Context-Dependent Effects in Guarded Interaction Trees

Sergei Stepanenko, Emma Nardino, Dan Frumin, Amin Timany, and Lars Birkedal

Wednesday, May 7th 2025 @ ESOP'25 – Hamilton, Ontario, Canada

Outline

Context-Dependent Effects in Guarded Interaction Trees

Outline

Motivation

- Denotational semantics and studying language interactions

Interaction Trees

- Denotational semantics for **first-order** programs with **first-order** effects

Guarded Interaction Trees

- Extends interaction trees to **higher-order** programs and effects

Context-Dependent Effects (This Work)

- Extends guarded interaction trees to **context-dependent effects**, *e.g.*, (delimited) continuations, exceptions

Motivation

Denotational Semantics & Language Interaction

Studying Programs Formally

- Goal: study programming languages and their interactions formally
  - In a proof assistant like Rcoq (formerly Coq)
- We need to define programs' behavior (semantics)
  - Should capture the individual languages *and their interaction*
- Common choices:
  - Operational semantics
  - Denotational semantics

Operational Semantics

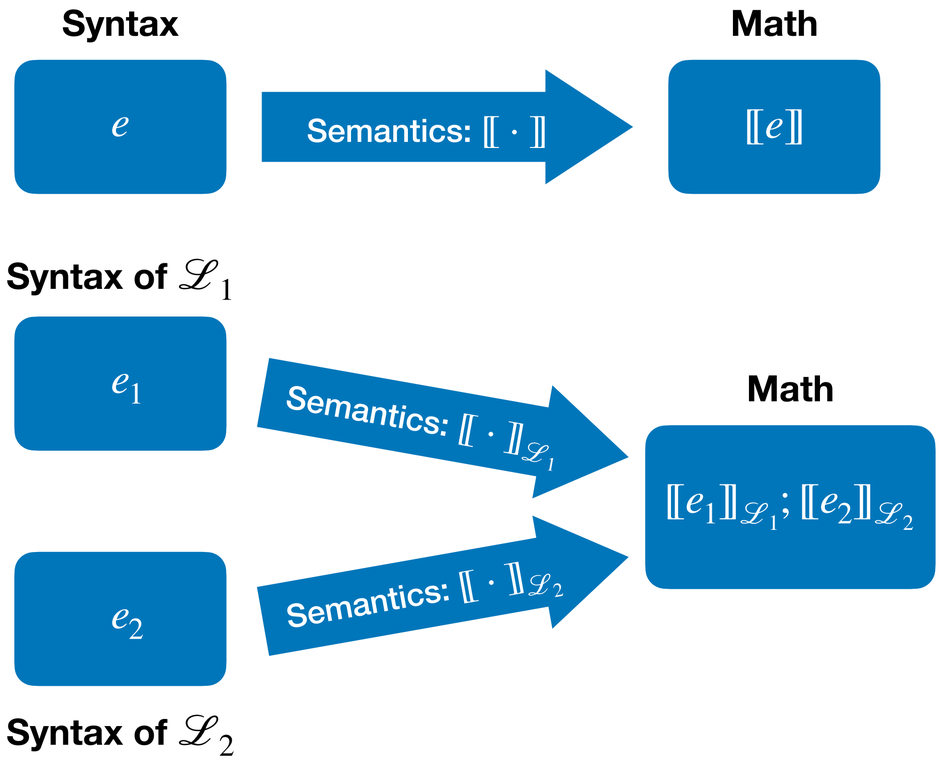
- Describe the mechanics of program execution
- Advantage
  - Simplicity: easy to understand and explain
  - Easily scales to sophisticated languages and features
- Disadvantage
  - Needs to be defined for each language (or combination/interaction there of) we formalize

Denotational Semantics

- Assign **denotations** (mathematical objects) to programs as their semantics
  - Study those mathematical objects
    (also sound w.r.t. operational semantics)

