MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Modular, Compositional, and Executable Formal Semantics for LLVM IR

Authors: Zakowski, Beck, Yoon, Zaichuk, Zaliva, Zdancewic
Presentor: Eric Bond (group 27)

Outline

- Motivation & Background
- Introduction to Interaction Trees (ITrees)
- Modeling a simple assembly language (ASM) using ITrees
- Extending ASM to LLVM IR
- Authors' results
- "Group" commentary

Formal Models

Specifications

- Informal:
  - Express correctness, fuzzing
  - Easy to create
  - Only partial correctness
  - Errors can easily “sneak by”
- Formal:
  - Mathematical model
  - Model can prove total correctness
  - Challenge: coherence of model and implementation

VELLVM

- Model of LLVM IR
- Used to prove correctness of LLVM transformations, optimizations, and compilation

Denotational Semantics

- A method of mapping a language into a mathematical object and designing an equational theory to prove properties of the language.

Modular, Compositional, and Executable Formal Semantics for LLVM IR

Proof Engineering

Math

~~Coq~~ Gallina

“Specification” by Bing Image Creator

Interaction Trees

```
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).
```

- Tree with 3 types of nodes
  - Ret: a leaf holding a value of type R
  - Tau: an empty node that has one successor
  - Vis: a node with an **Effect** and a **Continuation**

Interaction Trees

```
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).
```

- Tree with 3 types of nodes
  - Ret: a leaf holding a value of type R
  - Tau: an empty node that has one successor
  - Vis: a node with an **Effect** and a **Continuation**
- An ITree is parameterized by two types.
  - E: The type of effects this tree supports
  - R: The “return type” of the computation
- An ITree is a CoInductive type
  - Analogy: lists are Inductive, streams are CoInductive

Example Interaction Trees

```
CoFixpoint boring : itree IO nat
  := Ret 42.
```

Ex 1) A program that just returns 42

```
CoFixpoint spin : itree IO nat
  := Tau spin.
```

Ex 2) A program that spins forever

```
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).
```

ITree definition

```
Inductive IO : Type -> Type :=
| Input : IO string
| Output : string -> IO unit.
```

An input/output effect

Example Interaction Trees

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

Ex 3) A program that
takes input and prints it (forever)

```
CoFixpoint kill9 : itree IO string
  := Vis (Input)
    (fun str =>
      if (str =? "9")
      then (Ret "done")
      else kill9).
```

Ex 4) A program that
terminates upon receiving input “9”

```
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).
```

ITree definition

```
Inductive IO : Type -> Type :=
| Input : IO string
| Output : string -> IO unit.
```

An input/output effect

Example Interaction Trees

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

A program that takes input and prints it (forever)

```
CoFixpoint echo : itree IO void :=
    n <- trigger Input ;
    trigger (Output n) ;
    Tau echo.
```

The same program, using monad syntax

Equational Reasoning with ITrees

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

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
```

Fig. 5. Core equational theory of ITrees.

```
Theorem compile_correct (s : stmt) : ⟦s⟧ ≈ ⟦(compile s)⟧.
```

Bisimulation is a way to define when two systems “behave the same” relative to an external observer and independent of their internal structure.

Outline

- Motivation & Background
- Introduction to Interaction Trees (ITrees)
- Modeling a simple assembly language (ASM) using ITrees
- Extending ASM to LLVM IR
- Authors' results
- "Group" commentary

Simple Assembly Language → ITree

1. Define the syntax of ASM
2. Decide what effects ASM has
3. Map the syntax of ASM into an Itree

Step 1: Define ASM

```
Definition addr : Set := string.
Definition reg : Set := nat.
Definition value : Set := nat.

Variant operand : Set :=
| Oimm (_ : value)
| Oreg (_ : reg).

