Climbing Infinite Trees

a VeLLVM/ASM Tutorial

Eric Bond

December 13, 2023

1 Introduction

In this tutorial, I will demonstrate how to prove that a series of assembly optimizations preserve the semantics of a concrete assembly program. To do this, I will use the Coq[coq] theorem prover and the VeLLVM[ZBY$^+$21] framework based on Interaction Trees(abv. itrees)[XZH$^+$19]. We will begin in Section 2, where the model assembly language (abv. asm) and itrees will be introduced along with some simple examples. Section 3 will demonstrate the equational reasoning principles of itrees via a proof that two basic blocks are syntactically bisimilar. We will see that syntactic bisimilarity of assembly programs is limited and introduce semantic bisimilarity in Section 4. At that point, we will have enough tools to demonstrate the equivalence of an assembly program after multiple transformations including dead branch elimination, block fusion, constant propagation, and constant folding Section 5. All the code for this tutorial can be found here.

2 Preliminaries

We will begin with the definition of a model assembly language followed by the definition of itrees.

2.1 Asm

The definition of Asm presented in Figure 1 is taken almost directly from the Itrees tutorial here.

```
(* type of memory addresses, registers,
and values this language can manipulate *)
Definition addr : Set := string.
Definition reg : Set := nat.
Definition value : Set := nat.

(* constants and registers are operands. *)
Variant operand : Set :=
| Oimm (_ : value)
| Oreg (_ : reg).

(* instructions (not all displayed) *)
Variant instr : Set :=
| Imov   (dest : reg) (src : operand)
| Iadd   (dest : reg) (src : reg) (o : operand)
| Iload  (dest : reg) (addr : addr)
...
(** both direct and conditional jumps *)
Variant branch {label : Type} : Type :=
| Bjmp (_ : label)                (* jump to label *)
| Bbrz (_ : reg) (yes no : label) (* conditional jump *)
| Bhalt.                          (* used to represent termination *)

(* a block is a sequence of instructions followed by a branch *)
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))
}.
```

Figure 1: Asm Definition

This model assembly language is quite simple when contrasted with the model of LLVM in Vellvm seen here. There is nothing particularly surprising about this definition except for the definition of `asm (A B : nat)`. Block labels in an `asm` program categorized as `internal` or `external` labels. `Internal` represent the number of internal labels used to link up basic blocks within an `asm` program which are not exposed to other `asm` programs. The number `A` in `asm A B` represent the number of blocks have an entry branch into the `asm` program while `B` represent the number of blocks that have an exit branch out of the `asm` program. The interaction trees tutorial provides basic combinators to build up assembly programs including: `seq_asm` to sequence two assembly programs, `app_asm` to juxtapose two blocks next to eachother, and `loop_asm` to form loops. As an example, Figure 2 shows a simple assembly program and Figure 3 shows the code to describe this program. We will come back to this program in Section 5.

