Modular Semantics ▸ (and Meta-theory) for LLVM IR

Cambium Seminar

Irene Yoon
12 / 04 / 2024

Joint work with collaborators at...
LLVM Compiler Infrastructure [Lattner et al.]
A modular and reusable infrastructure for compiler pipelines
Understanding LLVM Intermediate Representation (IR)

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

typed SSA IR

analysis

optimizations/
transformations
CompCert [Leroy et al.]

Why 🎉 for a compiler infrastructure?
The Vellvm Project ("Vellvm 1.0")

- [Zhao and Zdancewic - CPP 2012] Verified computation of dominators
- [Zhao et al. - POPL 2012] Formal semantics of IR + verified SoftBound
- [Zhao et al. - POPL 2013] Verification of (v)mem2reg!

A success, but monolithic

\[ G \vdash pc, \text{mem} \rightarrow pc', \text{mem}' \]

https://github.com/vellvm/vellvm-legacy
Operational Semantics

- A single relation encompasses all aspects of the semantics
- The relation is propositional (and Coq "Prop" cannot be extracted), thus the semantics cannot be extracted
- Classic simulation proofs

Testing matters!
You can model the lambda-calculus as a mathematician.
To model C or LLVM, you need to embrace your physicist side!
The Vellvm Project, Revamped

https://github.com/vellvm/vellvm

Selected publications and drafts*

[Zakowski et al. - ICFP 2021]
Modular and executable semantics for LLVM IR

[Yoon et al. - ICFP 2022]
Meta-theory for layered monadic interpreters

[Zaliva et al.]
Verified HELIX front-end

[Beck et al.]
Infinite/finite memory model for LLVM IR

[Yoon et al.]
Relational separation logic for LLVM IR

*: all results mechanized in the Coq Proof Assistant
Modular and Executable Semantics for LLVM IR

joint work with
Yannick Zakowski, Calvin Beck, Ilia Zaichuk, Vadim Zaliva, Steve Zdancewic
LLVM Intermediate Representation

- LLVM IR
  - Control-flow Graphs:
    - Labeled blocks
    - Straight-line Code
    - Block Terminators
    - Static Single Assignment Form (phi-nodes)
- Types:
  - i64 $\Rightarrow$ 64-bit integers
  - i64* $\Rightarrow$ pointer

entry:

```
%1 = alloca %1
%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
```
LLVM LangRef

https://llvm.org/docs/LangRef.html
Studying the (formal) semantics of LLVM IR

Formal Semantics

Crellvm [Kang et al., 18]
K-LLVM [Li and Gunter, 20]
Vellvm [Zhao et al., 12]
Taming UB [Lee et al., 17]
Concurrency [Chakraborty and Vafeiadis, 17]

Realistic Memory Models

Integer-Pointer Cast [Kang et al., 15]
Twin-Allocation [Lee et al., 18]

Bug Finding

Alive [Lopes et al., 15]
Alive 2 [Lopes et al., 21]
**Vellvm 2.0: A redesign of Vellvm**

A Coq formal semantics for a large, sequential fragment of LLVM IR coming with:

- Interaction Trees (itrees)
  - [Xia et al. 2020](https://github.com/DeepSpec/InteractionTrees)
  - A generic toolkit to **define** and **reason about** the semantics of interactive systems
  - Semantics: **Compositional, Modular, Executable**
  - Reasoning: **Equational, termination sensitive**

- (Re)Vellvm
  - [Vellvm 2.0](https://github.com/vellvm/vellvm)
  - VIR: a **compositional, modular and executable** formal semantics for (sequential) LLVM IR
  - New reasoning principles to verify program transformations

---

**Notes:**

- [Xia et al. 2020](https://github.com/DeepSpec/InteractionTrees)
- [Vellvm 2.0](https://github.com/vellvm/vellvm)
Event-based Semantics

Modularity
Interaction Trees

A data structure for modelling potentially diverging programs in Coq

Programs as trees

[Xia et al. 2020]
**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
Basic combinators and notation

• pure computation: \texttt{ret }x

• monadic bind (sequence):
  • \texttt{bind }t \texttt{ k}
  • \texttt{x <- k ;; k x}

• performing an event: \texttt{trigger }e
  • note that an "event" is merely syntactic, until it is given a semantics via a handler
Two-phased Denotation

Phase 1
Control flow and potential divergence are internalized

Phase 2
The semantics of effects is introduced

Uninterpreted!