Variant instr : Set :=
| Imov   (dest : reg) (src : operand)
| Iload  (dest : reg) (addr : addr)
| Istore (addr : addr) (val : operand)
| Iadd   (dest : reg) (src : reg) (o : operand)
...
```

```
Variant branch {label : Type} : Type :=
| Bjmp (_ : label)
| Bbrz (_ : reg) (yes no : label)
| Bhalt.

Inductive block {label : Type} : Type :=
| bbi (_ : instr) (_ : block)
| bbb (_ : branch label).

Record asm (A B: nat) : Type :=
{
    internal : nat;
    code     : fin (internal + A) -> block (fin (internal + B))
}.
```

ASM syntax

Step 2: Determine Effects

```
Variant Reg : Type -> Type :=
| GetReg (x : reg) : Reg value
| SetReg (x : reg) (v : value) : Reg unit.

Inductive Memory : Type -> Type :=
| Load  (a : addr) : Memory value
| Store (a : addr) (val : value) : Memory unit.

Definition RegAndMem : Type -> Type := Memory ⊕ Reg.
```

```
Definition addr : Set := string.
Definition reg : Set := nat.
Definition value : Set := nat.

Variant operand : Set :=
| Oimm (_ : value)
| Oreg (_ : reg).

Variant instr : Set :=
| Imov   (dest : reg) (src : operand)
| Iload  (dest : reg) (addr : addr)
| Istore (addr : addr) (val : operand)
| Iadd   (dest : reg) (src : reg) (o : operand)
...

Variant branch {label : Type} : Type :=
| Bjmp (_ : label)
| Bbrz (_ : reg) (yes no : label)
| Bhalt.

Inductive block {label : Type} : Type :=
| bbi (_ : instr) (_ : block)
| bbb (_ : branch label).

Record asm (A B: nat) : Type :=
{
   internal : nat;
   code     : fin (internal + A) -> block (fin (internal + B))
}.
```

ASM syntax

Step 3: ASM → ITree RegAndMem void

```
Definition denote_operand (o : operand) : itree RegAndMem value :=
  match o with
  | Oimm v => Ret v
  | Oreg v => trigger (GetReg v)
  end.

Definition denote_instr (i : instr) : itree RegAndMem unit :=
  match i with
  | Iload d addr =>
    val <- trigger (Load addr) ;;
    trigger (SetReg d val)
  | Istore addr v =>
    val <- denote_operand v ;;
    trigger (Store addr val)
  | Imov d s =>
    v <- denote_operand s ;;
    trigger (SetReg d v)
  | Iadd d l r =>
    lv <- trigger (GetReg l) ;;
    rv <- denote_operand r ;;
    trigger (SetReg d (lv + rv))
  ...
```

```
Definition denote_br {B} (b : branch B) : itree RegAndMem B :=
  match b with
  | Bjmp l => Ret l
  | Bbrz v y n =>
    val <- trigger (GetReg v) ;;
    if val:nat then Ret y else Ret n
  | Bhalt => exit
  end.

Fixpoint denote_bk {B} (b : block B) : itree RegAndMem B :=
  match b with
  | bbi i b =>
    denote_instr i ;; denote_bk b
  | bbb b =>
    denote_br b
  end.
```

Outline

- Motivation & Background
- Introduction to Interaction Trees (ITrees)
- Modeling a simple assembly language (ASM) using ITrees
- Extending ASM to LLVM IR
- Authors' results
- "Group" commentary

LLVM → ITree

1. Define the syntax of LLVM
2. Decide what effects LLVM has
3. Map the syntax of LLVM into an Itree

Step 1: Define LLVM Syntax → ITree

- Accounts for full LLVM IR
  - Straightforward, but tedious
  - Including phi nodes, metadata, data layout, attributes, module flags,..
- Authors' provide a parser from ll files into this syntax

Step 2: Determine Effects

```
Variant GlobalE (k v:Type) : Type -> Type :=
| GlobalWrite (id: k) (dv: v): GlobalE k v unit
| GlobalRead  (id: k): GlobalE k v v.

