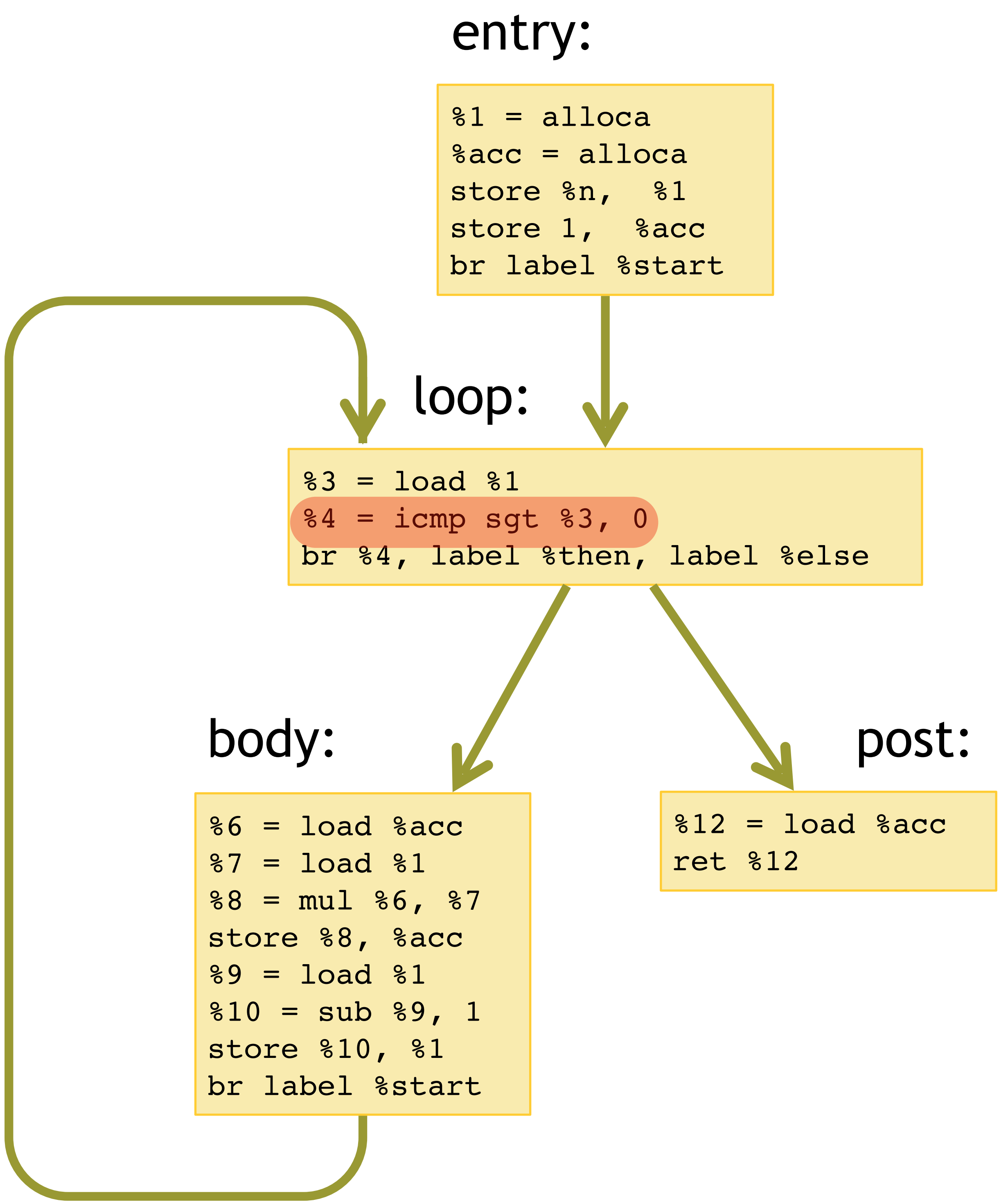


To Some Extent: Same Story on Another Scale



Fixpoint den_exp t