Write X 1 → {X → 1}
Read X
Write Y 0 → {X → 1, Y → 1}
Write Y 1 → {X → 1}

The tree is interpreted via an event handler

X = 1;
Y = X;

X = 1;
Y = X;

Answer = 1

The tree is interpreted via an event handler

{} → {X → 1}
{} → {X → 1}
{} → {X → 1, Y → 1}

Uninterpreted!
Defining a modular LLVM IR semantics

SSA $\approx$ functional program [Appel 1998]
+ 
  • Undefined values / poison
  • Effects
    • structured heap load/store
    • system calls (I/O)
  • Types & Memory Layout
    • structured, recursive types
    • type-directed projection

Q: How do we decompose these effects MODULARLY?

We know how to model this and prove properties about the models.
Vellvm Effects

1. External Calls
2. Intrinsics
3. Global Environment
4. Local Environment
5. Stack
6. Memory
7. Nondeterminism
8. Undefined Behavior
9. (Debugging)

Each layer of effects can be interpreted separately, making proofs modular and changes orthogonal.
Interpreting LLVM Events

1. Denoting events

(* Interactions with the memory *)

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

2. Giving semantics to events

Generalized semantic domain: the resulting events contain failure and undefined behavior events

Definition handle_memory {E} `{FailureE <- E} `{UBE <- E}: MemoryE ~> stateT memory_stack (itree E)

3. Fold and layer

Definition interp {E M : Type -> Type} (h : E ~> M) : itree E ~> M
Fold and layer?

3. Fold and layer

\[ \text{Definition interp \{E M : Type -> Type\} (h : E ~> M) : itree E ~> M} \]

An event handler \([h]\) defines a monad morphism (i.e. respects bind and ret*) for any monad \([M]\) with a loop operator.

“fold over a tree, and return something you can iterate over”

[Yoon et al. - ICFP 2022]
Meta-theory for layered monadic interpreters

*and additionally, rules for iteration
Layered interpretation, plug-and-play
Layered interpretation, plug-and-play

Event-based semantics for LLVM IR

Is this the right memory model? [Beck et al.]

Infinite/finite memory model for LLVM IR
Layered interpretation, plug-and-play

Event-based semantics for LLVM IR

VIR
↓ structural representation

Level 0

itree VellvmE V_u
↓ intrinsics

Level 1

itree E_0 V_u
↓ global environment

Level 2

stateT_{Env_G} (itree E_1) V_u
↓ local environment

Level 3

stateT_{Env_L * Env_G} (itree E_2) V_u
↓ memory model

Level 4

stateT_{Mem * Env_L * Env_G} (itree E_3) V_u

propositional model

stateT_{Mem * Env_L * Env_G} (itree E_4) V_u * P
model undef_{\tau}

stateT_{Mem * Env_L * Env_G} (itree E_5) V_u * P

executable interpreter

stateT_{Mem * Env_L * Env_G} (itree E_4) V_u
interpret undef_{\tau} = 0_{\tau}

stateT_{Mem * Env_L * Env_G} (itree E_5) V_u
Layered interpretation, plug-and-play

Model supports nondeterminism defines a set of possible behaviors. ⇒ to account for undef

Executable reference interpreter ⇒ for debugging and validation

Event-based semantics for LLVM IR
Denotational Semantics
Compositionality
Denotational semantics for LLVM IR

- **⟦ - ⟧_{expr}** expressions
- **⟦ - ⟧_{instr}** instructions
- **⟦ - ⟧_{block}** blocks
- **⟦ - ⟧_{cfg}** control-flow graphs
- **⟦ - ⟧_{llvm}** programs
  (mutually recursive functions)

Meaning built up by induction on the structure of the syntax.
- open programs
- fixpoint combinators
- pure monadic computations

A Denotational Semantics for LLVM IR
Denoting expressions

"Simple arithmetic"

• Division by zero
  • Undefined behavior: behavior undefined by the language standard ("no promises!")
  • Compilers often assume source has no UB

• undef: set of defined values at a certain type
  • \( x + x \neq 2 \times x \)

• poison: "deferred undefined behavior", a value useful for representing signed overflow in LLVM IR

\[
[e_1/e_2]_e = uv_1 \leftarrow [e_1]_e ;; uv_2 \leftarrow [e_2]_e ;;
\]
\[
dv \leftarrow \text{pick}(uv_2) ;; uv_1 \odot dv
\]
General recursion in Gallina

Gallina

pure fragment of OCaml, with dependent types

Fixpoint fact n :=
  match n with
  l 0 => 1
  l S m => n * fact m

Fixpoint zip xs ys :=
  match xs with
  l nil => ys
  l x::xs => x::zip ys xs

Fixpoint interp c s :=
  match c with
  l ..
  l while b c =>
    if is_true b s
    then interp (while b c) (interp c s)
    else s