Variant LocalE (k v:Type) : Type -> Type :=
| LocalWrite (id: k) (dv: v): LocalE k v unit
| LocalRead  (id: k): LocalE k v v.

Variant StackE (k v:Type) : Type -> Type :=
| StackPush (args: list (k * v)) : StackE k v unit
| StackPop : StackE k v unit.

Variant CallE : Type -> Type :=
| Call        : forall (t:dtyp) (f:uvalue) (args:list uvalue), CallE uvalue.

Variant ExternalCallE : Type -> Type :=
| ExternalCall        : forall (t:dtyp) (f:uvalue) (args:list dvalue), ExternalCallE dvalue.

Variant IntrinsicE : Type -> Type :=
| Intrinsic : forall (t:dtyp) (f:string) (args:list dvalue), IntrinsicE dvalue.
```

Step 2: Determine Effects

```
Variant GlobalE (k v:Type) : Type -> Type :=
| GlobalWrite (id: k) (dv: v): GlobalE k v unit
| GlobalRead  (id: k): GlobalE k v v.

Variant LocalE (k v:Type) : Type -> Type :=
| LocalWrite (id: k) (dv: v): LocalE k v unit
| LocalRead  (id: k): LocalE k v v.

Variant StackE (k v:Type) : Type -> Type :=
| StackPush (args: list (k * v)) : StackE k v unit
| StackPop : StackE k v unit.

Variant CallE : Type -> Type :=
| Call        : forall (t:dtyp) (f:uvalue) (args:list uvalue), CallE uvalue.

Variant ExternalCallE : Type -> Type :=
| ExternalCall        : forall (t:dtyp) (f:uvalue) (args:list dvalue), ExternalCallE dvalue.

Variant IntrinsicE : Type -> Type :=
| Intrinsic : forall (t:dtyp) (f:string) (args:list dvalue), IntrinsicE dvalue.
```

```
Variant MemoryE : Type -> Type :=
| MemPush : MemoryE unit
| MemPop  : MemoryE unit
| Alloca  : forall (t:dtyp),                                  (MemoryE dvalue)
| Load    : forall (t:dtyp)   (a:dvalue),                     (MemoryE uvalue)
| Store   : forall (a:dvalue) (v:dvalue),                     (MemoryE unit)
| GEP     : forall (t:dtyp)   (v:dvalue) (vs:list dvalue),    (MemoryE dvalue)
| ItoP    : forall (i:dvalue),                                (MemoryE dvalue)
| PtoI    : forall (t:dtyp) (a:dvalue),                       (MemoryE dvalue)
.

Variant PickE : Type -> Type :=
| pick (u:uvalue) (P : Prop) : PickE dvalue.

Variant UBE : Type -> Type :=
| ThrowUB : string -> UBE void.

Variant exceptE (Err : Type) : Type -> Type :=
| Throw : Err -> exceptE Err void.

Variant DebugE : Type -> Type :=
| Debug : string -> DebugE unit.
```

Step 3: LLVM → ITree VellvmE V

```
| OP_GetElementPtr dt1 (dt2, ptrval) idxs =>
  vptr <- denote_exp (Some dt2) ptrval ;;
  vs <- map_monad (fun '(dt, index) => denote_exp (Some dt) index) idxs ;;

  let maybe_dvs := dvptr <- uvalue_to_dvalue vptr ;;
                   dvs <- map_monad uvalue_to_dvalue vs ;;
                   ret (dvptr, dvs)
  in

  match maybe_dvs with
  | inr (dvptr, dvs) => fmap dvalue_to_uvalue (trigger (GEP dt1 dvptr dvs))
  | inl _ =>
    (* Pick to get dvalues *)
    dvptr <- concretize_or_pick vptr True ;;
    dvs <- map_monad (fun v => concretize_or_pick v True) vs ;;
    fmap dvalue_to_uvalue (trigger (GEP dt1 dvptr dvs))
  end
```

Mapping GetElementPrt instruction to ITree program

Step4: ITree E R → Monad Transformer Stack