e : itree exp_E value

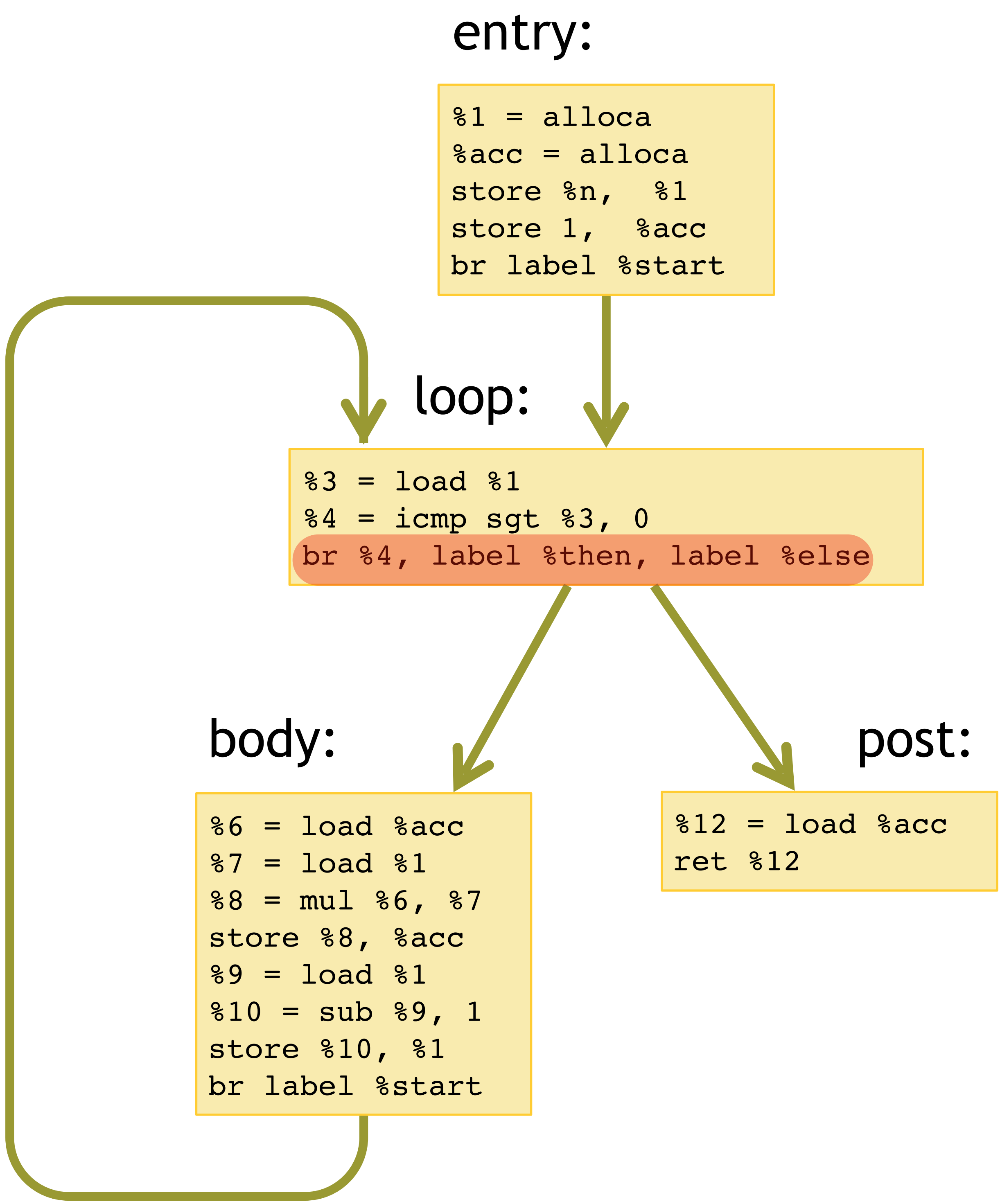
To Some Extent: Same Story on Another Scale