Ok!  Prove it!  Tough luck!
Combinator for recursion

First step: modeling tail-recursive calls

- Iteration as a primitive for tail recursion
- Iterate a function updating an accumulator \([A]\), until it produces an output \([B]\)

\[
\text{iter : } (A \rightarrow \text{itree E } (A + B)) \rightarrow (A \rightarrow \text{itree E } B)
\]

\[
\text{CoFixpoint iter (body : A \rightarrow \text{itree E } (A + B)) : A \rightarrow \text{itree E } B :=}
\]
\[
\text{fun a \Rightarrow ab \Leftarrow body a ;;}
\]
\[
\text{match ab with}
\]
\[
| \text{inl a \Rightarrow Tau (iter body a)}
\]
\[
| \text{inr b \Rightarrow Ret } b
\]
\[
\text{end.}
\]

Iterative laws

\[
\text{iter } f \approx f >>> \text{case_ (iter } f \text{) id_}
\]
\[
\text{iter } f >>> g \approx \text{iter (f >>> bimap id_ } g)
\]
Mutual recursion combinator

- Technique for general recursion developed by McBride 2015

- Signature of a mutually recursive function

  \[
  \text{Inductive} \quad \text{ackermannE} : \text{Type} \to \text{Type} := \\
  | \quad \text{Ackermann} : \text{nat} \to \text{nat} \to \text{ackermannE} \text{ nat}.
  \]

- Body of function makes recursive calls with 'Ackermann' events without ensuring well-foundedness

  \[
  \text{Definition} \quad h\_\text{ackermann} : \text{ackermannE} \rightsquigarrow \text{itree} (\text{ackermannE} + \text{'emptyE}) := \\
  \text{fun} \_ \text{'}(\text{Ackermann} \ m \ n) \Rightarrow \text{if} \ m =? \ 0 \ \text{then} \ \text{Ret} (n + 1) \\
  \quad \text{else if} \ n =? \ 0 \ \text{then} \ \text{trigger} (\text{inl1} (\text{Ackermann} (m-1) 1)) \\
  \quad \text{else} \ (\text{ack} \leftarrow \text{trigger} (\text{inl1} (\text{Ackermann} m (n-1))));; \\
  \quad \text{trigger} (\text{inl1} (\text{Ackermann} (m-1) \text{ack}))).
  \]

- Given the recursive handler and [mrec] combinator, we can tie the knot

  \[
  \text{Definition} \quad \text{ackermann} : \text{nat} \to \text{nat} \to \text{itree emptyE} \text{ nat} := \\
  \text{fun} \ m \ n \Rightarrow \text{mrec} \ h\_\text{ackermann} (\text{Ackermann} m n).
  \]
Weak Bisimulation

(Coinductive) relation ignoring finite amount of internal steps

*eutt*: “equivalence up-to taus”
### Interaction Tree equational theory (excerpt)

#### Structural laws

- \((\text{Tau } t) \approx t\)
- \((x \leftarrow (\text{Tau } t) ;; k) \approx \text{Tau } (x \leftarrow t ;; k)\)
- \((x \leftarrow (\text{Vis } e \text{ k1}) ;; k2) \approx (\text{Vis } e (\text{fun } y \Rightarrow (k1 \ y) ;; k2))\)

#### Monad laws

- \((x \leftarrow \text{ret } v ;; k \ x) \approx (k \ v)\)
- \((x \leftarrow t ;; \text{ret } x) \approx t\)
- \((x \leftarrow (y \leftarrow s ;; t) ;; u) \approx (y \leftarrow s ;; x \leftarrow t ;; u)\)

#### Iterative laws

- \(\text{iter } f \approx f \triangleright\triangleright\triangleright \text{case}_- (\text{iter } f) \ \text{id}_-\)
- \(\text{iter } f \triangleright\triangleright\triangleright g \approx \text{iter } (f \triangleright\triangleright\triangleright \text{bimap } \text{id}_- \ g)\)

#### Interp laws

- \(\text{interp } h \ (\text{trigger } e) \approx h \_ \ e\)
- \(\text{interp } h \ (\text{Ret } r) \approx \text{ret } r\)
- \(\text{interp } h \ (x \leftarrow t ;; k \ x) \approx x \leftarrow (\text{interp } h \ t) ;; \text{interp } h \ (k \ x)\)
Benefits of Interaction-Tree based reasoning

Reasoning about control-flow

- Proof of block-merging optimization
- Reasoning about composing control-flow operators is simple
- Benefit

Proof involves reasoning only about control flow, not other side-effects (e.g. state, exception..)
VIR

**Level 0**

\[ \text{itree VellvmE } \mathcal{V}_u \]

**Level 1**

\[ \text{stateT}_{Env_G} (\text{itree } E_1) \mathcal{V}_u \]