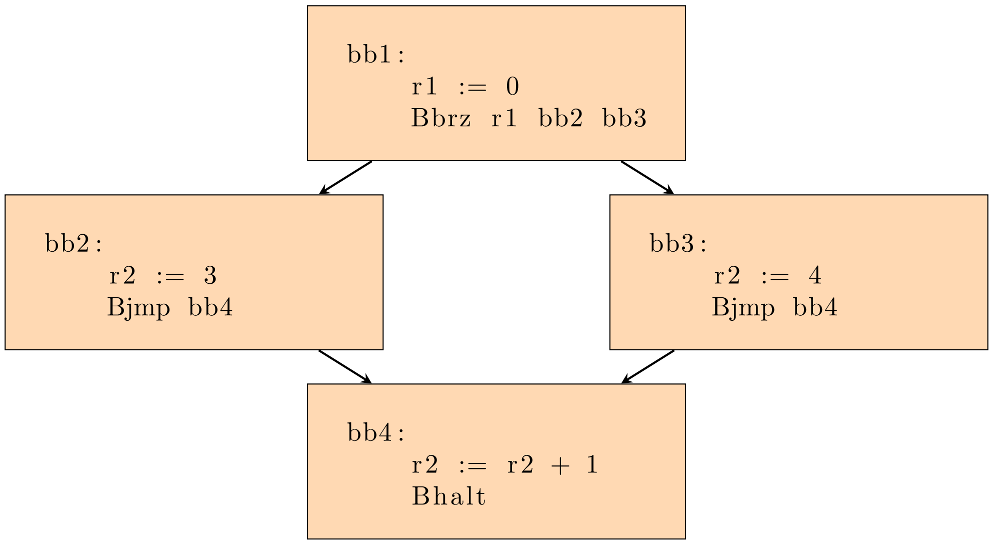

Figure 2: Asm program 1

```
(* definition of the basic blocks of program 1 *)
Definition bb1 : block (fin 2):=
after [
    Imov 1 (Oimm 0)
] (Bbrz 1 f0 (fS f0)).

Definition bb2 : block (fin 1) :=
after [
    Imov 2 (Oimm 3)
] (Bjmp f0).

Definition bb3 : block (fin 1) :=
after [
    Imov 2 (Oimm 4)
] (Bjmp f0).

Definition bb4 : block (fin 1) :=
after [
    Iadd 2 2 (Oimm 1)
] (Bhalt).

(* block -> asm *)
Definition a_bb1 : asm 1 2 := raw_asm_block bb1.
Definition a_bb2 : asm 1 1 := raw_asm_block bb2.
Definition a_bb3 : asm 1 1 := raw_asm_block bb3.
Definition a_bb4 : asm 2 1 := raw_asm (fun _ => bb4).

(* use asm combinators to link blocks together *)
Definition middle : asm (1 + 1) (1 + 1)
    := app_asm a_bb3 a_bb2. (* tensor product *)
Definition bottom : asm (1 + 1) 1
    := seq_asm middle a_bb4. (* loop combinator + renaming *)
Definition prog1 : asm 1 1
    := seq_asm a_bb1 bottom. (* loop combinator + renaming *)
```

Figure 3: Code for program 1

2.2 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).
```

Figure 4: itree definition

An itree is a potentially infinite tree with three types of nodes and is parameterized by two types; `E` represent the type of effects this tree supports and `R` is the return type of the computation.

- A `Ret` node, which stands for return, is a leaf holding a value `r` of type `R`.
- A `Tau` node, empty node which has one successor, is used to represent computation which has no "visible" effect on the environment.
- A `Vis` node, or visual node, is a node with an effect `e` and a continuation `k` which determines the successors of this node.

To give some intuition, Figure 6 demonstrates four simple itrees. For these examples, our effect type `E` will be a simple `IO` effect representing the ability to read (`In`) and write(`out`) from a terminal.

- `boring : itree IO nat` $\triangleq$ a program with just returns the natrual number `42`.
- `spin : itree IO nat` $\triangleq$ a program which spins forever without acting on its environment.
- `echo : itree IO void` $\triangleq$ a program which prompts the user for input, prints the input, and repeats forever.
- `kill9 : itree IO string` $\triangleq$ a program which prompts the user for input, only halting if the input is `"9"`.

```
(* a simple IO effect *)
Inductive IO : Type -> Type :=
| Input : IO string
| Output : string -> IO unit.

(* constant *)
CoFixpoint boring : itree IO nat := Ret 42.

(* spins forever *)
CoFixpoint spin : itree IO nat := Tau spin.

(* prompt for input, print input, repeat forever *)
CoFixpoint echo : itree IO void
  := Vis (Input)
      (fun (str : string) =>
        Vis (Output str)
          (fun (_ : unit) => echo)).

