Compositional Compiler Correctness in Coq

Steve Zdancewic
University of Pennsylvania

Collaborators

Paul He
Gil Hur
Gregory Malecha
Benjamin Pierce
Li-yao Xia
Yannick Zackowski

Goal: model (in Coq) *interactive* systems

- web servers
- operating systems
- language semantics

and prove properties about them.

Question: how to do that when Coq is a pure, total language?

Interaction Trees

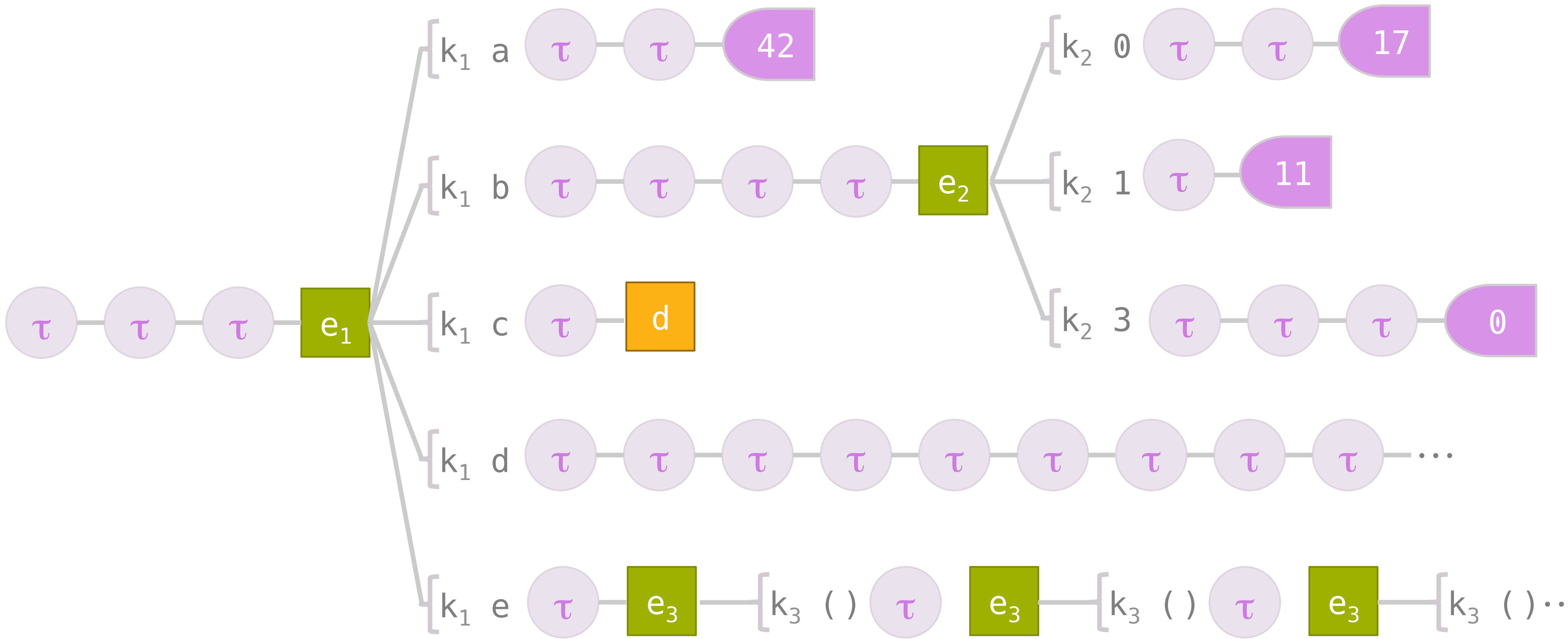
https://github.com/DeepSpec/InteractionTrees

`tutorial/*.v`

(This talk is based on the `AsmOptimization` git branch, which will be merged with master early next week ☺)

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

See also:
- Capretta's "Delay" Monad,
- Kiselyov & Ishii "Freer" monad,
- Hancock & Setzer – line of work, including Agda implementation
- McBride
- Plotkin & Power and much other work: Algebraic Effects



τ τ τ e1
k1 a τ τ 42
k1 b τ τ τ τ e2
k2 0 τ τ 17
k2 1 τ 11
k2 3 τ τ τ 0
k1 c τ d
k1 d τ τ τ τ τ τ τ τ τ ···
k1 e τ e3 k3 ( ) τ e3 k3 ( ) τ e3 k3 ( )···

Good Qualities of Interaction Trees

- `(ITree E)` is a monad
  - `bind` is defined coinductively (it grafts on subtrees)
- Extractable from Coq
  - yields a way of (externally) running computations described by interaction trees
  - interpretation of events can be defined in the metalanguage (e.g. Ocaml)
- Behavioral Equivalences
  - strong bisimulation
  - weak bisimulation (insert a finite no. of Tau's anywhere)
  - rich equational theory

Quite intricate coinductive proofs needed here…
… but, they're encapsulated in the library.