**Level 3**

\[ \text{stateT}_{Env_L*Env_G} (\text{itree } E_2) \mathcal{V}_u \]

**Level 4**

\[ \text{stateT}_{Mem*Env_L*Env_G} (\text{itree } E_3) \mathcal{V}_u \]

**propositional model**

\[ \text{stateT}_{Mem*Env_L*Env_G} (\text{itree } E_4) \mathcal{V}_u \vdash \mathbb{P} \]

**model undef_\tau**

\[ \text{stateT}_{Mem*Env_L*Env_G} (\text{itree } E_5) \mathcal{V}_u \vdash \mathbb{P} \]

\[ \sim_0 \]

\[ \sim_1 \]

\[ \sim_2 \]

\[ \sim_3 \]

\[ \sim_4 \]

\[ \sim_5 \]

\[ \sim_6 \]

**Block-Fusion (proof)**

**Simulation relations over richer and richer states**

**Block-Fusion (result)**

**Set inclusion up-to the underlying refinement relation**
The need for a state-aware program logic

Stateful reasoning in VIR

- Relational reasoning on ITree-based semantics
  Two programs \( e_t \) and \( e_s \)
  \( e_t \approx_R e_s \)
  (1) Both terminate and satisfy the postcondition \( R \) over the result of the computation, OR
  (2) Both diverge in simulation with each other

- Stateful Hoare-style reasoning

  Given a stateful interpretation function \( \llbracket - \rrbracket : \text{itree} (E \, '+\, F) \, A \rightarrow \text{stateT} \, S \, (\text{itree} \, F) \, A \)

  \[
  \{ \mathcal{P} \} e_t \approx e_s \{ Q \} := \forall \sigma_t, \sigma_s. \mathcal{P}(\sigma_t, \sigma_s) \Rightarrow \llbracket e_t \rrbracket \sigma_t \approx_Q \llbracket e_s \rrbracket \sigma_s
  \]

- Localize reasoning about state using separation logic
Coq Extraction

Executability
Reference Interpreter: Executability

```ocaml
define i64 @factorial(i64 %n) {
  %1 = alloca i64
  %acc = alloca i64
  store i64 %n, i64* %1
  store i64 1, i64* %acc
  br label %start

  start:
  %2 = load i64, i64* %1
  %3 = icmp sgt i64 %2, 0
  br i1 %3, label %then, label %end

  then:
  %4 = load i64, i64* %acc
  %5 = load i64, i64* %1
  %6 = mul i64 %4, %5
  store i64 %6, i64* %acc
  %7 = load i64, i64* %1
  %8 = sub i64 %7, 1
  store i64 %8, i64* %1
  br label %start

  end:
  %9 = load i64, i64* %acc
  ret i64 %9
}

define i64 @main(i64 %argc, i8** %argv) {
  %1 = alloca i64
  store i64 0, i64* %1
  %2 = call i64 @factorial(i64 5)
  ret i64 %2
}
```

→ Parser  →  Extracted interpreter  →  Tiny OCaml driver to crawl the tree  →  120

Tested against clang over:
- A collection of unit tests
- A handful of significant programs from the HELIX frontend
- Experiments over randomly generated programs using QuickChick

External calls
Debugging messages
Failure
A validated and verified semantics

Reference Interpreter Validation: Executability

- **Debugging**

- **Validate** LLVM Semantics against other implementations
  - test suite of \(~140\) "semantic tests"
  - e.g., integrate with QuickChick, Csmith, ALIVE

- **Find bugs** in the existing LLVM infrastructure
  - thinking hard about corner cases while formalizing is a good way to find real bugs
  - identify inconsistent assumptions about the LLVM compiler

- **Property-based testing**: QuickChick-based generator to generate interesting, UB-free LLVM IR programs (in-progress work, Chen et al.)
SPIRAL/HELIX

[Püschel, et al. 2005] [Franchetti et al., 2005, 2018] [Zaliva et al., 2015 2018, 2019]

DSL for high-performance numerical computing.

- Numerical computations compiled down to LLVM IR
- Formalized in Coq, targets Vellvm
- Bottom of the compilation chain proved* w.r.t. this technique

* Some operators are currently not proved
The Vellvm Project, Revamped

https://github.com/vellvm/vellvm

Selected publications and drafts*

[Zakowski et al. - ICFP 2021]
Modular and executable semantics for LLVM IR

[Yoon et al. - ICFP 2022]
Meta-theory for layered monadic interpreters

[Zaliva et al.]
Verified HELIX front-end

[Beck et al.]
Infinite/finite memory model for LLVM IR

[Yoon et al.]
Relational separation logic for LLVM IR

* : all results mechanized in the Coq Proof Assistant