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…

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?
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
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

\[
\text{ColInductive itree } (E: \text{Type} \rightarrow \text{Type}) (R: \text{Type}): \text{Type} :=
\begin{align*}
&\text{Ret } (r: R) \\
&\text{Tau } (t: \text{itree } E R) \\
&\text{Vis } \{X: \text{Type}\} (e: E X) (k: X \rightarrow \text{itree } E R).
\end{align*}
\]

A value of the datatype \((\text{itree } E R)\) represents:
Interaction Trees

\[
\text{ColInductive } \text{itree} (E: \text{Type} \to \text{Type}) (R: \text{Type}) : \text{Type} := \\
| \text{Ret} (r: R) \\
| \text{Tau} (t: \text{itree} E R) \\
| \text{Vis} \{X: \text{Type}\} (e: E X) (k: X \to \text{itree} E R).
\]

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

\[
\text{CoInductive } \text{itree } (E : \text{Type} \to \text{Type}) (R : \text{Type}) : \text{Type} := \\
\text{Ret } (r : R) \\
\text{Tau } (t : \text{itree } E R) \\
\text{Vis } \{X : \text{Type}\} (e : E X) (k : X \to \text{itree } E R).
\]

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

- a potentially diverging computation,
A value of the datatype (itree E R) represents:

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

ColInductive $\text{itree} (E : \text{Type} \rightarrow \text{Type}) (R : \text{Type}) : \text{Type} :=$
\begin{align*}
\text{Ret} \ (r : R) \\
\text{Tau} \ (t : \text{itree} \ E \ R) \\
\text{Vis} \ \{X : \text{Type}\} \ (e : E \ X) \ (k : X \rightarrow \text{itree} \ E \ R).
\end{align*}

A value of the datatype ($\text{itree} \ E \ R$) 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

```
ColInductive 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

\[
\text{ColInductive } \text{itree } (E: \text{Type} \to \text{Type}) (R: \text{Type}): \text{Type} := \\
\text{Ret } (r: R) \\
\text{Tau } (t: \text{itree } E R) \\
\text{Vis } \{X: \text{Type}\} (e: E X) (k: X \to \text{itree } E R).
\]

A value of the datatype \((\text{itree } E R)\) 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:
* Composable 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**

- \text{Ret 1789}

**Silent divergence**

- \text{CoFixpoint spin := Tau spin}

\[
\begin{align*}
\tau & \rightarrow \tau \\
\tau & \rightarrow \tau \\
\end{align*}
\]
ITrees Come in All Shapes and Forms

<table>
<thead>
<tr>
<th>Pure computations</th>
<th>Ret 1789</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tau (Tau (Ret 1776))</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Silent divergence</th>
<th>CoFixpoint spin := Tau spin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tau (Tau (Vis e (fun b =&gt; match b with k true =&gt; Ret 1789 k false =&gt; Tau (Ret 1776) end)))</td>
<td></td>
</tr>
<tr>
<td>spin := Tau spin</td>
<td></td>
</tr>
<tr>
<td>Tau (Tau (Ret 1776))</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Effectful computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tau (Tau (Vis e (fun b =&gt; match b with k true =&gt; Ret 1789 k false =&gt; Tau (Ret 1776) end)))</td>
</tr>
<tr>
<td>spin := Tau spin</td>
</tr>
<tr>
<td>Tau (Tau (Ret 1776))</td>
</tr>
</tbody>
</table>
ITrees Come in All Shapes and Forms

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 \( \{ X : Type \} \) \( (x : X) : \text{itree} E X := \text{Ret} x \)

CoFixpoint bind \( \{ R S \} \) \( (t : \text{itree} E R) (k : R \rightarrow \text{itree} E S) : \text{itree} E S := \)

\[
\text{match} \ t \ \text{with} \\
\quad \text{| Ret } r \Rightarrow k \ r \\
\quad \text{| Tau } t \Rightarrow \text{Tau} (\text{bind} \ t \ k) \\
\quad \text{| Vis } e \ h \Rightarrow \text{Vis} \ e \ \text{(fun} \ x \Rightarrow \text{bind} \ (h \ x) \ k) \\
\text{end.}
\]
Composing Computations: the ITree Monad

Monadic structure

**Definition** \( \text{ret} \{ X \colon \text{Type} \} (x \colon X) \colon \text{itree} \ E \ X \equiv \text{Ret} \ x \)

**CoFixpoint** \( \text{bind} \{ R, S \} (t \colon \text{itree} \ E \ R) (k \colon R \rightarrow \text{itree} \ E \ S) \colon \text{itree} \ E \ S \equiv \)

\[
\begin{align*}
\text{match } & t \text{ with} \\
\mid \text{Ret} \ r & \Rightarrow k \ r \\
\mid \text{Tau} \ t & \Rightarrow \text{Tau} \ (\text{bind} \ t \ k) \\
\mid \text{Vis} \ e \ h & \Rightarrow \text{Vis} \ e \ (\text{fun} \ x \Rightarrow \text{bind} \ (h \ x) \ k) \\
\end{align*}
\]

\text{end}.

\[
\begin{align*}
\text{Notation:} & \\
& x \leftarrow s \ ;; \ k \\
& \triangleq \\
& \text{bind} \ s \ (\text{fun} \ x \Rightarrow k)
\end{align*}
\]
Composing Computations: the ITree Monad

Monadic structure

**Definition** \( \text{ret} \{ X : \text{Type} \} (x : X) : \text{itree} \ E \ X := \text{Ret} \ x \)

**CoFixpoint** \( \text{bind} \{ R S \} (t : \text{itree} \ E \ R) (k : R \rightarrow \text{itree} \ E \ S) : \text{itree} \ E \ S := \)

\[
\begin{align*}
&\text{match } t \text{ with} \\
&\quad | \text{Ret } r \Rightarrow k \ r \\
&\quad | \text{Tau } t \Rightarrow \text{Tau} \ (\text{bind } t \ k) \\
&\quad | \text{Vis } e \ h \Rightarrow \text{Vis } e \ (\text{fun } x \Rightarrow \text{bind} \ (h \ x) \ k)
\end{align*}
\]

end.

Monad laws:

**ret_bind**: \( x \leftarrow \text{ret } v ;; k \ x \approx k \ v \)

**bind_ret**: \( x \leftarrow t ;; \text{ret } x \approx t \)

**bind_bind**: \( x \leftarrow (y \leftarrow s ;; t) ;; u \approx y \leftarrow s ;; x \leftarrow t ;; u \)

Notation:

\( x \leftarrow s ;; k \)

\( \triangleq \)

bind \( s \) (fun \( x \Rightarrow k) \)
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.`

Monad laws:

- **ret_bind**: \( x \leftarrow \text{ret } v ;; k x \quad \approx \quad k v \)
- **bind_ret**: \( x \leftarrow t ;; \text{ret } x \quad \approx \quad t \)
- **bind_bind**: \( x \leftarrow (y \leftarrow s ;; t) ;; u \quad \approx \quad y \leftarrow s ;; x \leftarrow t ;; u \)

ITree equivalence?
ITree Equivalence

Option 1: Coq’s propositional equality?

\[ t \approx s \iff t = s \]