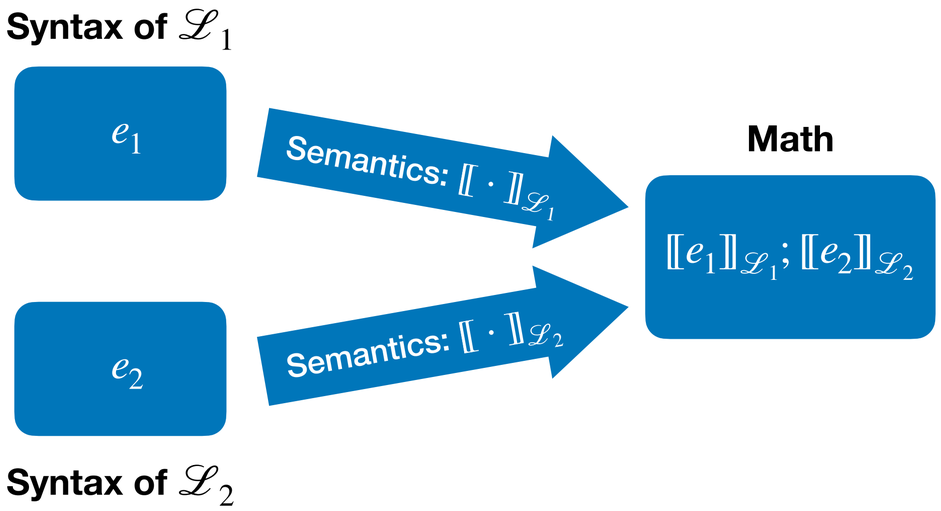
Denotational Semantics

- Assign **denotations** (mathematical objects) to programs as their semantics
  - Study those mathematical objects (also sound w.r.t. operational semantics)
  - Provides a rich common language (math) to combine programs and reason about language interactions



Denotational Semantics

- **Main challenge**:
  - Coming up with a rich enough mathematical universe (**domain**)
  - Ordinary math (Rocq) is not sufficient
    - No non-termination
    - No side effects, *e.g.*, I/O, state
- Literature explores many solutions:
  - Scott domain theory, monads, *etc.*
  - **ITrees** are a recent proposal

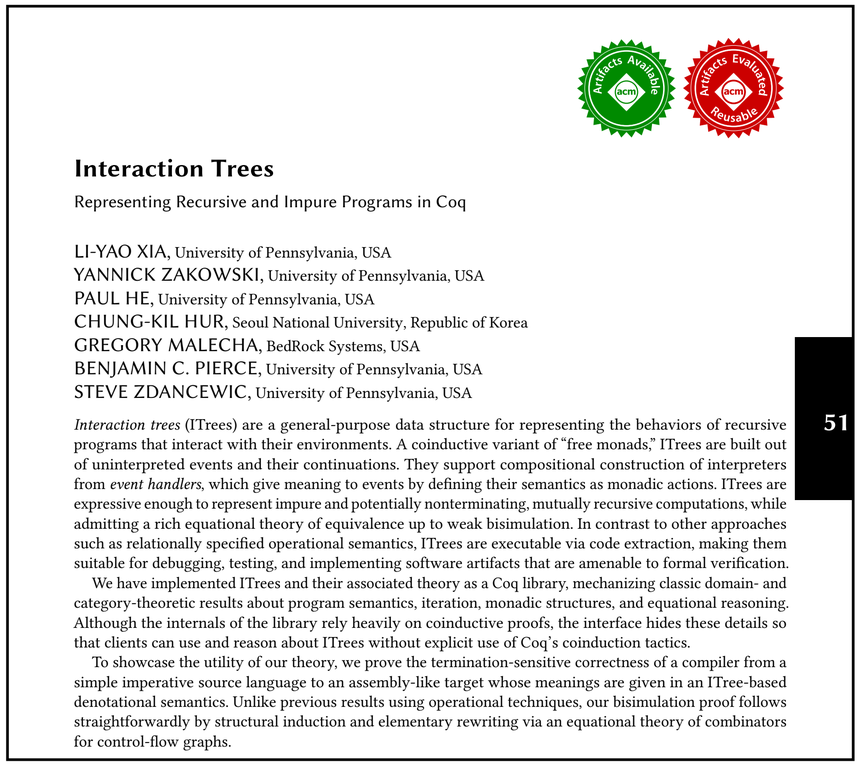


Interaction Trees (ITrees)

Denotational Semantics for First-Order Languages

ITrees

- **Main idea**: construct a domain for
  - Possibly non-terminating programs
  - With side effects
  - In the Rocq prover in which
    - All programs must terminate
    - There are no side effects
- Has since been used for several applications



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

51

Interaction trees (ITrees) are a general-purpose data structure for representing the behaviors of recursive programs that interact with their environments. A coinductive variant of "free monads," ITrees are built out of uninterpreted events and their continuations. They support compositional construction of interpreters from *event handlers*, which give meaning to events by defining their semantics as monadic actions. ITrees are expressive enough to represent impure and potentially nonterminating, mutually recursive computations, while admitting a rich equational theory of equivalence up to weak bisimulation. In contrast to other approaches such as relationally specified operational semantics, ITrees are executable via code extraction, making them suitable for debugging, testing, and implementing software artifacts that are amenable to formal verification.

We have implemented ITrees and their associated theory as a Coq library, mechanizing classic domain- and category-theoretic results about program semantics, iteration, monadic structures, and equational reasoning. Although the internals of the library rely heavily on coinductive proofs, the interface hides these details so that clients can use and reason about ITrees without explicit use of Coq's coinduction tactics.

To showcase the utility of our theory, we prove the termination-sensitive correctness of a compiler from a simple imperative source language to an assembly-like target whose meanings are given in an ITree-based denotational semantics. Unlike previous results using operational techniques, our bisimulation proof follows straightforwardly by structural induction and elementary rewriting via an equational theory of combinators for control-flow graphs.