Fixpoint den_exp t e : itree exp_E value

Definition den_instr i : itree instr_E unit

To Some Extent: Same Story on Another Scale

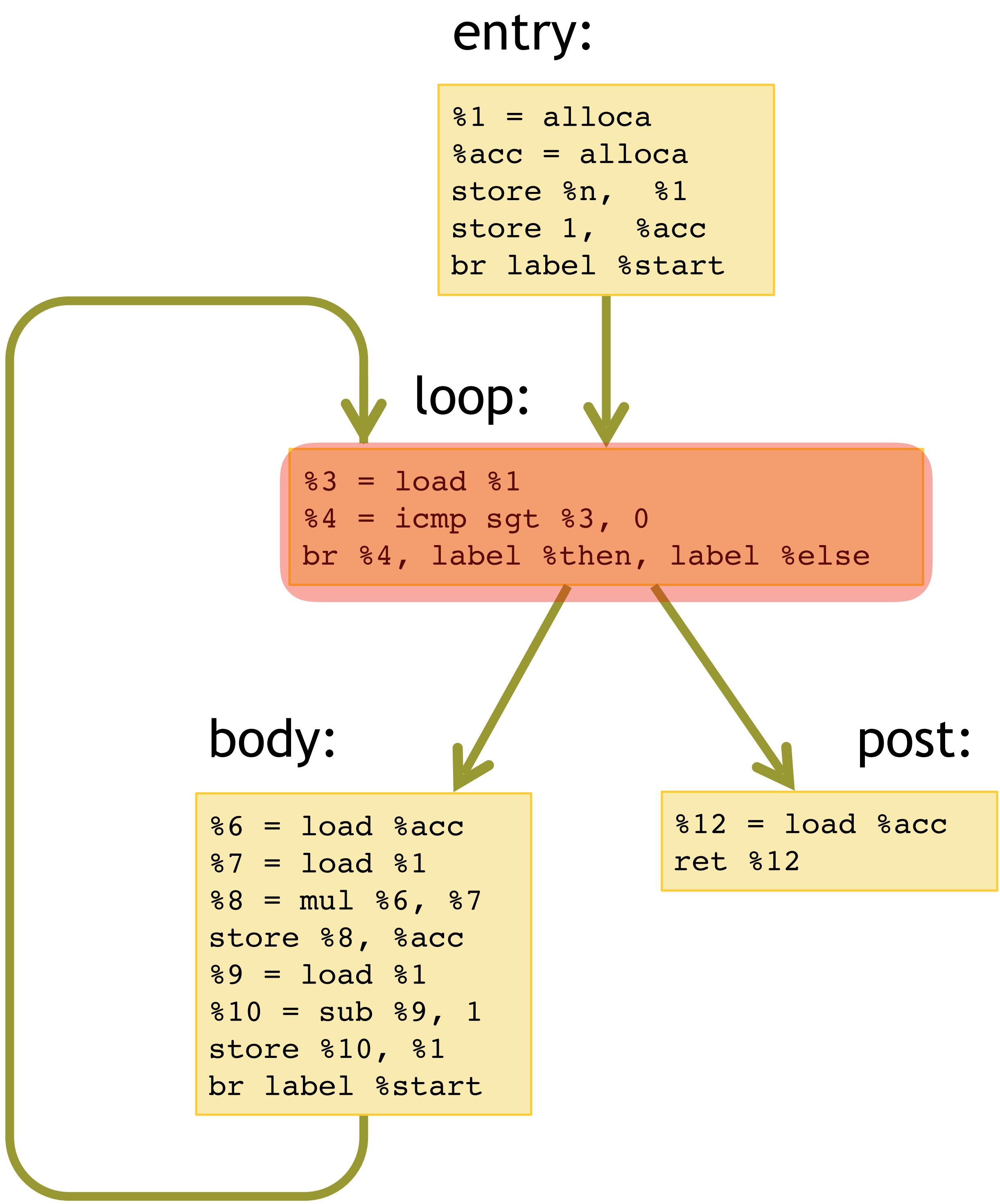


Fixpoint den_exp t e : itree exp_E value

Definition den_instr i : itree instr_E unit

Definition den_terminator t : itree exp_E (bid + value)