(* prompt for input until "9" is given,
    in which case terminate *)
CoFixpoint kill9 : itree IO string
  := Vis (Input)
    (fun (str : string) =>
      if (str =? "9")
      then (Ret "done")
      else kill9).
```

Figure 5: Coq code for example programs

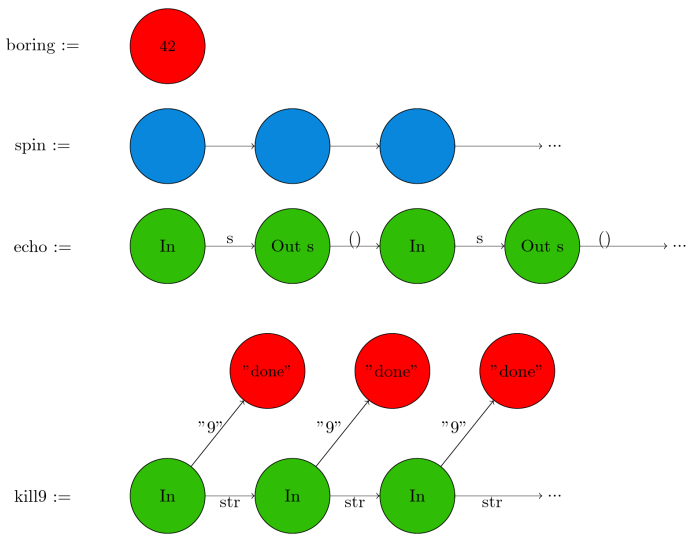

Figure 6: Example itree programs

Interaction trees may appear deceptively simple, but they are able to capture many notions of computation. The main reason we are interested in interaction trees is that they have a rich equational theory (which will shown in Section 3) to prove when two interaction trees are bisimilar. Bisimulation is a way to define when two systems "behave the same" relative to an external observer and independent of their internal structure. This is a powerful reasoning principle that can be used to express equivalence between different languages. For instance, if we can denote programs of a simple imperative programming language and an assembly language as interaction trees, we can reason about when both programs are bisimilar, or behave the same.

2.3 Asm Programs as Itrees

In order to reason about bisimilarity of `asm` programs, we must first provide a map from `asm` programs to `itrees`. This mapping is partially demonstrated by Figure 7 and the code can be found in the IteractionTrees repostitory here. The transformation is relatively obvious once you are familiar with the language of itrees. For example, the denotation of add instruction `Iadd d l r` with destination register `d`, left operand (register) `l`, and right operand `r` is the denotation of each of the operands which results in natural numbers `lv` and `rv` respectively which are added together and mapped to the destination register by `trigger (SetReg d (lv + lr))`. The full denotation function is constructed piece wise from each `asm` language component (`block`, `instr`, etc..).

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

Definition denote_instr (i : instr) : itree E unit :=
     match i with
     | 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 E 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 E B := ...

Definition denote_bks {A B : nat} (bs: bks A B): sub (ktree E) fin A B :=

Definition denote_asm {A B} : asm A B -> sub (ktree E) fin A B :=
```

Figure 7: Denote asm as itree

3 Syntactic Bisimilarity

We can see the equational reasoning of `itrees` in action with a simple example: proving that two exactly equal basic blocks are bisimilar. The code for this demo is in the VeLLVM-Exploration repository here.

```
Definition bb0 : block (fin 1) :=
    after [
        Iadd 1 1 (Oimm 1)
    ] (Bjmp f0).

Definition bb1 : block (fin 1) :=
    after [
        Iadd 1 1 (Oimm 1)
    ] (Bjmp f0).
```

Figure 8: Coq definition of basic blocks

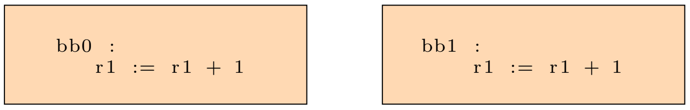

Figure 9: exactly equal basic blocks

```
lv <- trigger (GetReg 1);;
rv <- Ret 1;;
trigger (SetReg 1 (lv + rv)));;
Ret f0
```

Figure 10: Denotation of BB0 and BB1

Since these basic blocks are exactly equal, their denotations are exactly equal by congruence. But for demonstrative purposes, we are going to manually prove their bismiliarity with equational rules provided by the interaction trees library:

- `eqit_Ret`: Says that two `Ret` nodes `(Ret r1),(Ret r2)` are bisimilar when their return values are related by a relation `Rel`.
- `eqit_Vis`: Says that two `Vis` nodes `(Vis e k1),(Vis e k2)` are bisimilar when they carry the same effect `e` and that their continuations `k1,k2` are bisimilar on all possible arguments.
- `eutt_clo_bind`: says that we can peel off the first expression in an `itree` program when the expressions are bisimilar (`t1` $\approx$ `t2`) and the remaining programs are bisimilar (`(k1 u1)` $\approx$ `(k2 u2)`) under all related inputs (`u1 Rel u2`).