The original ITrees paper (POPL'20)

ITrees

- A monad defined as a co-inductive type in Rocq

```
CoInductive itree (E : Type -> Type) (R : Type) :=
| Ret : R -> itree E R
| Tau : itree E R -> itree E R
| Vis {A : Type } : E A -> (A -> itree E R ) -> itree E R
```

The continuation

- Constructors embed different kinds of computations:
  - `Ret`: the final value of the computation
  - `Tau`: delay (`Tau c` produces the same result as `c` delayed by a single **step**)
  - `Vis`: an effect with input type `E A` and output type `A`

ITrees are First-Order

- ITrees can embed first-order imperative programs
- Xia *et al.* showed this for `imp`
  - Proved correctness of a compiler from `imp` to assembly
- ITrees do not support languages with higher-order (first class) functions, *e.g.*, OCaml, Haskell
  - How do we translate λ's?

```
(* Imp Syntax ---------------------------------- *)
Inductive expr : Set := ... (* omitted *)

Inductive imp : Set :=
| Assign (x : var) (e : expr)
| Seq    (a b : imp)
| If     (i : expr) (t e : imp)
| While  (t : expr) (b : imp)
| Skip.

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

Taken verbatim from the ITrees paper (Xia *et al.* 2020)

ITrees are First-Order

- **Q**: How do we give a direct translation of λ's?
- In higher-order languages λ's are values, but do not fit in `Ret`
- Ideally, we add a new constructor to embed functions into the type `itree`:

```
CoInductive itree (E : Type -> Type) (R : Type) :=
| Ret : R -> itree E R
| Tau : itree E R -> itree E R
| Vis {A : Type } : E A -> (A -> itree E R ) -> itree E R
```

ITrees are First-Order

- **Q**: How do we give a direct translation of λ's?
- In higher-order languages λ's are values, but do not fit in `Ret`
- Ideally, we add a new constructor to embed functions into the type `itree`:

```
CoInductive itree (E : Type -> Type) (R : Type) :=
| Ret : R -> itree E R
| Fun : (itree E R -> itree E R) -> itree E R
| Tau : itree E R -> itree E R
| Vis {A : Type } : E A -> (A -> itree E R ) -> itree E R
```

Negative occurence

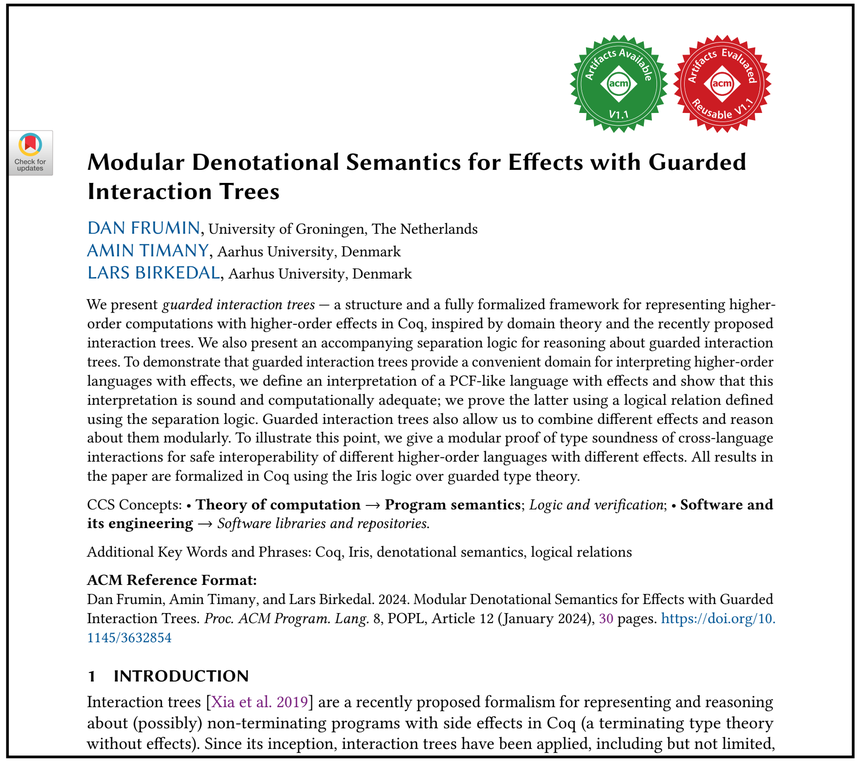
- **Not possible**: the constructor `Fun` is not strictly positive (required by Rocq)
  - Guarded Interaction Trees (GITrees) solve this problem

Guarded Interaction Trees (GITrees)

Denotational Semantics for Higher-Order Languages

GITrees

- **Main idea**: Use guarded type theory
  - Lifts the strict positivity requirement
  - Uses Iris's guarded type theory
- Gave a program logic over GITrees
  - Using Iris's step-indexed program logic
- Studied language interactions
  - Safety of embedding linear types into an affine language



Modular Denotational Semantics for Effects with Guarded Interaction Trees

DAN FRUMIN, University of Groningen, The Netherlands
AMIN TIMANY, Aarhus University, Denmark
LARS BIRKEDAL, Aarhus University, Denmark

We present *guarded interaction trees* — a structure and a fully formalized framework for representing higher-order computations with higher-order effects in Coq, inspired by domain theory and the recently proposed interaction trees. We also present an accompanying separation logic for reasoning about guarded interaction trees. To demonstrate that guarded interaction trees provide a convenient domain for interpreting higher-order languages with effects, we define an interpretation of a PCF-like language with effects and show that this interpretation is sound and computationally adequate; we prove the latter using a logical relation defined using the separation logic. Guarded interaction trees also allow us to combine different effects and reason about them modularly. To illustrate this point, we give a modular proof of type soundness of cross-language interactions for safe interoperability of different higher-order languages with different effects. All results in the paper are formalized in Coq using the Iris logic over guarded type theory.

CCS Concepts: • **Theory of computation** → **Program semantics**; *Logic and verification*; • **Software and its engineering** → *Software libraries and repositories.*

Additional Key Words and Phrases: Coq, Iris, denotational semantics, logical relations

ACM Reference Format:
Dan Frumin, Amin Timany, and Lars Birkedal. 2024. Modular Denotational Semantics for Effects with Guarded Interaction Trees. *Proc. ACM Program. Lang.* 8, POPL, Article 12 (January 2024), 30 pages. https://doi.org/10.1145/3632854

1 INTRODUCTION

Interaction trees [Xia et al. 2019] are a recently proposed formalism for representing and reasoning about (possibly) non-terminating programs with side effects in Coq (a terminating type theory without effects). Since its inception, interaction trees have been applied, including but not limited,

The original GITrees paper (POPL'23)

GITrees (Definition)

Note: very heavy use of syntactic sugar;
Rocq does not directly support guarded types!