```coq
Inductive eq {X: Type}: Prop :=
  I eq_refl: forall (x: X), eq x x.
```
ITree Equivalence

Option 1: Coq’s propositional equality?

\[ t \simeq s \iff t = s \]

\[ \text{Inductive eq \{X: Type\}: Prop :=} \]
\[ \text{eq_refl: \forall (x: X), eq x x.} \]

\( \not\vdash \text{spin = Tau spin} \)
ITree Equivalence

Option 2: Strong bisimulation?
ITree Equivalence

Option 2: Strong bisimulation?

\[ t \sim s \triangleq \text{bisim } t \text{ s} \]

**Inductive bisimF** (\( \text{sim}: \text{relation (itree E R)} \)): \( \text{relation (itree E R)} := \)

- EqRet: \( \text{bisimF } (\text{Ret } v) (\text{Ret } v) \)
- EqTau: \( \text{sim } t \text{ s} \rightarrow \text{bisimF } \text{sim } (\text{Tau } t) (\text{Tau } s) \)
- EqVis (\( e: \text{E X} \)): (forall (\( v: X \), \( \text{sim } (k1 v) (k2 v) \))
  \( \rightarrow \text{bisimF } \text{sim } (\text{Vis } e k1) (\text{Vis } e k2) \)
ITree Equivalence

Option 2: Strong bisimulation?

\[ t \approx s \trianglelefteq bisim \ t \ s \]

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

1. EqRet: \( bisimF \ (Ret \ v) \ (Ret \ v) \)
2. EqTau: \( sim \ t \ s \rightarrow bisimF \ sim \ (Tau \ t) \ (Tau \ s) \)
3. EqVis (\( e: E X \)): (forall (\( v: X \)), \( sim \ (k1 \ v) \ (k2 \ v) \))
   \[ \rightarrow bisimF \ sim \ (Vis \ e \ k1) \ (Vis \ e \ k2) \]
ITree Equivalence

Option 2: Strong bisimulation?

\[ t \simeq s \triangleq \text{bisim } t \ s \]

**Inductive bisimF** (\( \text{sim} \): relation (itree E R)): relation (itree E R) :=

1. **EqRet:** \( \text{bisimF} (\text{Ret } v) (\text{Ret } v) \)
2. **EqTau:** \( \text{sim } t \ s \rightarrow \text{bisimF sim } (\text{Tau } t) (\text{Tau } s) \)
3. **EqVis (e: E X):** (forall (v: X), \( \text{sim } (k1 \ v) (k2 \ v) \))
   \( \rightarrow \text{bisimF sim } (\text{Vis e } k1) (\text{Vis e } k2) \)
ITree Equivalence

Option 2: Strong bisimulation?

\[ t \sim s \triangleq \text{bisim } t \text{ s} \]

\textbf{Inductive bisimF} (\(\text{sim}: \text{relation (itree E R)): \text{relation (itree E R)}) := \)

1. EqRet: \(\text{bisimF (Ret } v) \text{ (Ret } v)\)
2. EqTau: \(\text{sim } t \text{ s } \rightarrow \text{bisimF sim (Tau } t) \text{ (Tau } s)\)
3. EqVis (\(e: \text{E } X): (\text{forall (v: } X), \text{sim (k1 } v) \text{ (k2 } v)) \rightarrow \text{bisimF sim (Vis } e \text{ k1) (Vis } e \text{ k2)}\)

\[ 1789 \equiv 1789 \]

\[ \tau \equiv \tau \]

\[ e \equiv e \]
ITree Equivalence

Option 2: Strong bisimulation?

\[ t \sim s \triangleq \text{bisim } t \text{ s} \]

Inductive \( \text{bisimF} \) \((\sim: \text{relation } (\text{itree } E \text{ R})): \text{relation } (\text{itree } E \text{ R}) := \)

- EqRet: \( \text{bisimF} (\text{Ret } v) (\text{Ret } v) \)
- EqTau: \( \sim t \text{ s } \rightarrow \text{bisimF} \sim (\text{Tau } t) (\text{Tau } s) \)
- EqVis \((e: E X)\): \((\text{forall } (v: X), \sim (k1 v) (k2 v)) \rightarrow \text{bisimF} \sim (\text{Vis } e k1) (\text{Vis } e k2) \)

\( \approx \)
**ITree Equivalence**

**Option 2: Strong bisimulation?**

\[ t \simeq s \triangleq \text{bisim} \ t \ s \]

**Inductive** \( \text{bisimF} \) \((\text{sim}: \text{relation} (\text{itree} \ E \ R)): \text{relation} (\text{itree} \ E \ R)) := \)

1. **EqRet**: \( \text{bisimF} (\text{Ret} \ v) (\text{Ret} \ v) \)
2. **EqTau**: \( \text{sim} t s \rightarrow \text{bisimF} \text{sim} (\text{Tau} t) (\text{Tau} s) \)
3. **EqVis** \((e: E X)\): \((\forall (v: X), \text{sim} (k1 v) (k2 v)) \rightarrow \text{bisimF} \text{sim} (\text{Vis} e k1) (\text{Vis} e k2)\)