To Some Extent: Same Story on Another Scale



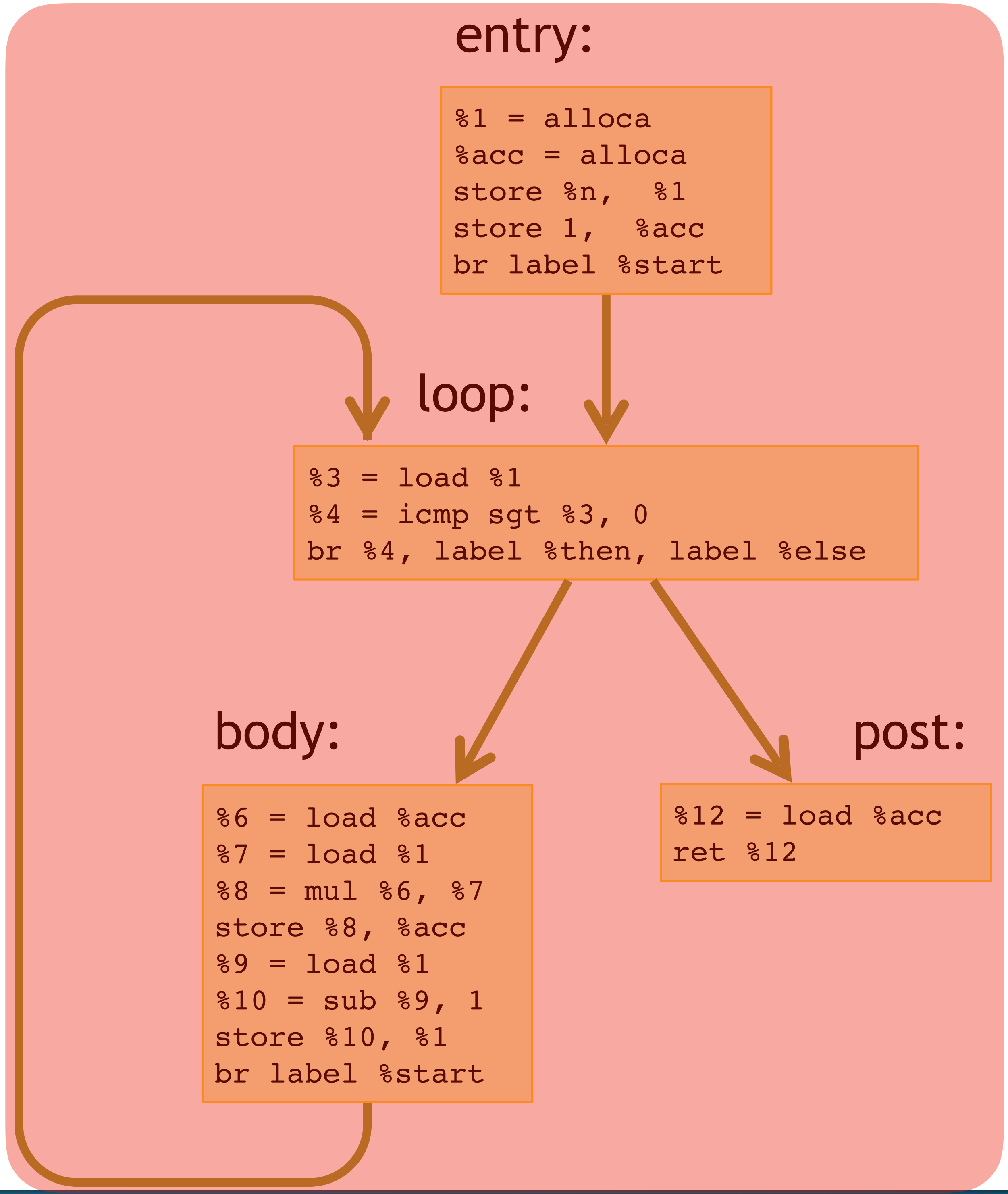
Fixpoint den_exp t e : itree exp_E value

Definition den_instr i : itree instr_E unit

Definition den_terminator t : itree exp_E (bid + value)

Definition den_block b : itree instr_E (bid + value)