```
Guarded Type gitree (E : Effect) (R : Type) :=
| Ret : R -> gitree E R
| Fun : ▶(gitree E R -> gitree E R) -> gitree E R
| Err : gitree E R
| Tau : ▶gitree E R -> gitree E R
| Vis (i : E) :
  (Ins i (▶gitree E R)) ∗ (Outs i (▶gitree E R)) -> ▶gitree E R ) ->
  gitree E R
```

- In the above the `gitree` arguments only appear **guarded** (*i.e.*, under a ▶)
- `Fun`: embeds function values
- `Err`: Indicates error, *e.g.*, if a non-function value is applied to another value
- Input and output types can now also include ITrees (enables higher-order effects)

GITrees

Embedding λ_{ref}: STLC with higher-order store

- A couple of examples:

Meta-level (Rocq) λ

Syntactic (STLC) λ

$$[\![\lambda x\,.\,e]\!]_\rho = \texttt{Fun}\big(\text{next}\big(\lambda v\,.\,[\![e]\!]_{\rho[x\mapsto v]}\big)\big)$$

$$[\![!\ell]\!]_\rho = \texttt{Vis Load}\ (\ell, \lambda x\,.\,x)$$

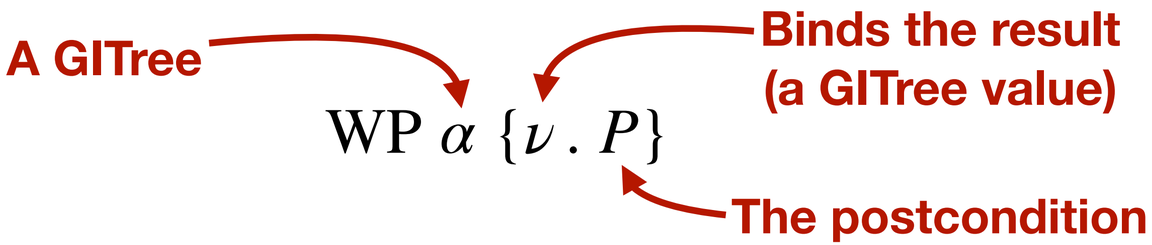
The continuation

- Where
  - ρ maps free variables to `gitree E R`
  - next : $A \to \blacktriangleright A$

GITrees

The Program Logic

- A program logic (weakest precondition calculus)



  - Where GITree values (ranged over by $\nu, \mu, \ldots$): a GITree of the form `Ret` or `Fun`
- Intuitive reading:
  - The GITree α does not result in an `Err`, **and**
  - The result, a GITree value, *i.e.*, a `Ret` or `Fun`, satisfies P

GITrees

The Program Logic — Effect Reification

- To define what GITrees "result in" we need to **reify** effects
  - We pick a **state** for each effect
    - For the heap of λ_{ref} we use finite partial maps $\mathrm{Loc} \xrightarrow{\text{fin}} \mathrm{GITV}$
  - An **effect reifier** maps a pair of input and state to an output and a state:

```
(Ins i (gitree E R) * State) -> option (Outs i (gitree E R) * State)
```

Effect's input

The state before the effect

Effect's output

The state after the effect

  - Note the option: If the reifier returns `None`, e.g., loading from an unallocated location, effect reification results in an `Err`

GITrees

The Program Logic

- Provides basic reasoning principles for GITree operations, including effects
- Which, based on their reification allow us to derive, *e.g.,* **WP-Load** below:

Separation logic's points-to: the location ℓ stores value μ

$$\frac{\ell \mapsto \mu \qquad \Phi(\mu)}{\mathrm{WP}\ [\![!\ell]\!]_\rho\ \{\nu\,.\,\Phi(\nu)\}}\ \textbf{WP-Load}$$

GITrees

The Program Logic

- Provides basic reasoning principles for GITree operations, including effects
- Which, based on their reification allow us to derive, *e.g.,* **WP-Load** below:

Separation logic's points-to: the location ℓ stores value μ

$$\frac{\ell \mapsto \mu \qquad \Phi(\mu)}{\mathrm{WP}\ [\![!\ell]\!]_\rho\ \{\nu\ .\ \Phi(\nu)\}}\ \textbf{WP-Load}$$

- **Q**: Modular reasoning? Ideally, from **WP-Load** should be able to **easily derivable**:

$$\frac{\mathrm{WP}\ [\![e]\!]_\rho\ \left\{\nu\ .\ \exists \ell\ .\ \nu = \mathit{Ret}(\ell) \wedge \mathrm{WP}\ [\![!\ell]\!]_\rho\ \{\mu\ .\ \Phi(\mu)\}\right\}}{\mathrm{WP}\ [\![!e]\!]_\rho\ \{\nu\ .\ \Phi(\nu)\}}\ \textbf{WP-Load-Compound}$$

GITrees

The Program Logic — Modular Reasoning

$$\frac{\mathrm{WP}\ [\![e]\!]_\rho\ \left\{\nu\,.\,\exists \ell\,.\,\nu = \mathit{Ret}(\ell) \wedge \mathrm{WP}\ [\![!\ell]\!]_\rho\ \{\mu\,.\,\Phi(\mu)\}\right\}}{\mathrm{WP}\ [\![!e]\!]_\rho\ \{\nu\,.\,\Phi(\nu)\}}\ \textbf{WP-Load-Compound} \qquad \frac{\ell \mapsto \mu \qquad \Phi(\mu)}{\mathrm{WP}\ [\![!\ell]\!]_\rho\ \{\nu\,.\,\Phi(\nu)\}}\ \textbf{WP-Load}$$

- The **WP-Hom** rule, crucial for modular reasoning

Semantic criterion ($\mathrm{Hom} \subseteq \mathrm{GITree} \to \mathrm{GITree}$); see Frumin *et al.* 2023

$$\frac{f \in \mathrm{Hom} \qquad \mathrm{WP}\ \alpha\ \left\{\nu\,.\,\mathrm{WP}\ f(\nu)\ \{\mu\,.\,\Phi(\mu)\}\right\}}{\mathrm{WP}\ f(\alpha)\ \{\nu\,.\,\Phi(\nu)\}}\ \textbf{WP-Hom}$$

- For any evaluation context K of λ_{ref}, $[\![K]\!] \in \mathrm{Hom}$
  - We can derive **WP-Load-Compound** from **WP-Load** and **WP-Hom** as $!\,\cdot$ is an evaluation context

Context-Dependent Effects

Denotational Semantics for Control Effects

Control Effects

- Control effects' evaluation depends on and/or manipulates (part of) the valuation context, *e.g.*, call/cc, shift-reset, exceptions
  - Operational semantics call/cc:

$$K[\text{call/cc}(x\,.\,e)] \to K[e[\text{cont } K/x]]$$

$$K[\text{throw } v\ (\text{cont } K')] \to K'[v]$$

- **Q**: Can we embed control effects in GITrees?

Can We Embed Control Effects in GITrees?

- The type of GITrees is rich enough to support control effects
- However, GITree reifiers are not suitable for control effects because
  - Reifiers do not have access to the continuations

```
(Ins i (gitree E R) * State) -> option (Outs i (gitree E R) * State)
```

Can We Embed Control Effects in GITrees?

- The type of GITrees is rich enough to support control effects
- However, GITree reifiers are not suitable for control effects because
  - Reifiers do not have access to the continuations

```
(Ins i (gitree E R) ∗ State) -> option (Outs i (gitree E R) ∗ State)
```

- Solution: give the reifier access to the continuation of the effect