$$\texttt{eqit\_Ret}\ \frac{\texttt{r1 Rel r2}}{\texttt{(Ret r1)} \approx \texttt{(Ret r2)}}$$

$$\texttt{eqit\_Vis}\ \frac{\texttt{forall u, (k1 u)} \approx \texttt{(k2 u)}}{\texttt{(Vis e k1)} \approx \texttt{(Vis e k2)}}$$

$$\texttt{eutt\_clo\_bind}\ \frac{\texttt{forall u1 u2, u1 Rel u2 -> (k1 u1)} \approx \texttt{(k2 u2)} \qquad \texttt{t1} \approx \texttt{t2}}{\texttt{(x <- t1;; k1 x)} \approx \texttt{(x <- t2;; k2 x)}}$$

Figure 11: Some equations for reasoning about bisimilarity

Using these rules, we can prove that the denotations of `bb0` and `bb1` are bisimilar as seen in Figure 12. The base of the proof tree begins with a statement that the denotations of `bb0` and `bb1` are bisimilar. The proof proceeds by peeling off the first expression of each itree program by applying the rules in Figure 11. For syntactic bisimilarity, the relation `Rel` we are using here is definitional equality (`eq`) which will not suffice for semantic bisimilarity; we will discuss this in the next section. The code for this proof is here.

$$\texttt{eutt\_clo\_bind}\ \frac{\texttt{eqit\_Vis}\ \dfrac{\texttt{eqit\_Ret}\ \dfrac{\texttt{reflexivity}\ \dfrac{}{\texttt{u = u}}}{\texttt{forall u : nat, (Ret u)} \approx \texttt{(Ret u)}}}{\texttt{vis (GetReg 1) (fun x : nat => Ret x)} \approx \texttt{vis (GetReg 1) (fun x : nat => Ret x)}} \qquad \dfrac{\texttt{eqit\_Ret}\ \dfrac{\texttt{reflexivity}\ \dfrac{}{\texttt{1 = 1}}}{\texttt{(Ret 1)} \approx \texttt{(Ret 1)}} \quad \ldots}{\begin{array}{l}\texttt{r0 <- Ret 1;;}\\ \texttt{trigger (SetReg 1 (r + r0));;}\\ \texttt{Ret f0}\end{array} \approx \begin{array}{l}\texttt{r0 <- Ret 1;;}\\ \texttt{trigger (SetReg 1 (r + r0));;}\\ \texttt{Ret f0}\end{array}}}{\begin{array}{l}\texttt{r <- trigger (GetReg 1);;}\\ \texttt{r0 <- Ret 1;;}\\ \texttt{trigger (SetReg 1 (r + r0));;}\\ \texttt{Ret f0}\end{array} \approx \begin{array}{l}\texttt{r <- trigger (GetReg 1);;}\\ \texttt{r0 <- Ret 1;;}\\ \texttt{trigger (SetReg 1 (r + r0));;}\\ \texttt{Ret f0}\end{array}}$$

Figure 12: Partial proof that BB0 $\approx$ BB1

4 Semantic Bisimilarity

Syntactic bisimilarity can be used to prove the correctness of some *structural* transformations on LLVM and `asm` programs(like block fusion seen here) but, as we will see shortly, it does have limitations. Let's try to prove syntactic bisimilarity of two instructions: `i1 : Iadd r3 r1 r2` and `i2 : Iadd r3 r2 r1`.

$$\texttt{eutt\_clo\_bind}\ \frac{\textit{eqit\_Vis}\ \dfrac{(\textit{GetReg}\ 1) \neq (\textit{GetReg}\ 2)}{\texttt{Vis (GetReg 1) (fun x : nat => Ret x)} \approx \texttt{Vis (GetReg 2) (fun x : nat => Ret x)}} \qquad \ldots}{\begin{array}{l}\texttt{lv <- trigger (GetReg 1);;}\\ \texttt{rv <- trigger (GetReg 2);;}\\ \texttt{trigger (SetReg 3 (lv + rv))}\end{array} \approx \begin{array}{l}\texttt{lv <- trigger (GetReg 2);;}\\ \texttt{rv <- trigger (GetReg 1);;}\\ \texttt{trigger (SetReg 3 (lv + rv))}\end{array}}$$

Figure 13: Failed proof that i1 $\approx$ i2

This proof fails because the `eqit_Vis` rule requires that the effects `e` in `(Vis e k)` be equal. But we have[1] `(Vis (GetReg 1) ..)` ≈ `(Vis (GetReg 2) ...)` as a subgoal. The solution to this dilemma requires two components; 1) provide an interpretation of the effects in `asm` (GetReg, SetReg, Load,..) and 2) enhance the relation `Rel`.