To Some Extent: Same Story on Another Scale



Fixpoint den_exp t e : itree exp_E value

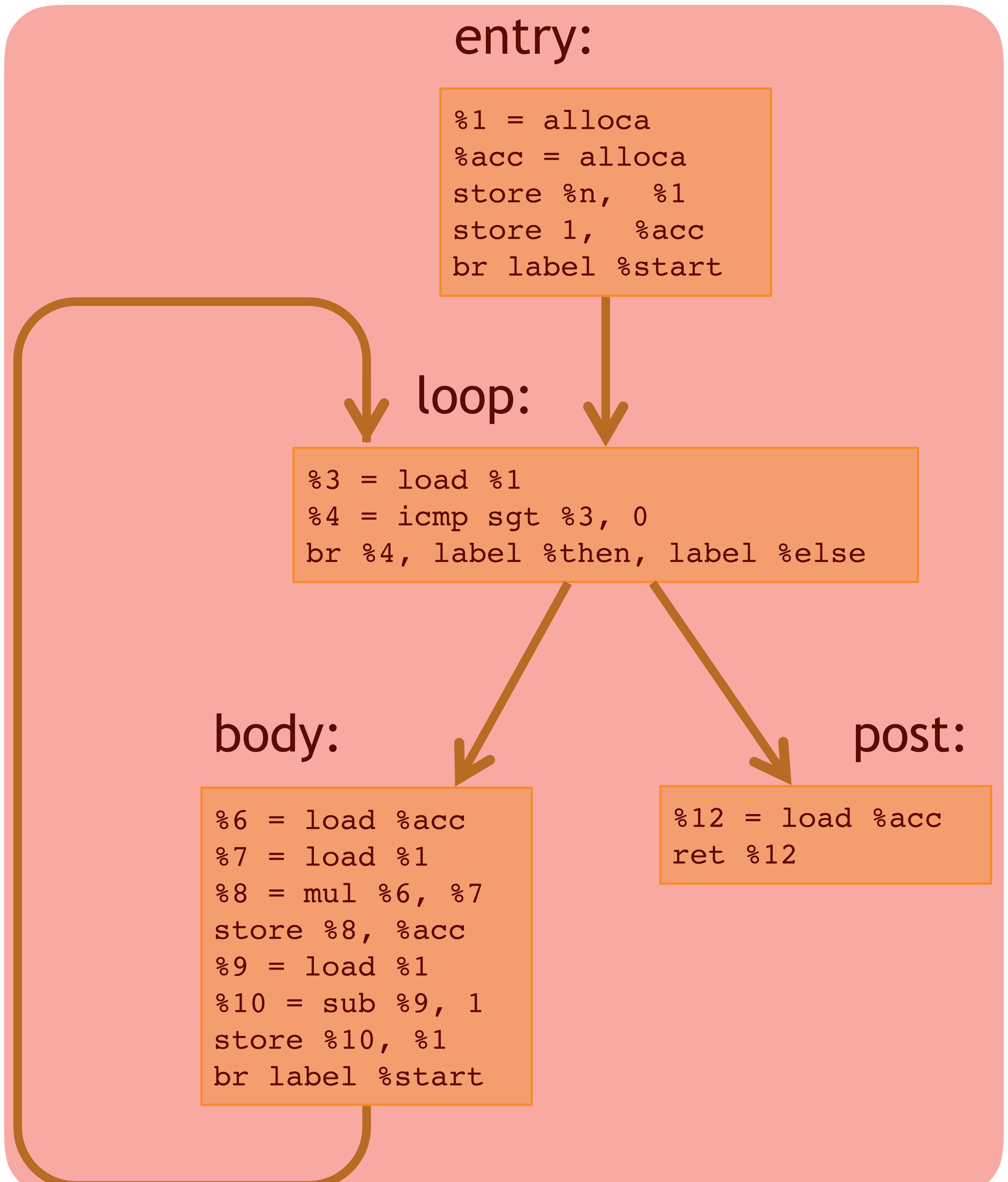
Definition den_instr i : itree instr_E unit

Definition den_terminator t : itree exp_E (bid + value)

Definition den_block b : itree instr_E (bid + value)

Definition den_cfg f : itree instr_E value

To Some Extent: Same Story on Another Scale



Fixpoint den_exp t e : itree exp_E value

Definition den_instr i : itree instr_E unit

Definition den_terminator t : itree exp_E (bid + value)

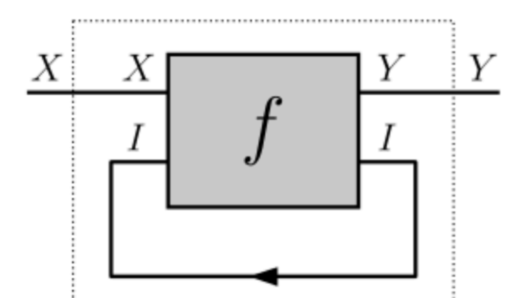
Definition den_block b : itree instr_E (bid + value)

Definition den_cfg f : itree instr_E value

It's a fixed-point!

den_block: ktree instr_E bid (bid + value)

den_block := iter …



A Chain of Interpreters

denote_llvm p

↓

itree E_llvm value

A Chain of Interpreters

denote_llvm p

$\downarrow$

itree E_llvm value

itree E0 value

interp_intrinsics

A Chain of Interpreters

denote_llvm p

itree E_llvm value

itree E0 value

itree E1 (gstate * value)

interp_intrinsics

interp_globals

A Chain of Interpreters

denote_llvm p

itree E_llvm value

itree E0 value

itree E1 (gstate * value)

itree E2 (lstack * (gstate * value))

interp_intrinsics

interp_globals

interp_locals

A Chain of Interpreters

denote_llvm p

↓

itree E_llvm value

interp_intrinsics

itree E0 value

interp_globals

itree E1 (gstate * value)

interp_locals

itree E2 (lstack * (gstate * value))

interp_memory

itree E3 (memory * (lstack * (gstate * value)))