```
(Ins i (gitree E R) ∗ State) -> (Outs i (gitree E R) -> gitree E R) -> option (gitree E R ∗ State)
```

The continuation

  - This changes the program logic. It breaks **WP-Hom**!

Context-Dependent Reifiers Break WP-Hom

- The **WP-Hom** rule is no longer valid (the result of α could depend on f)

$$\frac{f \in \mathrm{Hom} \qquad \mathrm{WP}\ \alpha\ \{\nu\ .\ \mathrm{WP}\ f(\nu)\ \{\mu\ .\ \Phi(\mu)\}\}}{\mathrm{WP}\ f(\alpha)\ \{\nu\ .\ \Phi(\nu)\}}\ \textbf{WP-Hom}$$

- Inspired by previous work, we introduce context-local WP's (clwp)
  - Intuitively: clwp of α holds if wp holds and α has no context-dependent effect

$$\frac{f \in \mathrm{Hom} \qquad \mathrm{CLWP}\ \alpha\ \{\nu\ .\ \mathrm{CLWP}\ f(\nu)\ \{\mu\ .\ \Phi(\mu)\}\}}{\mathrm{CLWP}\ f(\alpha)\ \{\nu\ .\ \Phi(\nu)\}}\ \textbf{CLWP-Hom}$$

  - No clwp inference rule for context-dependent effects, *e.g.*, $[\![\mathrm{call/cc}(x\ .\ e)]\!]_\rho$

Denotational Semantics for Control Effects

- We give denotational semantics for
  - a language with call/cc
  - a language with shift-reset
- Show both soundness and adequacy wrt operational semantics

See the exact statements in the paper!

$$\text{Soundness}: \quad \Sigma_1 \to \Sigma_2 \implies [\![\Sigma_1]\!] \cong [\![\Sigma_2]\!]$$

$$\text{Adequacy}: \quad [\![e]\!] \cong [\![n]\!] \implies e \to^* n$$

$$\begin{aligned}
&\mathbf{E}[\![x]\!]_\rho = \rho(x)\\
&\mathbf{E}[\![\mathsf{call/cc}\ (\mathsf{x.}\ e)]\!]_\rho = \mathrm{Callcc}(\lambda(f : \blacktriangleright\mathbf{IT} \to \blacktriangleright\mathbf{IT}).\, \mathbf{E}[\![e]\!]_{\rho[x \mapsto \mathsf{Fun}(\mathsf{next}(\lambda y.\ \mathsf{Tau}(f(\mathsf{next}(y)))))]})\\
&\mathbf{E}[\![\mathsf{throw}\ e_1\ \mathsf{to}\ e_2]\!]_\rho = \mathsf{get_val}(\mathbf{E}[\![e_1]\!]_\rho, \lambda x.\, \mathsf{get_fun}(\mathbf{E}[\![e_2]\!]_\rho, \lambda f.\, \mathrm{Throw}(x, f)))\\
&\mathbf{V}[\![\mathsf{cont}\ K]\!]_\rho = \mathsf{Fun}(\mathsf{next}(\lambda x.\, \mathsf{Tau}(\mathbf{K}[\![K]\!]_\rho\ (\blacktriangleright\bullet)\ \mathsf{next}(x))))\\
&\mathbf{K}[\![\mathsf{throw}\ K\ \mathsf{to}\ e]\!]_\rho = \lambda x.\, \mathsf{get_val}(\mathbf{K}[\![K]\!]_\rho\ x, \lambda y.\, \mathsf{get_fun}(\mathbf{E}[\![e]\!]_\rho, \lambda f.\, \mathrm{Throw}(y, f)))\\
&\mathbf{K}[\![\mathsf{throw}\ v\ \mathsf{to}\ K]\!]_\rho = \lambda x.\, \mathsf{get_val}(\mathbf{V}[\![v]\!]_\rho, \lambda y.\, \mathsf{get_fun}(\mathbf{K}[\![K]\!]_\rho\ x, \lambda f.\, \mathrm{Throw}(y, f)))
\end{aligned}$$

An excerpt of the denotational semantics for call/cc

$$\begin{aligned}
&\mathbf{E}[\![\mathcal{D}\ \mathsf{e}]\!]_\rho = \mathsf{Reset}(\mathbf{P}(\mathbf{E}[\![e]\!]_\rho))\\