ITree Interface

```
Context {E M : Type → Type} `{Functor M} `{Monad M} `{ALoop M}.
```

Operations

```
trigger : ∀ T, E T → itree E T.
interp  : (∀ T : Type, E T → M T) → (∀ T : Type, itree E T → M T).
loop    : ∀ I A B, (I + A → M (I + B)) → A → M B.
```

Equivalence

$t_1 \cong t_2$ - strong bisimulation

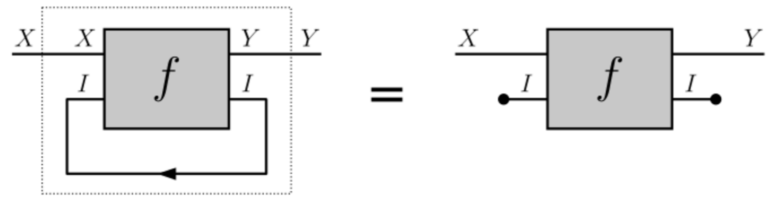
$t_1 \approx t_2$ - weak bisimulation

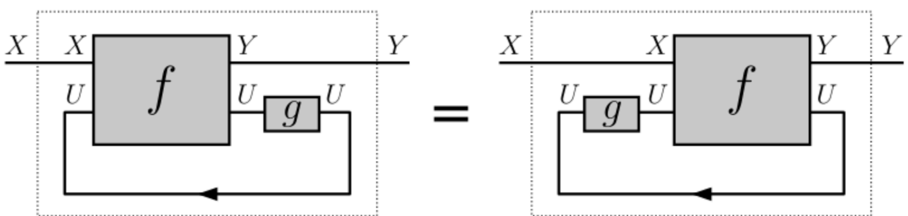
Equations

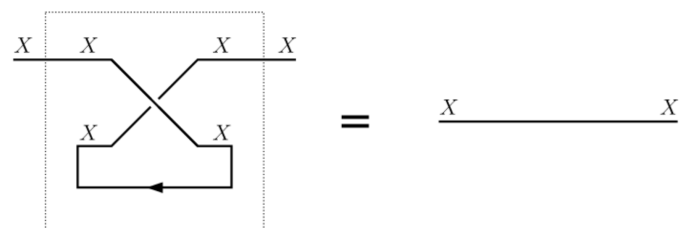
```
Tau t ≈ t
interp h (ret r) ≅ ret r
interp h (x ← t;; k x) ≅ x ← (interp h t);; interp h (k x)
interp h (Vis e k) ≈ (x ← (h e);; interp h (k x))
```

Loop Equivalences*

```
  (loop body a) >>= f
≅ loop (λ ca ⇒
          cb ← body ca ;;
          match cb with
          | inl c ⇒ Ret (inl c)
          | inr b ⇒ ITree.map inr (f b)
          end) a.
```







* Traced Monoidal Categories, a.k.a. Arrows with loops

State Interpreter Equivalences

```
interp_state (ret x) s ≈ ret (s, x)
interp_state (x ← t;; k x) s ≈ '(news, x) ← interp_state t s;; interp_state (k x) news
interp_state (Vis getE k) s ≈ interp_state (k s) s
interp_state (Vis (putE s') k) s ≈ interp_state (k tt) s'

interp_state (x ← get ;; y ← get ;; k y x) s ≈ interp_state (x ← get ;; k x x) s
interp_state (put s1 ;; put s2 ;; k) s ≈ interp_state (put s2 ;; k) s
```

ITrees Library Features

- ITree monad
- parameterized equivalences

  ≅ = `eq_itree eq` `eq_itree R`

  ≈ = `eutt eq` `eutt R`
- KTrees "continuation trees"
  - Coq functions of type: `A → itree E B`
  - Supports looping but not recursion
- Interpreters
  - state, environment, choice, loops, etc.
- Typeclasses
  - for "subevent" declarations e.g. `StateE -< E`

Verifying a (simple) Compiler

Strategy:

- use denotational semantics for source and target languages
- build bisimulation compositionally (by induction on syntax)
- all key proof steps: rewriting via equational reasoning

Benefits:

- not an operational semantics (e.g. no program counter, etc.)
- no (explicit) codinduction
- proof factors into two parts
  - control-flow, reasoning about CFG composition (compiler independent)
  - language-specific correctness of individual instructions
- modular & robust of proofs (?)