4.1 Interpreting Effects

To prove `i1` and `i2` are bisimilar, we need to say what effects like `GetReg,SetReg,..` *mean.* In order to do that, we need to model memory and registers. This simplest model of memory is a map from addresses to values and for registers, a map from registers to values. This simplistic model will be good enough for this tutorial but, as stated in the VeLLVM paper [ZBY$^+$21], having an accurate memory model is challenging and necessary to prove certain LLVM optimizations. If we model memory/registers as maps from addresses/registers to values, then `GetReg`/`Load` is just a map lookup and `SetReg`/`Store` is just an insertion into the map. This interpretation is represented by `h_reg` and `h_memory` respectively and the code can be found here.

```
Definition registers := alist reg value.
Definition memory    := alist addr value.

Definition h_reg {E: Type -> Type} `{mapE reg 0 -< E}
  : Reg ~> itree E :=
  fun _ e =>
    match e with
    | GetReg x => lookup_def x
    | SetReg x v => insert x v
    end.

Definition h_memory {E : Type -> Type} `{mapE addr 0 -< E} :
  Memory ~> itree E :=
  fun _ e =>
    match e with
    | Load x => lookup_def x
    | Store x v => insert x v
    end.

(** The _asm_ interpreter takes as inputs a starting heap [mem] and register
    state [reg] and interprets an itree in two nested instances of the [map]
    variant of the state monad.
*)
Definition interp_asm {E A} (t : itree (Reg +' Memory +' E) A) :
  memory -> registers -> itree E (memory * (registers * A)) :=
  let h := bimap h_reg (bimap h_memory (id_ _)) in
  let t' := interp h t in
  fun  mem regs => interp_map (interp_map t' regs) mem.
```

Figure 14: Interpretation of `asm` effects

`interp_asm` converts an `itree` (`t : itree (Reg + Memory + E) A`) with the register (`Reg`) effect, memory (`Memory`) effect, and return type `A` to an `itree` (`itree E (memory * registers * A)`) with effect `E` and return type `(memory * registers * a)`. Next we need to define when `itree`s of type `itree E (memory * registers * A)` are bisimilar.

[1] `(trigger (GetReg r))` desugars to `((Vis (GetReg r) (fun (x : nat) => Ret x ))`

4.2 Enhance `Rel`

Relations are built up compositionaly. The code can be found here

- `eq_map` : two maps are equal when they return the same value for all possible keys.[2]
- `EQ_registers`/`EQ_memory` : Both are instance of `eq_maps` fixed to their respective types.
- `rel_asm` : describes what it means for two elements `a,b` of type `memory * registers * B` to be related. For this we take the product of relations `EQ_memory,EQ_registers,eq` where `eq` is definitional equality.
- `EQ_asm` : describes what it means for two itrees `t1,t2` of type `itree (Reg + Memory + E) A` to be related. We say that `t1` $\approx$ `t2` when their results are related by `rel_asm` and they "execute" in a setting where their memory states are related (by `EQ_memory`) and register content is related (by `EQ_registers`).

```
Definition eq_map (m1 m2 : map) : Prop :=
  forall k, lookup k m1 = lookup k m2.

Definition EQ_registers (regs1 regs2 : registers) : Prop :=
  eq_map regs1 regs2.

Definition EQ_memory (mem1 mem2 : memory) : Prop :=
  eq_map mem1 mem2.

Definition rel_asm {B} : memory * (registers * B) -> memory * (registers * B) -> Prop :=
    EQ_memory ⊗ EQ_registers ⊗ eq.

Definition EQ_asm {E A} (f g : memory -> registers -> itree E (memory * (registers * A))) : Prop :=
  forall mem1 mem2 regs1 regs2,
    EQ_memory mem1 mem2 ->
    EQ_registers regs1 regs2 ->
    eutt rel_asm (f mem1 regs1) (g mem2 regs2).
```