&\mathbf{E}[\![\mathcal{S}\ \mathsf{x.}\ e]\!]_\rho = \mathsf{Shift}(\mathbf{P} \circ (\lambda\kappa.\, \mathbf{E}[\![e]\!]_{\rho, x \mapsto \mathsf{Fun}(\mathsf{next}(\lambda y.\ \mathsf{Tau}(\kappa(\mathsf{next}y))))}))\\
&\mathbf{E}[\![e_1\ @\ e_2]\!]_\rho = \mathsf{get_val}(\mathbf{E}[\![e_2]\!]_\rho, \lambda x.\, \mathsf{get_fun}(\mathbf{E}[\![e_1]\!]_\rho, \lambda y.\, \mathsf{Appcont}(\mathsf{next}(x), y)))\\
&\mathbf{V}[\![\mathsf{cont}\ K]\!]_\rho = \mathsf{Fun}(\mathsf{next}(\lambda x.\, \mathsf{Tick}(\mathbf{P}(\mathbf{K}[\![K]\!]_\rho\ x))))\\
&\mathbf{K}[\![K[\square\ @\ v]]\!]_\rho = \lambda x.\, \mathbf{K}[\![K]\!]_\rho(\mathbf{E}[\![x\ @\ v]\!]_\rho)\\
&\mathbf{M}[\![mk]\!]_\rho = \mathrm{map}(\lambda k.\, \mathbf{P} \circ \mathbf{K}[\![k]\!]_\rho)mk\\
&\mathbf{S}[\![\langle e,\ K,\ mk\rangle_{\mathsf{eval}}]\!]_\rho = (\mathbf{P}(\mathbf{E}[\![K[e]]\!]_\rho), \mathbf{M}[\![mk]\!]_\rho)\\
&\mathbf{S}[\![\langle K,\ v,\ mk\rangle_{\mathsf{cont}}]\!]_\rho = (\mathbf{P}(\mathbf{E}[\![K[v]]\!]_\rho), \mathbf{M}[\![mk]\!]_\rho)\\
&\mathbf{S}[\![\langle mk,\ v\rangle_{\mathsf{mcont}}]\!]_\rho = (\mathbf{P}(\mathbf{V}[\![v]\!]_\rho), \mathbf{M}[\![mk]\!]_\rho)\\
&\mathbf{S}[\![\langle e\rangle_{\mathsf{term}}]\!]_\rho = (\mathbf{P}(\mathbf{E}[\![e]\!]_\rho), [])\\
&\mathbf{S}[\![\langle v\rangle_{\mathsf{ret}}]\!]_\rho = (\mathbf{V}[\![v]\!]_\rho, [])
\end{aligned}$$

An excerpt of the denotational semantics for shift-reset

Denotational Semantics for Control Effects

See the exact statements in the paper!

$$\text{Soundness}: \quad \Sigma_1 \to \Sigma_2 \implies [\![\Sigma_1]\!] \cong [\![\Sigma_2]\!]$$

$$\text{Adequacy}: \quad [\![e]\!] \cong [\![n]\!] \implies e \to^* n$$

- Adequacy is proven
  - Using bi-orthogonal logical relations (**LR**)
    - A well-known technique
    - Expressed in terms of wp's (not clwp's)
    - Bi-orthogonal LR "bake in" modularity w.r.t. evaluation contexts

$$\mathbf{E}[\![x]\!]_\rho = \rho(x)$$
$$\mathbf{E}[\![\text{call/cc } (\text{x. } e)]\!]_\rho = \text{Callcc}(\lambda(f : \blacktriangleright\mathbf{IT} \to \blacktriangleright\mathbf{IT}).\, \mathbf{E}[\![e]\!]_{\rho[x \mapsto \text{Fun}(\text{next}(\lambda y.\ \text{Tau}(f(\text{next}(y)))))]})$$
$$\mathbf{E}[\![\text{throw } e_1 \text{ to } e_2]\!]_\rho = \text{get\_val}(\mathbf{E}[\![e_1]\!]_\rho, \lambda x.\, \text{get\_fun}(\mathbf{E}[\![e_2]\!]_\rho, \lambda f.\, \text{Throw}(x, f)))$$
$$\mathbf{V}[\![\text{cont } K]\!]_\rho = \text{Fun}(\text{next}(\lambda x.\, \text{Tau}(\mathbf{K}[\![K]\!]_\rho\ (\blacktriangleright\bullet)\ \text{next}(x))))$$