```
| Load t dv =>
  match dv with
  | DVALUE_Addr ptr =>
    match read m ptr t with
    | inr v => ret (m, v)
    | inl s => raiseUB s
    end
  | _ => raise "Attempting to load from a non-address dvalue"
  end
```

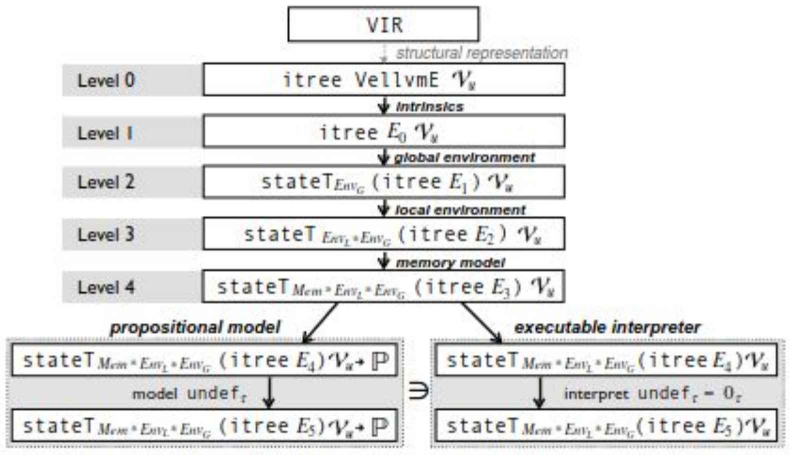


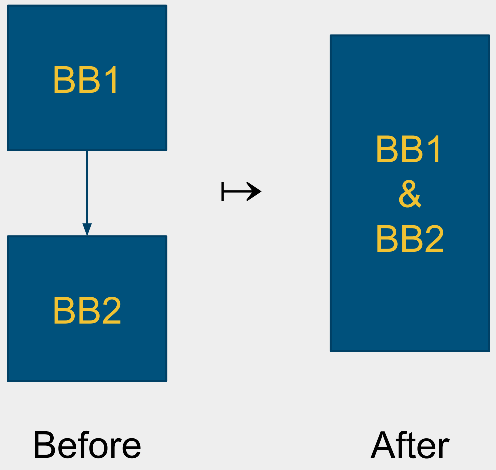
Fig. 5. Levels of interpretation

Outline

- Motivation & Background
- Introduction to Interaction Trees (ITrees)
- Modeling a simple assembly language (ASM) using ITrees
- Extending ASM to LLVM IR
- Authors' results
- "Group" commentary

Authors' Results

- Block fusion
  - Conditions:
    - BB1 has a direct jump to BB2
    - BB1 is the only predecessor of BB2
    - BB1 ≠ BB2
  - Transformation
    - Remove BB1 branch
    - Merge BB1 and BB2
    - Update Phi nodes of BB2's successors



Authors' Results

