Irene Yoon
University of Pennsylvania

Through the Interaction Forest: *Modeling Concurrency in Coq*

Advised by
Benjamin Pierce,
Steve Zdancewic

*How can we **verify** **concurrent programs**?*

In the presence of concurrency, programs must be viewed as components that interact with their environment. This intuition has developed into *bisimulation*, an observational equational theory for reasoning about concurrent programs.

Can we extend this theory and bring **mechanical verification** of concurrent programs? To start, we need to inch towards a *verifiable representation of concurrent models*. We present an encoding of Milner's Calculus of Communicating Systems, a basic calculus for synchronous handshakes, in Coq using **Interaction Trees**.

1. Proof Framework

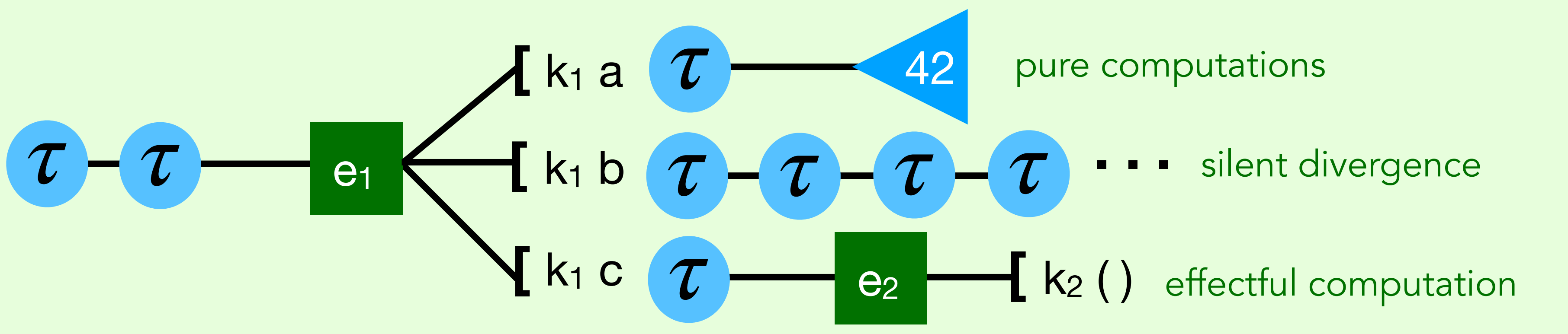
Interaction Trees[1] (*ITrees*)

[1] ***Interaction Trees: Representing Effectful and Recursive Programs in Coq.*** Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, Steve Zdancewic. POPL 2020.

General-purpose data structure representing **recursive** and **impure** programs in Coq.

```
CoInductive itree (E: Type → Type) (R: Type): Type :=
| Ret (r: R)                                  (* computation terminating with value r *)
| Tau (t: itree E R)                          (* "silent" tau transition with child t *)
| Vis {A: Type} (e: E A) (k: A → itree E R). (* visible event e yielding an answer in A *)
```

" [View] computations as a sequence of *visible events* — **interactions** — each of which might carry a response from the environment back to the computation. "



Proof Powertool!

1. Free Monadic Structure →	**modular reasoning**
2. Coinduction →	models **recursion**
3. Rich Equational Theory →	**easy** client-side **proofs**
4. Coq Extraction →	**executable**

See Also:

- Partial Functions in Type Theory: Capretta's "Delay" Monad
- Composable Effects: Kiselyov & Ishii "Freer" Monad
- Effectful Computations in Type Theory: Hancock, McBride's general monad
- Algebraic Effects: Plotkin & Power

2. Concurrency Model

Milner's Calculus of Communicating Systems

A predecessor π-*calculus*, CCS is a basic calculus for synchronous handshakes. The primitive in the calculus is a *process* that can have *ports* that processes can communicate through.

$P := \varnothing$	Empty Process
$\mid\ \alpha\,.\,P$	Action
$\mid\ P_1 \mid P_2$	Parallel Composition
$\mid\ P_1 + P_2$	Choice
$\mid\ \nu\,.\,P$	Restriction *(Hide)*
$\mid\ !P$	Process Generation *(Bang)*

As an example, here is a simple process:

$A \equiv a\,.\,A'$ can handle an **input** action on port a and continue to A'.

$A' \equiv \bar{b}\,.\,A$ can handle an **output** action on port b and continue to A.

Processes can only communicate through a port with the same label with opposing polarity.