\[ \text{bisim} \ t \ s \triangleq \text{paco bisimF bot} \]

[https://github.com/snu-sf/paco]
**ITree Equivalence**

**Option 2: Strong bisimulation?**

\[ t \sim s : \triangleq \text{bisim } t \ s \]

**Inductive bisimF**  
\[ \text{sim}: \text{relation (itree } E \text{ R)}: \text{relation (itree } E \text{ R)} := \]

- \[ \text{EqRet}: \text{bisimF (Ret } v \text{)} (\text{Ret } v) \]
- \[ \text{EqTau}: \text{sim } t \ s \rightarrow \text{bisimF sim (Tau } t \text{)} (\text{Tau } s) \]
- \[ \text{EqVis (e: E X)}: \text{(forall } \text{(v: X), sim (k1 } v \text{)} (k2 } v\text{))} \rightarrow \text{bisimF sim (Vis } e \text{ k1)} (\text{Vis } e \text{ k2)} \]

\[ \vdash \text{Tau spin} \sim \text{spin} \]

\[ \text{bisim } t \ s \triangleq \text{paco bisimF bot} \]

[https://github.com/snu-sf/paco](https://github.com/snu-sf/paco)
ITree Equivalence

Equivalence Up-To Tau

\[ \tau \approx \]
ITree Equivalence

Equivalence Up-To Tau

t \approx s \triangleq \text{eutt } t \ s

Inductive \text{eutt} (\text{sim}: \text{relation (itree E R)}): \text{relation itree E R} :=

1. EqRet: \text{eutt} (\text{Ret } v) (\text{Ret } v)
2. EqTau: sim t s \rightarrow \text{eutt} \ sim (\text{Tau } t) (\text{Tau } s)
3. EqVis (e: E X): (\text{forall } (v: X), \text{sim } (k1 v) (k2 v))
   \rightarrow \text{eutt} \ sim (\text{Vis } e k1) (\text{Vis } e k2)
4. EqTauL: \text{eutt} \ sim t s \rightarrow \text{eutt} \ sim (\text{Tau } t) s
5. EqTauR: \text{eutt} \ sim t s \rightarrow \text{eutt} \ sim t (\text{Tau } s)

\text{eutt } t \ s \triangleq \text{paco eutt} \ \text{bot}2

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

Equivalence Up-To Tau

\[ t \approx s \iff \text{eutt } t s \]

Inductive \( \text{euttF} \) (sim: relation (itree E R)): relation itree E R :=

| EqRet: \( \text{euttF} \) (Ret \( v \)) (Ret \( v \)) |
| EqTau: sim \( t \) \( s \) \( \rightarrow \) \( \text{euttF} \) sim (Tau \( t \)) (Tau \( s \)) |
| EqVis \( (e: E X) \): (forall \( (v: X) \), sim \( (k1 \ v) \) \( (k2 \ v) \)) \( \rightarrow \) \( \text{euttF} \) sim (Vis \( e \) \( k1 \)) (Vis \( e \) \( k2 \)) |
| EqTauL: \( \text{euttF} \) sim \( t \) \( s \) \( \rightarrow \) \( \text{euttF} \) sim (Tau \( t \)) \( s \) |
| EqTauR: \( \text{euttF} \) sim \( t \) \( s \) \( \rightarrow \) \( \text{euttF} \) sim \( t \) (Tau \( s \)) |

\( \text{eutt} \) \( t \) \( s \) \( \triangleq \) \( \text{paco euttF bot2} \)

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 \texttt{imp} : Type :=
\begin{itemize}
  \item Skip
  \item Assign (x: var) (e: exp)
  \item Seq (c1 c2: imp)
  \item If (b: exp) (t e: imp)
  \item While (b: exp) (c: imp).
\end{itemize}

Our objective:
\begin{itemize}
  \item Give a denotation to imp
  \item That is executable
  \item Suitable to verify a compiler
\end{itemize}
Everyone’s Favorite Case Study: Imp

Inductive \texttt{imp} : Type :=
\begin{itemize}
  \item Skip
  \item Assign (x: var) (e: exp)
  \item Seq (c1, c2: imp)
  \item If (b: exp) (t, e: imp)
  \item While (b: exp) (c: imp).
\end{itemize}

Our objective:
\begin{itemize}
  \item Give a denotation to imp
  \item That is executable
  \item Suitable to verify a compiler
\end{itemize}

Proceeds in two steps
\begin{itemize}
  \item Syntax is denoted in terms of itrees;
  \item Events contained in the trees are given a semantics into a monad.
\end{itemize}
Imp Programs as ITrees

Denotation of imp in term of itrees:

```ocaml
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 \( \text{den}_\text{imp} (c: \text{imp}): \text{itree E_imp unit} := \)

\[
\begin{align*}
\text{match } c \text{ with} \\
\text{I Skip } & \Rightarrow \text{ret tt} \\
\text{I Assign } x \ e & \Rightarrow v \leftarrow \text{den_exp e ;; trigger (EWrite x v)} \\
\text{I Seq } c1 \ c2 & \Rightarrow \text{den_imp c1 ;; den_imp c2} \\
\text{I If } b \ t \ e & \Rightarrow v \leftarrow \text{den_exp b ;;}
\quad \text{if is_true v then den_imp t else den_imp e} \\
\text{I While } b \ c & \Rightarrow ??? \\
\text{end.}
\end{align*}
\]

Effect interface of Imp:

Inductive E_imp : Type -> Type :=

\[
\begin{align*}
\text{I ERead } (x: \text{var}) & \quad : \text{E_imp value} \\
\text{I EWrite } (x: \text{var}) (v: \text{value}) : \text{E_imp unit}
\end{align*}
\]
Imp Programs as ITrees

Denotation of imp in term of itrees:

Fixpoint \texttt{den\_imp} (c:\imp): 
\texttt{itree E\_imp unit :=}
\begin{align*}
\text{match } c \text{ with} \\
\text{ Skip } & \to \text{ ret } \texttt{tt} \\
\text{ Assign } x \ e & \to \ v \leftarrow \text{ den\_exp } e \ ;; \text{ trigger } (\text{EWrite } x \ v) \\
\text{ Seq } c_1 c_2 & \to \text{ den\_imp } c_1 \ ;; \text{ den\_imp } c_2 \\
\text{ If } b \ t \ e & \to \ v \leftarrow \text{ den\_exp } b \ ;; \text{ if } \text{is\_true } v \text{ then } \text{den\_imp } t \text{ else } \text{den\_imp } e \\
\text{ While } b \ c & \to \ ??? \\
\text{ end.}
\end{align*}

Effect interface of Imp:

\begin{align*}
\textbf{Inductive E\_imp : Type } \to \text{ Type } := \\
\text{ ERead } (x: \text{ var}) & \quad : E\_imp \ \text{value} \\
\text{ EWrite } (x: \text{ var}) \ (v: \text{ value}): & \quad E\_imp \ \text{unit}
\end{align*}
Imp Programs as ITrees

Denotation of imp in term of itrees:

\[
\text{Fixpoint } \text{den}_\text{imp} (c : \text{imp}): \text{itree } E_{\text{imp}} \text{ unit} := \\
\text{match } c \text{ with} \\
\text{ Skip } \Rightarrow \text{ ret } tt \\
\text{ Assign x e } \Rightarrow v \leftarrow \text{ den}_\text{exp} e ;; \text{ trigger } (E\text{Write} x v) \\
\text{ Seq c1 c2 } \Rightarrow \text{ den}_\text{imp} c1 ;; \text{ den}_\text{imp} c2 \\
\text{ If b t e } \Rightarrow v \leftarrow \text{ den}_\text{exp} b ;; \\
\quad \text{ if is_true } v \text{ then } \text{ den}_\text{imp} t \text{ else } \text{ den}_\text{imp} e \\
\text{ While b c } \Rightarrow ??? \\
\text{ end.}
\]