Figure 15: Defining a relation for interpreted `asm` programs

4.3 Bisimilarity of `i1` and `i2`

We now have some additional proof rules to deal with `interp_asm`. We will denote the bisimulation relation `EQ_asm` by $\approxeq$.

$$\text{interp\_asm\_SetReg}\ \frac{}{\texttt{interp\_asm (trigger(SetReg r v);; f) mem reg} \approxeq \texttt{interp\_asm f mem (add r v reg)}}$$

$$\text{interp\_asm\_GetReg}\ \frac{}{\texttt{interp\_asm (v <- trigger(GetReg r);; f v) mem reg} \approxeq \texttt{interp\_asm (f (looup r reg)) mem reg}}$$

Figure 16: New rules for interpreted effects

```
Variable x : nat.
Variable y : nat.
Variable mem : memory.

Definition startReg : registers :=
[
    (1,x);
    (2,y)
].
```

Figure 17: Starting memory/register configuration

[2]This is a restrictive notion of map equality, we could relax this to say that each map has the same values, but we won't do that here

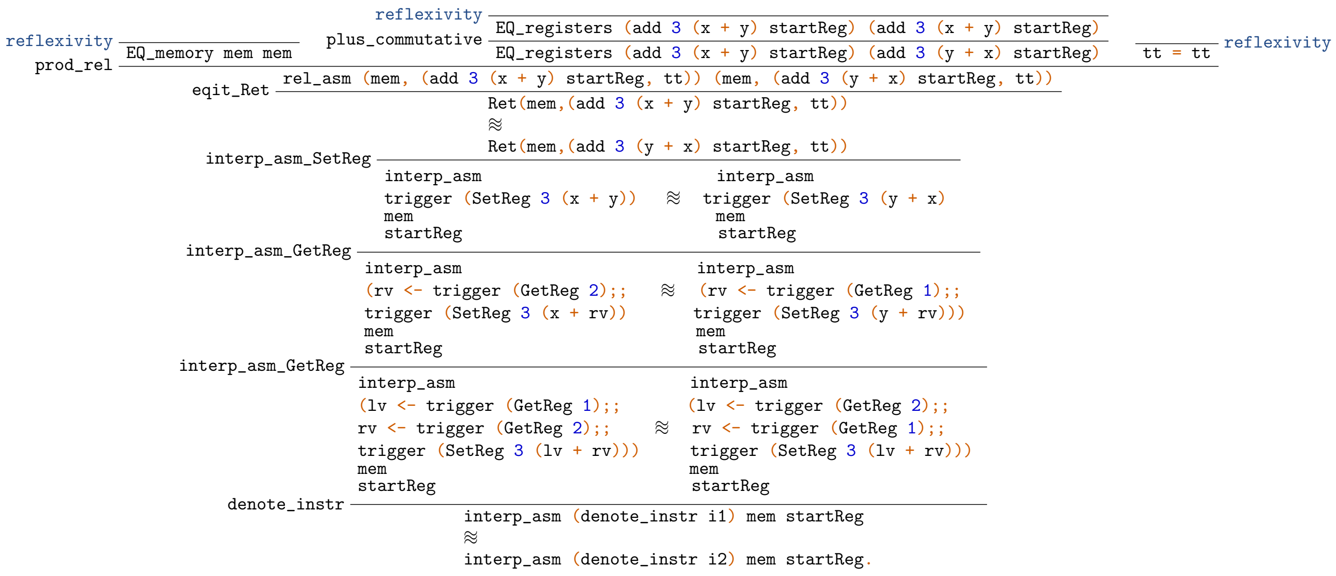
reflexivity
EQ_registers (add 3 (x + y) startReg) (add 3 (x + y) startReg)
reflexivity
plus_commutative
EQ_registers (add 3 (x + y) startReg) (add 3 (y + x) startReg)
reflexivity
tt = tt
EQ_memory mem mem
prod_rel
rel_asm (mem, (add 3 (x + y) startReg, tt)) (mem, (add 3 (y + x) startReg, tt))
eqit_Ret
Ret(mem,(add 3 (x + y) startReg, tt)) ≈ Ret(mem,(add 3 (y + x) startReg, tt))
interp_asm_SetReg
interp_asm trigger (SetReg 3 (x + y)) mem startReg ≈ interp_asm trigger (SetReg 3 (y + x) mem startReg
interp_asm_GetReg
interp_asm (rv <- trigger (GetReg 2);; trigger (SetReg 3 (x + rv)) mem startReg ≈ interp_asm (rv <- trigger (GetReg 1);; trigger (SetReg 3 (y + rv))) mem startReg
interp_asm_GetReg
interp_asm (lv <- trigger (GetReg 1);; rv <- trigger (GetReg 2);; trigger (SetReg 3 (lv + rv))) mem startReg ≈ interp_asm (lv <- trigger (GetReg 2);; rv <- trigger (GetReg 1);; trigger (SetReg 3 (lv + rv))) mem startReg
denote_instr
interp_asm (denote_instr i1) mem startReg ≈ interp_asm (denote_instr i2) mem startReg.