$$\mathbf{K}[\![\text{throw } K \text{ to } e]\!]_\rho = \lambda x.\, \text{get\_val}(\mathbf{K}[\![K]\!]_\rho\ x, \lambda y.\, \text{get\_fun}(\mathbf{E}[\![e]\!]_\rho, \lambda f.\, \text{Throw}(y, f)))$$
$$\mathbf{K}[\![\text{throw } v \text{ to } K]\!]_\rho = \lambda x.\, \text{get\_val}(\mathbf{V}[\![v]\!]_\rho, \lambda y.\, \text{get\_fun}(\mathbf{K}[\![K]\!]_\rho\ x, \lambda f.\, \text{Throw}(y, f)))$$

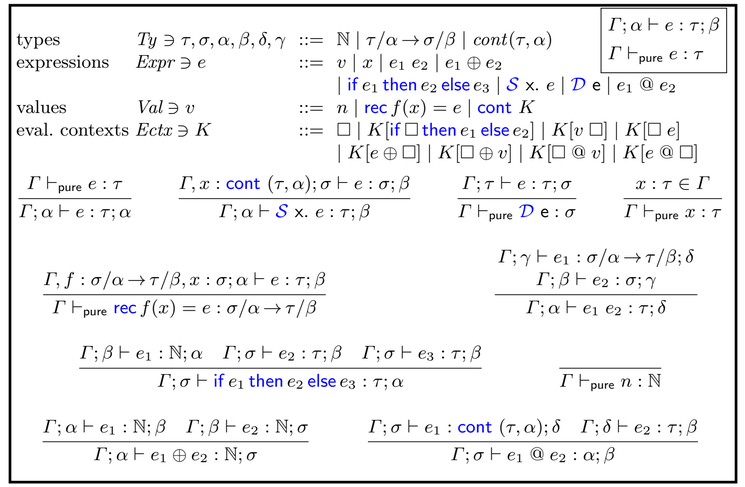
An excerpt of the denotational semantics for call/cc

$$\mathbf{E}[\![\mathcal{D}\ \text{e}]\!]_\rho = \text{Reset}(\mathbf{P}(\mathbf{E}[\![e]\!]_\rho))$$
$$\mathbf{E}[\![\mathcal{S}\ \text{x. } e]\!]_\rho = \text{Shift}(\mathbf{P} \circ (\lambda\kappa.\, \mathbf{E}[\![e]\!]_{\rho, x \mapsto \text{Fun}(\text{next}(\lambda y.\ \text{Tau}(\kappa(\text{next}y)))))}))$$
$$\mathbf{E}[\![e_1\ @\ e_2]\!]_\rho = \text{get\_val}(\mathbf{E}[\![e_2]\!]_\rho, \lambda x.\, \text{get\_fun}(\mathbf{E}[\![e_1]\!]_\rho, \lambda y.\, \text{Appcont}(\text{next}(x), y)))$$
$$\mathbf{V}[\![\text{cont } K]\!]_\rho = \text{Fun}(\text{next}(\lambda x.\, \text{Tick}(\mathbf{P}(\mathbf{K}[\![K]\!]_\rho\ x))))$$
$$\mathbf{K}[\![K[\square\ @\ v]]\!]_\rho = \lambda x.\, \mathbf{K}[\![K]\!]_\rho(\mathbf{E}[\![x\ @\ v]\!]_\rho)$$
$$\mathbf{M}[\![mk]\!]_\rho = \text{map}(\lambda k.\, \mathbf{P} \circ \mathbf{K}[\![k]\!]_\rho)mk$$
$$\mathbf{S}[\![\langle e,\ K,\ mk\rangle_{\text{eval}}]\!]_\rho = (\mathbf{P}(\mathbf{E}[\![K[e]]\!]_\rho), \mathbf{M}[\![mk]\!]_\rho)$$
$$\mathbf{S}[\![\langle K,\ v,\ mk\rangle_{\text{cont}}]\!]_\rho = (\mathbf{P}(\mathbf{E}[\![K[v]]\!]_\rho), \mathbf{M}[\![mk]\!]_\rho)$$
$$\mathbf{S}[\![\langle mk,\ v\rangle_{\text{mcont}}]\!]_\rho = (\mathbf{P}(\mathbf{V}[\![v]\!]_\rho), \mathbf{M}[\![mk]\!]_\rho)$$
$$\mathbf{S}[\![\langle e\rangle_{\text{term}}]\!]_\rho = (\mathbf{P}(\mathbf{E}[\![e]\!]_\rho), [])$$
$$\mathbf{S}[\![\langle v\rangle_{\text{ret}}]\!]_\rho = (\mathbf{V}[\![v]\!]_\rho, [])$$

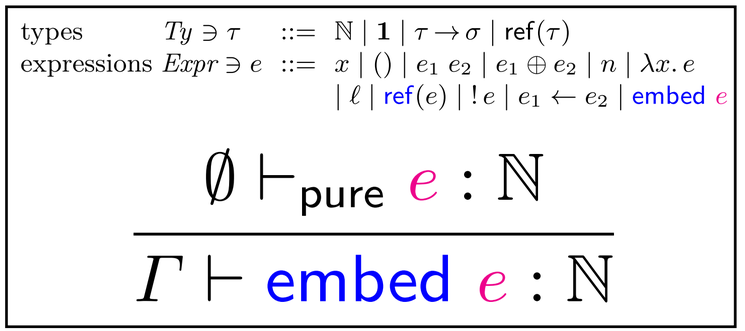
An excerpt of the denotational semantics for shift-reset

Language Interaction

- We show **semantic type safety** for λ_{embed}
  - Only allow embedding pure programs of λ_{delim}
- *Bi-orthogonal* LR for λ_{delim} using wp's
- *Non-bi-orthogonal* LR for λ_{embed} using clwp's
  - **Key point**: we can use clwp's for *all of* λ_{embed}
  - **Derive** clwp for embed e from a wp for e:
    - The wp comes from the bi-orthogonal LR



λ_{delim}: A language with shift-reset



λ_{embed}: A language with no control effects where we embed pure (properly delimited) expressions from λ_{delim}