A Chain of Interpreters

denote_llvm p

itree E_llvm value

interp_intrinsics

itree E0 value

interp_globals

itree E1 (gstate * value)

interp_locals

itree E2 (lstack * (gstate * value))

interp_memory

itree E3 (memory * (lstack * (gstate * value)))

model_undef

itree E4 (memory * (lstack * (gstate * value))) -> Prop

A Chain of Interpreters

denote_llvm p

itree E_llvm value

interp_intrinsics

itree E0 value

interp_globals

itree E1 (gstate * value)

interp_locals

itree E2 (lstack * (gstate * value))

interp_memory

itree E3 (memory * (lstack * (gstate * value)))

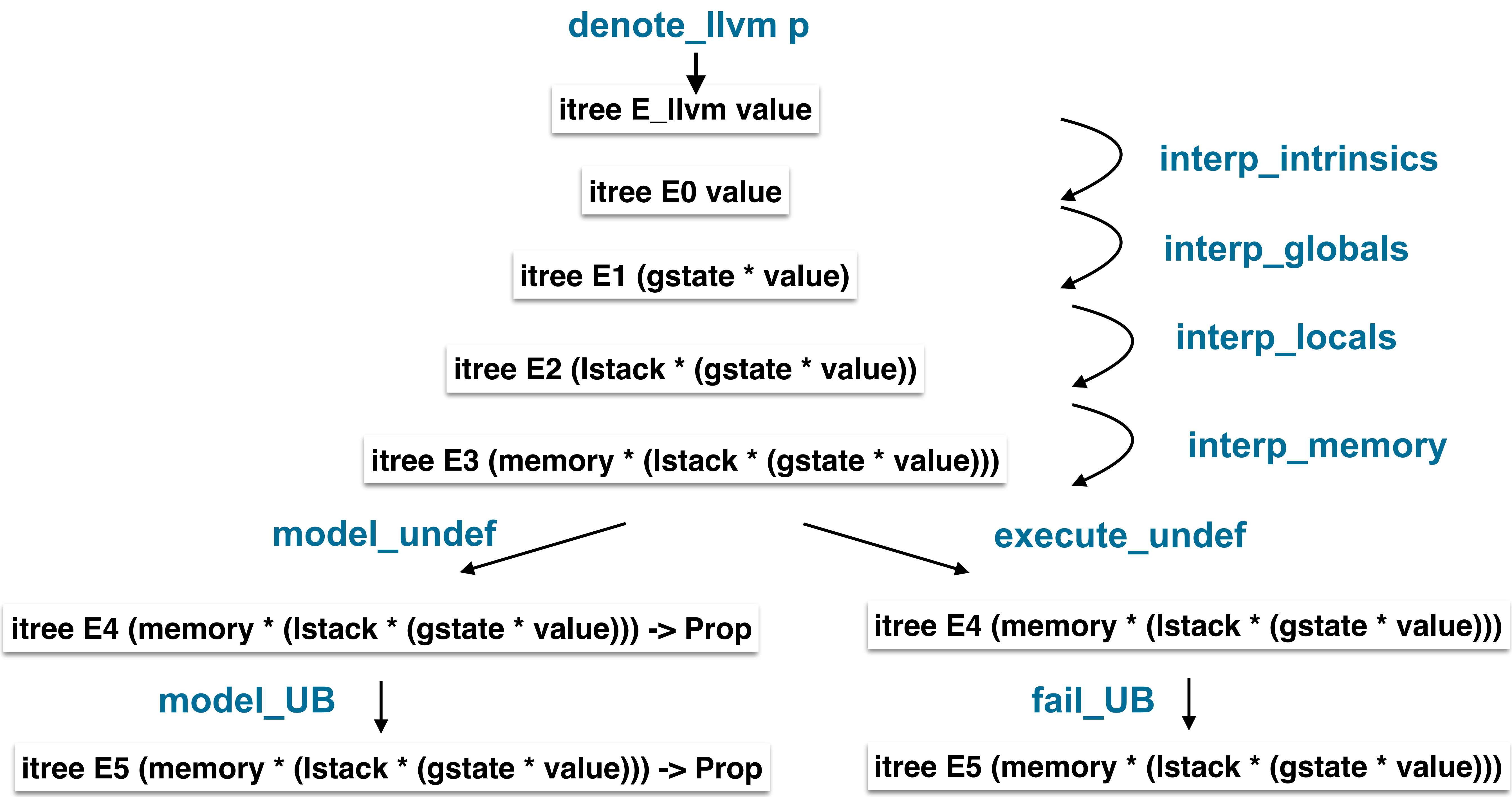
model_undef

itree E4 (memory * (lstack * (gstate * value))) -> Prop

model_UB

itree E5 (memory * (lstack * (gstate * value))) -> Prop

A Chain of Interpreters



State of the Project

The full story has more to say, including about:

- Treatment of poison and undef;
- Mutually recursive definition of functions;
- Memory model;
- Hierarchy of refinements.

State of the Project

The full story has more to say, including about:

- Treatment of poison and undef;
- Mutually recursive definition of functions;
- Memory model;
- Hierarchy of refinements.

Currently done: the new semantics is fully defined.

The proof of the meta-theory and its use to prove optimizations is in progress.

State of the Project

The full story has more to say, including about:

- Treatment of poison and undef;
- Mutually recursive definition of functions;
- Memory model;
- Hierarchy of refinements.

Currently done: the new semantics is fully defined.

The proof of the meta-theory and its use to prove optimizations is in progress.

This is still a work in progress, but it can be followed on Github:
https://github.com/vellvm/vellvm

State of the Project

The full story has more to say, including about:

- Treatment of poison and undef;
- Mutually recursive definition of functions;
- Memory model;
- Hierarchy of refinements.

Currently done: the new semantics is fully defined.

The proof of the meta-theory and its use to prove optimizations is in progress.

This is still a work in progress, but it can be followed on Github:
https://github.com/vellvm/vellvm

Already a user: Vadim Zaliva compiles Helix to Vellvm!

Conclusion

Interaction trees (POPL'20) offer a library for:

- A data-structure to represent recursive, effectful computations;