Figure 18: Successful proof that i1 ≈ i2

The proof begins at the base of Figure 18 by stating that the `asm` interpretation of the denotations of instruction `i1` and `i2` are bisimilar according to `EQ_asm`. We then interpret all the `GetReg` and `SetReg` effects as reads/writes to the `registers` map. At that point, we have a `Ret` node holding a value of type `memory * registers * unit`. Recall that `eqit_Ret` states that `Ret v1` ≈ `Ret v2` when `v1 Rel v2`, that is to say they are related by our bisimilation relation. In this case, our bisimilation relation is `rel_asm` which is a product of three relations: `EQ_memory, EQ_registers,eq`. The first and third case hold by `reflexivity`. The middle case which says `i1` and `i2` performed the same updates to the register map holds after we use commutativity of addition. The code for this proof can be found here.

5 Composing Transformations

At this point, we have enough tools to demonstrate the equivalence of an assembly program after multiple transformations including dead branch elimination, block fusion, constant propagation, and constant folding. We will not go into detail in this section as the proofs get quite long. Instead, the reader should refer to the code which has tutorial syle comments for guidance. Instead, we give a visualization of the transformations on a concrete program and point to where the proofs are in the repository.

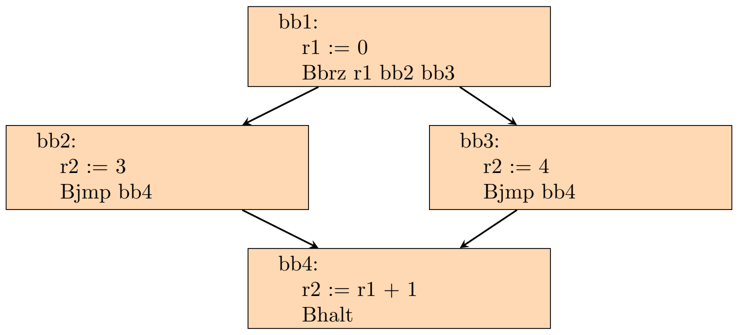

Figure 19: program 1: Initial `asm` program

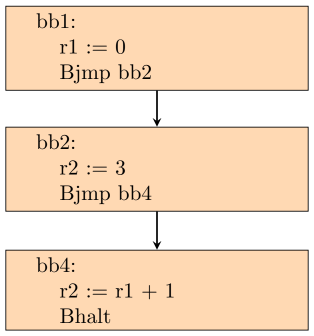

Figure 20: Program 2 : after dead branch elim

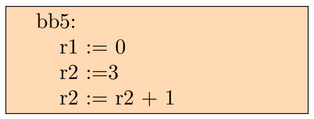

Figure 21: Program 3 : after block fusion

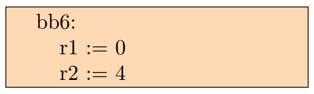

Figure 22: Program 4 : after constant propogation and folding

- `program1` ≈ `program2`: proof
- `program2` ≈ `program3`: proof
- `program3` ≈ `program4`: proof
- by transitivity, `program1` ≈ `program4` proof

These proofs follow the same style as Figure 18. However, there are many additional proof complications; the main one being jumping to new basic blocks. To ease some of the proof burden, I've created custom tactics to make pushing the denotation and interp functions through terms a bit easier.

References

[coq] https://coq.inria.fr/.

[XZH$^+$19] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: Representing recursive and impure programs in coq. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.

[ZBY$^+$21] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. Modular, compositional, and executable formal semantics for llvm ir. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021.