```
Theorem block_fusion_cfg_correct :
  forall (G : cfg dtyp),
    wf_cfg G ->
    ⟦ G ⟧ ≈ ⟦ block_fusion_cfg G ⟧.
Proof.
  intros G [WF1 WF2].
  unfold denote_cfg.
  simpl bind.
  unfold block_fusion_cfg.
  destruct (block_fusion G.(blks)) as [bks' [[src tgt] |]] eqn:EQ.
  - break_match_goal; [reflexivity |].
    simpl.
    apply Bool.orb_false_elim in Heqb as [INEQ1 INEQ2].
    unfold Eqv.eqv_dec in *.
    rewrite <- RelDec.neg_rel_dec_correct in INEQ1.
    rewrite <- RelDec.neg_rel_dec_correct in INEQ2.
    eapply block_fusion_correct_some
      with (f := G.(init)) (to := G.(init)) in EQ; auto.
    rewrite update_provenance_ineq in EQ; auto.
    eapply eutt_clo_bind; [apply EQ |].
    intros [[]|?] [[]|?] INV; try now inv INV.
    subst; reflexivity.
    eapply wf_cfg_src_not_in_phis; eauto.
    constructor; auto.
  - reflexivity.
Qed.
```

$$[\![G]\!] \approx [\![\text{fuse}(G)]\!]$$

Authors' Results

- This paper and the original ITrees paper have been used in recent developments
  - VELLVM is used in the HELIX verification chain
  - HELIX is code generation and formal verification system with a focus on the intersection of high-performance and high-assurance numerical computing
- Distinguished paper POPL 2020

deep spec

Outline

- Motivation & Background
- Introduction to Interaction Trees (ITrees)
- Modeling a simple assembly language (ASM) using ITrees
- Extending ASM to LLVM IR
- Authors' results
- "Group" commentary

“Group” Commentary

- Strengths
  - Elegant theory
  - Excellent proof engineering
    - Modular, reusable components
    - Abstracted over hard coinductive proofs
    - Provide great tactic library
- Weaknesses
  - Coherence
    - Memory model is not sufficient to prove certain optimizations (in progress)
    - Only sequential programs supported for now
  - While they provide a good equational theory and proof tactics, a better program logic will be needed to handle large programs. (In progress)

Questions?

Monad Laws

$$(x \leftarrow \texttt{ret}\ v\ ;;\ k\ x) \cong (k\ v)$$
$$(x \leftarrow t\ ;;\ \texttt{ret}\ x) \cong t$$
$$(x \leftarrow (y \leftarrow s\ ;;\ t)\ ;;\ u) \cong (y \leftarrow s\ ;;\ x \leftarrow t\ ;;\ u)$$

Structural Laws

$$(\texttt{Tau}\ t) \approx t$$
$$(x \leftarrow (\texttt{Tau}\ t)\ ;;\ k) \approx \texttt{Tau}\ (x \leftarrow t\ ;;\ k)$$
$$(x \leftarrow (\texttt{Vis}\ e\ k1)\ ;;\ k2) \approx (\texttt{Vis}\ e\ (\textbf{fun}\ y \Rightarrow (k1\ y)\ ;;\ k2))$$

Congruences

$$t1 \cong t2 \rightarrow \texttt{Tau}\ t1 \cong \texttt{Tau}\ t2$$
$$k1 \stackrel{\sim}{\cong} k2 \rightarrow \texttt{Vis}\ e\ k1 \approx \texttt{Vis}\ e\ k2$$
$$t1 \approx t2 \wedge k1 \stackrel{\sim}{\approx} k2 \rightarrow \texttt{bind}\ t1\ k1 \approx \texttt{bind}\ t2\ k2$$

Fig. 5. Core equational theory of ITrees.

```
Theorem block_fusion_cfg_correct :
  forall (G : cfg dtyp),
    wf_cfg G ->
    ⟦ G ⟧ ≈ ⟦ block_fusion_cfg G ⟧.
Proof.
  intros G [WF1 WF2].
  unfold denote_cfg.
  simpl bind.
  unfold block_fusion_cfg.
  destruct (block_fusion G.(blks)) as [bks' [[src tgt] |]] eqn:EQ.
  - break_match_goal; [reflexivity |].
    simpl.
    apply Bool.orb_false_elim in Heqb as [INEQ1 INEQ2].
    unfold Eqv.eqv_dec in *.
    rewrite <- RelDec.neg_rel_dec_correct in INEQ1.
    rewrite <- RelDec.neg_rel_dec_correct in INEQ2.
    eapply block_fusion_correct_some
      with (f := G.(init)) (to := G.(init)) in EQ; auto.
    rewrite update_provenance_ineq in EQ; auto.
    eapply eutt_clo_bind; [apply EQ |].
    intros [[]|?] [[]|?] INV; try now inv INV.
    subst; reflexivity.
    eapply wf_cfg_src_not_in_phis; eauto.
    constructor; auto.
  - reflexivity.
Qed.
```