Effect interface of Imp:

\[
\text{Inductive } E_{\text{imp}} : \text{Type} \rightarrow \text{Type} := \\
\text{ ERead (x: var) : E_{imp} value} \\
\text{ EWrite (x: var) (v: value): E_{imp} unit}
\]
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 ;;
| 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 ;;
  | 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:

```coq
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:

\[
\text{Fixpoint } \text{den}_\text{imp} (c: \text{imp}): \text{itree } \text{E}\_\text{imp} \text{ unit} :=
\]
\[
\begin{align*}
\text{match } c \text{ with} \\
\text{Skip} & \Rightarrow \text{ret } \text{ tt} \\
\text{Assign } x \ e & \Rightarrow v \Leftarrow \text{den}_{\text{exp}} e \;; \text{trigger } (\text{EWrite } x \ v) \\
\text{Seq } c_1 \ c_2 & \Rightarrow \text{den}_\text{imp} c_1 ;; \text{den}_\text{imp} c_2 \\
\text{If } b \ t \ e & \Rightarrow \begin{cases} \\
& v \Leftarrow \text{den}_{\text{exp}} b ;; \\
& \text{if } \text{is_true } v \text{ then } \text{den}_\text{imp} t \text{ else } \text{den}_\text{imp} e \\
\end{cases} \\
\text{While } b \ c & \Rightarrow ??? \\
\end{align*}
\]

Effect interface of Imp:

\[
\text{Inductive } \text{E}\_\text{imp} : \text{Type} \rightarrow \text{Type} :=
\]
\[
\begin{align*}
\text{ERead } (x: \text{var}) & : \text{E}\_\text{imp} \text{ value} \\
\text{EWrite } (x: \text{var}) \ (v: \text{value}) : \text{E}\_\text{imp} \text{ unit}
\end{align*}
\]

Minimal effectful computation:

\[
\text{Definition } \text{trigger } \{ \text{E} \ X\}(e: \text{E} \ X): \text{itree } \text{E} \ X :=
\]
\[
\text{Vis } e \ (\text{fun } x \Rightarrow \text{Ret } x)
\]
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:

```fsharp
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 ;
  else ret tt
```
An Iteration Combinator

One would like to write:

\[
\text{den\_imp (while } b \text{ do } c) = \?
\]
\[
v \leftarrow \text{den\_exp } b ;
\]
\[
\text{if is\_true } v
\]
\[
\text{then } \text{den\_imp } c ; ; \text{den\_imp (while } b \text{ do } c)
\]
\[
\text{else } \text{ret tt}
\]

Continuation trees:

\textbf{Definition} \ ktree \ E \ A \ B := A \rightarrow \text{itree } E \ B.

Continuation trees have a nice structure:
An Iteration Combinator

One would like to write:

```plaintext
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 $ktree \ E \ A \ B := A \rightarrow itree \ E \ B$.

Continuation trees have a nice structure:

- They can be composed;

  $k1 >>> k2$
An Iteration Combinator

One would like to write:

```plaintext
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:

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

Continuation trees have a nice structure:

- They can be composed;
  - \( k_1 >>> k_2 \)
- They support case analysis;
  - case \( k_1 \) \( k_2 \)
An Iteration Combinator

One would like to write:

```plaintext
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:

```plaintext
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;  \( \text{case } k1 \ k2 \)
- They can be iterated over!  \( \text{iter } k \)
An Iteration Combinator

Continuation trees:

Definition \( ktree \ E \ A \ B := A \rightarrow itree \ E \ B. \)

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** \( \text{ktree} \ E \ A \ B := A \rightarrow \text{itree} \ E \ B. \)

Iteration combinator:

**CoFixpoint** \( \text{iter} \) (body: ktree \( E \ A \) (A + B)): ktree \( E \ A \) B :=

\[
\begin{align*}
& \text{fun } a \Rightarrow ab \leftarrow \text{body } a \ ; ; \\
& \text{match } ab \text{ with} \\
& \text{inl } a \Rightarrow \text{Tau (iter body } a) \\
& \text{inr } b \Rightarrow \text{Ret } b \\
& \text{end.}
\end{align*}
\]

Termination
An Iteration Combinator

Continuation trees:

Definition \( \text{ktree} \ E \ A \ B := A \rightarrow \text{itree} \ E \ B. \)

Iteration combinator:

CoFixpoint \( \text{iter} \) (body: \( \text{ktree} \ E \ A \ (A + B) \)): \( \text{ktree} \ E \ A \ B := \)

\[
\begin{align*}
\text{fun} \ a & \Rightarrow \text{ab} \leftarrow \text{body} \ a ;; \\
\text{match} \ \text{ab} \ \text{with} \\
\text{l} \ \text{inl} \ a & \Rightarrow \text{Tau} \ (\text{iter} \ \text{body} \ a) \\
\text{l} \ \text{inr} \ b & \Rightarrow \text{Ret} \ b \\
\text{end}.
\end{align*}
\]

New iteration (guarded)

Termination
An Iteration Combinator

Continuation trees:

\[
\text{Definition } \text{ktree } E \text{ A } B := A \rightarrow \text{itree } E \text{ B.}
\]

Iteration combinator:

\[
\text{CoFixpoint } \text{iter } (\text{body}: \text{ktree } E \text{ A } (A + B)) : \text{ktree } E \text{ A } B :=
\]
\[
\begin{array}{l}
\text{fun } a => ab \leftarrow \text{body } a ;;
\text{match } ab \text{ with}
| \text{inl } a => \text{Tau } (\text{iter } \text{body } a)
| \text{inr } b => \text{Ret } b
\text{end.}
\end{array}
\]

One would like to write:

\[
\text{den_imp } (\text{while } b \text{ do } c) =?
\]
\[
\begin{array}{l}
v \leftarrow \text{den_exp } b ;;
\text{if is_true } v
\text{then den_imp } c ;; \text{den_imp } (\text{while } b \text{ do } c)
\text{else ret tt}
\end{array}
\]

One can write:

\[
\text{den_imp } (\text{while } b \text{ do } c) = \text{iter}
\begin{array}{l}
(\text{fun } _ => v \leftarrow \text{den_exp } b ;;
\text{if is_true } v
\text{then den_imp } c ;; \text{ret } (\text{inl } tt)
\text{else ret } (\text{inr } tt))
\end{array}
\]

New iteration (guarded)
Termination
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

\[
\text{Inductive } E\text{\textunderscore imp} : \text{Type} \to \text{Type} := \\
\mid \text{ERead} (x: \text{var}) \quad : \ E\text{\textunderscore imp} \ \text{value} \\
\mid \text{EWrite} (x: \text{var}) \ (v: \text{value}) \quad : \ E\text{\textunderscore imp} \ \text{unit}
\]
Giving Meaning to Events: Handlers

**Inductive** $\text{E_imp} : \text{Type} \to \text{Type} :=$

- $\text{ERead} (x : \text{var}) : \text{E_imp} \text{ value}$
- $\text{EWrite} (x : \text{var}) (v : \text{value}) : \text{E_imp} \text{ unit}$

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

**Definition** $\text{handler} (E \ M : \text{Type} \to \text{Type}) := E \simto M.$
Giving Meaning to Events: Handlers

\textbf{Inductive} \textit{E\_imp} : Type -> Type :=

\begin{itemize}
  \item \textit{ERead} (x: \textit{var}) \quad : \textit{E\_imp} \textit{value}
  \item \textit{EWrite} (x: \textit{var}) (v: \textit{value}) : \textit{E\_imp} \textit{unit}
\end{itemize}

Events are given \textit{meaning} by handling them into monads:

\textbf{Definition} \textit{handler} (E M: Type -> Type) := E ~> M.

\textbf{Notation:}
\[ E ~> M \triangleq \forall X, E X \rightarrow M X \]
Giving Meaning to Events: Handlers

Inductive \(E_{\text{imp}} : \text{Type} \rightarrow \text{Type} :=\)

\(\text{ERead (x: var)} ~ : E_{\text{imp}} \text{ value}\)
\(\text{EWrite (x: var) (v: value)} : E_{\text{imp}} \text{ unit}\)

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

Definition \(\text{handler (E M: Type} \rightarrow \text{Type)} := E \rightarrow M.\)

Notation: \(E \rightarrow M \triangleq \text{forall X, } E \text{ X } \rightarrow M \text{ X}\)

Let's handle \(E_{\text{imp}}\) into the state monad.

Definition \(h_{\text{imp}} : E_{\text{imp}} \rightarrow \text{stateT (itree voidE)} := \)

\(\text{fun X e s => match e with}\)

\(\text{ERead x => Ret (s , s[x])}\)
\(\text{EWrite x v => Ret (s[x <- v], tt )}\)
\(\text{end}\)
Lifting Meaning to ITrees: Interpreters

The library provides an interpretation function:

$$\text{interp} (h : E \rightarrow M) : \text{itree} \ E \ R \rightarrow M \ R$$

Assuming that the monad $M$ supports a notion of iteration:

Class MonadIter ($M : \text{Type} \rightarrow \text{Type} ) : \text{Type} :=$

iter : forall {R A : \text{Type}}
  (body : A \rightarrow M (A + R)),
  A \rightarrow M R.$
Lifting Meaning to ITrees: Interpreters

The library provides an interpretation function:

\[
\text{interp} \ (h: \ E \ \Rightarrow \ M): \ \text{itree} \ E \ R \ \Rightarrow \ M \ R
\]

Assuming that the monad \( M \) supports a notion of iteration:

Class \( \text{MonadIter} \ (M : \ Type \ \Rightarrow \ Type) : \ Type \ := \ \\
\text{iter} : \ \forall \ \{R \ A: \ Type\} \ \\
\text{body}: \ A \Rightarrow M \ (A + R), \ \\
A \Rightarrow M \ R. \]

\[
\begin{align*}
\text{supports it:} & \quad \text{itree} \ E \\
& \quad \text{Prop}
\end{align*}
\]

\[
\begin{align*}
\text{preserves it:} & \quad \text{stateT} \ M \\
& \quad \text{readerT} \ M \\
& \quad \text{optionT} \ M \\
& \quad \text{eitherT} \ M
\end{align*}
\]
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:

```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:

```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: \( (t1 \approx t2 \land k1 \approx k2) \rightarrow (t1 ;; k1) \approx (t2 ;; k2) \)
What About Reasoning?

Rich equational reasoning over eutt (excerpt)

- Monad Laws: \((x \leftarrow t ;; x) \approx t\)
- Structural Laws: \((\text{Tau } t) \approx t\)
- Congruence Laws: \((t1 \approx t2 \land k1 \approx k2) \rightarrow (t1 ;; k1) \approx (t2 ;; k2)\)
- Monoidal Laws: \((\text{inl } >>> \text{ case } h k) \approx h\)
What About Reasoning?

Rich equational reasoning over eut (excerpt)

- Monad Laws: \((x \leftarrow t ;; x) \approx t\)
- Structural Laws: \((\text{Tau } t) \approx t\)
- Congruence Laws: \((t_1 \approx t_2 \land k_1 \approx k_2) \rightarrow (t_1 ;; k_1) \approx (t_2 ;; k_2)\)
- Monoidal Laws: \((\text{inl} >>> \text{case } h \ k) \approx h\)
- Iteration Laws: \((\text{iter } f) \approx (f >>> \text{case } (\text{iter } f) \ \text{id})\)
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: \((t1 \approx t2 \land k1 \approx k2) \rightarrow (t1 ;; k1) \approx (t2 ;; k2)\)
- Monoidal Laws: \((\text{inl } >>> \text{case } h \ k) \approx h\)
- Iteration Laws: \((\text{iter } f) \approx (f >>> \text{case } (\text{iter } f) \ \text{id})\)
- Interp Laws: \((\text{interp } h \ (\text{trigger } e)) \approx h \ e\)
  \((\text{interp } h \ (t ;; k)) \approx (x \leftarrow \text{interp } h \ t ;; \text{interp } h \ (k \ x))\)
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: \((t_1 \approx t_2 \land k_1 \approx k_2) \rightarrow (t_1 ;; k_1) \approx (t_2 ;; k_2)\)
- Monoidal Laws: \((\text{inl} \gggg case h k) \approx h\)
- Iteration Laws: \((\text{iter } f) \approx (f \gggg case (\text{iter } f) \text{ id})\)
- 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.

\[
egin{array}{c c c c}
    p & \xrightarrow{\text{den}_{\text{Imp}}} & t^E_{\text{Imp}} & \xrightarrow{\text{interp}_{\text{Imp}}} & t^\emptyset_{\text{Imp}} \\
    \downarrow & & \downarrow & & \downarrow & \xrightarrow{\text{eutt}_{\mathcal{R}_{\text{sim}}}} \\
    \mathcal{C}(p) & \xrightarrow{\text{den}_{\text{Asm}}} & t^F_{\text{asm}} & \xrightarrow{\text{interp}_{\text{Asm}}} & t^\emptyset_{\text{asm}}
\end{array}
\]
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:

ITrees Used in Projects

Embeds Haskell programs in Coq to verify them

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

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

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
define @factorial(%n) {
    %1 = alloca
    %acc = alloca
    store %n, %1
    store 1, %acc
    br label %start

    %3 = load %1
    %4 = icmp sgt %3, 0
    br %4, label %then, label %else

entry:

loop:

%6 = load %acc
%7 = load %1
%8 = mul %6, %7
store %8, %acc
%9 = load %1
%10 = sub %9, 1
store %10, %1
br label %start

body:

post:

%12 = load %acc
ret %12

post:
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**

\[
\text{denote}_{\text{llvm}} (p: \text{llvm}): \text{itree} E_{\text{llvm}} \text{ value := ...}
\]

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**\[\text{denote}_{\text{llvm}} (p: \text{llvm}) : \text{itree} \ E_{\text{llvm}} \ \text{value} := \ldots\]

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**

\[
\text{denote}_{\text{llvm}} (p : \text{llvm}) : \text{itree} E_{\text{llvm}} \text{ value := ...}
\]

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 \( \text{denote}_{\text{llvm}}(p:\text{llvm}): \text{itree} \ E_{\text{llvm}} \text{ value} := \ldots \)

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 \( \text{denote}_{\text{llvm}} (p: \text{llvm}) : \text{itree E}_{\text{llvm}} \text{ value} := \ldots \)

What kind of events can an llvm computation trigger?

- Global state
- Local state
- Stack of local frames
- Memory
- Pick

- 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
- Pick
- Undefined Behavior

- 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**

\[
denote_{\text{llvm}} (p: \text{llvm}): \text{itree} \ E_{\text{llvm}} \ value ::= \ldots
\]

What kind of events can an llvm computation trigger?

- Global state
- Local state
- Stack of local frames
- Memory
- Pick
- Undefined Behavior
- Calls

- 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 \( \text{denote}_{\text{llvm}}(p: \text{llvm}) : \text{itree} \ E_{\text{llvm}} \text{ value} := \ldots \)

What kind of events can an llvm computation trigger?

- Global state
- Local state
- Stack of local frames
- Memory
- Pick
- Undefined Behavior
- Calls
- Debugging

- 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**

\[
\text{denote}_{\text{LLVM}} (p: \text{LLVM}): \text{itree} \ E_{\text{LLVM}} \text{ value} := \ldots
\]

What kind of events can an LLVM computation trigger?

- Global state
- Local state
- Stack of local frames
- Memory
- Pick
- Undefined Behavior
- Calls
- Debugging
- Failure

- 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**
\[ \text{denote}_{\text{llvm}}(p: \text{llvm}): \text{itree E}_{\text{llvm}} \text{ value} := \ldots \]

What kind of events can an llvm computation trigger?

- Global state
- Local state
- Stack of local frames
- Memory
- Pick
- Undefined Behavior
- Calls
- Debugging
- Failure

\[ \text{MPush/MPop} \]
\[ \text{Load(t,l)/Store(a,v)} \]
\[ \text{Alloca(t)} \]
\[ \text{GEP(t,v,vs)} \]
\[ \text{PtoI(a)/ItoP(i)} \]

Raises challenges to compose interfaces!
To Some Extent: Same Story on Another Scale

entry:
%1 = alloca
%acc = alloca
store %n, %1
store 1, %acc
br label %start

loop:
%3 = load %1
%4 = icmp sgt %3, 0
br %4, label %then, label %else

body:
%6 = load %acc
%7 = load %1
%8 = mul %6, %7
store %8, %acc
%9 = load %1
%10 = sub %9, 1
store %10, %1
br label %start

post:
%12 = load %acc
ret %12
To Some Extent: Same Story on Another Scale

entry:

```
%1 = alloca
%acc = alloca
store %n, %1
store 1, %acc
br label %start
```

```
%3 = load %1
%4 = icmp sgt %3, 0
br %4, label %then, label %else
```

```
%6 = load %acc
%7 = load %1
%8 = mul %6, %7
store %8, %acc
%9 = load %1
%10 = sub %9, 1
store %10, %1
br label %start
```

```
%12 = load %acc
ret %12
```

Fixpoint den_exp t e : itree exp_E value
To Some Extent: Same Story on Another Scale

```assembly
entry:
%1 = alloca
%acc = alloca
store %n, %1
store 1, %acc
br label %start

%3 = load %1
%4 = icmp sgt %3, 0
br %4, label %then, label %else

loop:

body:
%6 = load %acc
%7 = load %1
%8 = mul %6, %7
store %8, %acc
%9 = load %1
%10 = sub %9, 1
store %10, %1
br label %start

post:
%12 = load %acc
ret %12

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

entry:

%1 = alloca
%acc = alloca
store %n, %1
store 1, %acc
br label %start

%3 = load %1
%4 = icmp sgt %3, 0
br %4, label %then, label %else

loop:

body:

%6 = load %acc
%7 = load %1
%8 = mul %6, %7
store %8, %acc
%9 = load %1
%10 = sub %9, 1
store %10, %1
br label %start

post:

%12 = load %acc
ret %12

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

entry:

%1 = alloca
%acc = alloca
store %n, %1
store 1, %acc
br label %start

%3 = load %1
%4 = icmp sgt %3, 0
br %4, label %then, label %else

loop:

%6 = load %acc
%7 = load %1
%8 = mul %6, %7
store %8, %acc
%9 = load %1
%10 = sub %9, 1
store %10, %1
br label %start

body:

%12 = load %acc
ret %12

post:

Fixpoint den_exp_te : 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

entry:

%1 = alloca
%acc = alloca
store %n, %1
store 1, %acc
br label %start

%3 = load %1
%4 = icmp sgt %3, 0
br %4, label %then, label %else

loop:

%6 = load %acc
%7 = load %1
%8 = mul %6, %7
store %8, %acc
%9 = load %1
%10 = sub %9, 1
store %10, %1
br label %start

body:

%12 = load %acc
ret %12

post:

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

entry:

```c
%1 = alloca
%acc = alloca
store %n, %1
store 1, %acc
br label %start
```

loop:

```c
%3 = load %1
%4 = icmp sgt %3, 0
br %4, label %then, label %else
```

body:

```c
%6 = load %acc
%7 = load %1
%8 = mul %6, %7
store %8, %acc
%9 = load %1
%10 = sub %9, 1
store %10, %1
br label %start
```

post:

```c
%12 = load %acc
ret %12
```

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

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
interpGlobals
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

itree E0 value

itree E1 (gstate * value)

itree E2 (lstack * (gstate * value))

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

interp_intrinsics

interpGlobals

interpLocals

interpMemory
A Chain of Interpreters

denote llvm p

itree E llvm value

itree E0 value

itree E1 (gstate * value)

itree E2 (lstack * (gstate * value))

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

model undef

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

interp intrinsics

interp globals

interp locals

interp memory
A Chain of Interpreters

denote_llvm p

itree E_llvm value

itree E0 value

itree E1 (gstate * value)

itree E2 (lstack * (gstate * value))

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

model_undefined

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

model_UB

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

interp_intrinsics

interp_globals

interp_locals

interp_memory
A Chain of Interpreters

- denote_llvm p
- itree E_llvm value
- itree E0 value
- itree E1 (gstate * value)
- itree E2 (lstack * (gstate * value))
- itree E3 (memory * (lstack * (gstate * value)))

- interp_intrinsics
- interpGlobals
- interp_locals
- interp_memory

- model_undef
- execute_undef

- itree E4 (memory * (lstack * (gstate * value))) -> Prop
- model_UB
- itree E5 (memory * (lstack * (gstate * value))) -> Prop

- execute_undef
- fail_UB
- itree E4 (memory * (lstack * (gstate * value)))
- itree E5 (memory * (lstack * (gstate * value)))
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.
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.