- Expressive combinators to build and compose them;
- A family of interpreters of itrees into monads;
- A rich equational theory to reason up-to taus about them;
- Tutorial to prove a compiler correct using itrees.

Interaction trees (POPL'20) offer a library for:

- A data-structure to represent recursive, effectful computations;
- Expressive combinators to build and compose them;
- A family of interpreters of itrees into monads;
- A rich equational theory to reason up-to taus about them;
- Tutorial to prove a compiler correct using itrees.

Generalized Parameterized Coinduction (CPP'20):

- Extends the paco library in a backward-compatible way;
- Demonstrates how to axiomatize reasoning up-to tau in a way sensitive to strong/weak guards.

Interaction trees (POPL'20) offer a library for:

- A data-structure to represent recursive, effectful computations;
- Expressive combinators to build and compose them;
- A family of interpreters of itrees into monads;
- A rich equational theory to reason up-to taus about them;
- Tutorial to prove a compiler correct using itrees.

Generalized Parameterized Coinduction (CPP'20):

- Extends the paco library in a backward-compatible way;
- Demonstrates how to axiomatize reasoning up-to tau in a way sensitive to strong/weak guards.

A modular Vellvm using ITrees (in progress):

- A new completely denotational semantics;
- A chain of interpreters allowing for both a model and an executable;
- Notions of refinements inheriting from itree's equational theory.

Interaction trees (POPL'20) offer a library for:

- A data-structure to represent recursive, effectful computations;
- Expressive combinators to build and compose them;
- A family of interpreters of itrees into monads;
- A rich equational theory to reason up-to taus about them;
- Tutorial to prove a compiler correct using itrees.

Generalized Parameterized Coinduction (CPP'20):

- Extends the paco library in a backward-compatible way;
- Demonstrates how to axiomatize reasoning up-to tau in a way sensitive to strong/weak guards.

A modular Vellvm using ITrees (in progress):

- A new completely denotational semantics;
- A chain of interpreters allowing for both a model and an executable;
- Notions of refinements inheriting from itree's equational theory.

Two early prospects:

- Denoting CCS as ITrees;
- Dijkstra's monad for ITrees.