Given $B \equiv b\,.\,B'$ and $B' \equiv \bar{c}\,.\,B'$

$A' \mid B \xrightarrow{\tau} A \mid B'$ a synchronous communication between A' and B can occur.

3. Denoting Model in Proof Framework

Denotation of ITrees with CCS

[2] ***A term model for CCS.*** Hennessy M.C.B., Plotkin G.D. (1980) In: Dembiński P. (eds) Mathematical Foundations of Computer Science 1980. MFCS 1980. Lecture Notes in Computer Science, vol 88. Springer, Berlin, Heidelberg

We provide a denotation of CCS based on Hennessy and Plotkin's model of CCS [2].

The trickiest bit is the *parallel composition* operator: how can we denote the nondeterministic choices that occur when processes are composed in parallel?

ITree Representation

```
Definition ccs := itree ccsE unit.
```

Variable Naming and Scope

We use *locally nameless terms* for actions (`Label`), which is labelled on whether it is an input or output action.

```
Variant Label: Type :=
| In (l: idx)
| Out (l: idx).
```

Events

The uninterpreted events are: non-deterministic choice, action, and synchronous communication.

```
Variant ccsE {A : Type}: Type → Type :=
| Or  (n: nat): ccsE nat
| Act: Label → ccsE unit
| Sync: idx → ccsE unit.
```

Nondeterministic choice is represented by the event `Or`, which indexes the possible set of choices.

```
(* Action operators. *)
Definition send (l : A) (k : ccs) :=
    Vis (Act (In l)) (λ _ ⇒ k).
Definition recv (l : A) (k : ccs) :=
    Vis (Act (Out l)) (λ _ ⇒ k).

(* Synchronous action (τ) operator. *)
Definition sync (l : A) (k : ccs) :=
    Vis (Sync l) (λ _ ⇒ k).
```

Atomic Operators

Nondeterministic operators are ***atomic***. These operators are easy to define with ITrees, as they each have separate event representations. For atomic operators, the continuations do not depend on the interpretation of the event.

Parallel Composition

We write $\sum_i t_i$ for the nondeterministic choice (sum), $t_1 + t_2 + \ldots t_i \ldots$ [2].

A parallel composition can be denoted as the composition of the sum of *atomic operators*, and is defined coinductively over the ITree.

1) Left/Right Reduce

$$(t|_{L/R}u) = \begin{cases} \alpha\,.\,(t'|u) & (t = \alpha\,.\,t') \\ \tau\,.\,(t'|u) & (t = \tau\,.\,t') \\ u & (t = \texttt{Ret}\ x) \\ \texttt{fail} & (\texttt{otherwise}) \end{cases}$$

Reducing either the left or right term is defined symmetrically, where an atomic operation is executed.

2) Communication

$$(t|_C u) = \begin{cases} \tau\,.\,(t'|u') & (t = \alpha\,.\,t', u = \bar{\alpha}\,.\,u') \\ \texttt{fail} & (\texttt{otherwise}) \end{cases}$$

Reducing both term represents a synchronous step. Note that the silent step here (τ) represents a synchronous handshake (`Sync`), which is different from the silently divergent ITree `Tau` nodes.

$$\Big(\sum_i t_i\Big)\Big|\Big(\sum_j u_j\Big) = \sum_i \Big(t_i\Big|_L\Big(\sum_j u_j\Big)\Big) + \sum_j \Big(\Big(\sum_i t_i\Big)\Big|_R u_j\Big) + \Big(\sum_i \sum_j (t_i|_C u_j)\Big)$$

This denotes the possible reduction strategies for the composition. The equational theory in CCS states that any CCS process can be represented as a nondeterministic sum at the top level, which allows us to use this denotation.

4. Verifying Our Encoding

Trace Equivalence

To verify our denotation, we prove an equivalence between the trace of the operational semantics of CCS (the Labeled Transition System(LTS)) and the trace semantics of ITrees. Showing trace equivalence is convenient, especially due to the presence of nondeterminism in our concurrency model.

Trace Semantic Equivalence Theorem

```
Theorem trace_equiv:
  (∀ proc trace, itree_trace proc trace → ∃ proc' trace',
     lts_trace proc' trace' ∧ trace ≡ trace') ∧
  (∀ proc' trace', lts_trace proc' trace' → ∃ proc trace,
     itree_trace proc_trace ∧ trace ≡ trace').
```

Future Work

- Extension of ***weak*** and ***strong bisimulation*** in ITrees.
- Modeling **π-calculus** (message passing) calculus

https://github.com/DeepSpec/InteractionTrees/blob/ccs/examples/DenoteCCS.v