COQ CODE

Itrees Library Problems & Challenges

- Dealing with many effects: `E1 +' E2 +' … +' En`
  - nested interpreters: `stateT S1 (stateT S2 M) X`
  - typeclass machinery is brittle: (`E -< F`)
  - writing generic lemmas, not so easy

  ⇒ ρ-polymorphism, more explicit inclusion witnesses, … ??
- Working modulo equivalences
  - (a.k.a. "setoid hell")
  - typeclasses, instances of Proper
  - ??
- More general equational theory for ALoop
  - mrec
- Coinductive definitions in Coq don't simplify
  - have to use tactics or "fuel" to

Verified Compiler Challenges

- Current Asm representation doesn't easily facilitate certain optimizations
  - need a bit more structure on labels

- Scaling up to more language features
  - Vellvm branch uses such denotational semantic
  - this representation is significantly simpler
  - still extractable as interpreter
  - … not many proofs yet
- Higher-order functions?

Conclusions

ITrees provide a useful way to represent effectful/non-terminating computations in Coq.

- Easy to program with
- Supports extraction
- Rich equational theory

https://github.com/DeepSpec/InteractionTrees

`tutorial/*.v`

This talk is based on the `AsmOptimization` git branch.

Recursion & Loops

ITrees support general mutual recursion

- no guardedness / termination requirements!

```
Polymorphic Class ALoop (M : Type → Type) : Type :=
  aloop : ∀ {R I: Type}, (I → M I + R) → I → M R.
```

- using aloop, one can define recursion, loop combinators
- whole family of structures that support loops

```
Definition _aloop {E : Type → Type} {R I : Type}
           (tau : _)
           (aloop_ : I → itree E R)
           (step_i : itree E I + R) : itree E R :=
  match step_i with
  | inl cont ⇒ tau (ITree.bind cont aloop_)
  | inr r ⇒ Ret r
  end.

Definition aloop {E : Type → Type} {R I: Type}
           (step : I → itree E I + R) : I → itree E R :=
  cofix aloop_ i := _aloop (λ t ⇒ Tau t) aloop_ (step i).
```

Imp Denotational Semantics

```
Variant ImpState : Type → Type :=
| GetVar (x : var) : ImpState value
| SetVar (x : var) (v : value) : ImpState unit.

Context {eff : Type → Type}.
Context {HasImpState : ImpState -< eff}.

Fixpoint denoteExpr (e : expr) : itree eff value :=
  match e with
  | Var v     ⇒ trigger (GetVar v)
  | Lit n     ⇒ ret n
  | Plus a b  ⇒ l ← denoteExpr a ;; r ← denoteExpr b ;; ret (l + r)
  | Minus a b ⇒ l ← denoteExpr a ;; r ← denoteExpr b ;; ret (l - r)
  | Mult a b  ⇒ l ← denoteExpr a ;; r ← denoteExpr b ;; ret (l * r)
  end.
```

```
Definition while {eff} (t : itree eff 𝔹) : itree eff unit :=
  loop
    (λ l : unit + unit ⇒
       match l with
       | inr _ ⇒ ret (inl tt)
       | inl _ ⇒ continue ← t ;;
                 if continue : 𝔹 then ret (inl tt) else ret (inr tt)
       end) tt.
```

```
Fixpoint denoteStmt (s : stmt) : itree eff unit :=
  match s with
  | Assign x e ⇒  v ← denoteExpr e ;; trigger (SetVar x v)
  | Seq a b    ⇒  denoteStmt a ;; denoteStmt b
  | If i t e   ⇒
    v ← denoteExpr i ;;
    if is_true v then denoteStmt t else denoteStmt e

  | While t b ⇒
    while (v ← denoteExpr t ;;
           if is_true v
           then denoteStmt b ;; ret true
           else ret false)

  | Skip ⇒ ret tt
  end.
```