Preserving consistency under dependent interoperability

Antoine Gaulin

McGill University

September 15, 2026

Introduction

What is interoperability?

- Two (or more) languages, $\mathcal{L}_1$ and $\mathcal{L}_2$, able to invoke each other's programs
- Dependent interoperability is when at least one of the languages is dependently-typed

Introduction

What is interoperability?

- Two (or more) languages, $\mathcal{L}_1$ and $\mathcal{L}_2$, able to invoke each other's programs
- Dependent interoperability is when at least one of the languages is dependently-typed

Two main approaches for dependent interoperability

1. [Osera et al., 2012, Dagand et al., 2018] Using boundary crossing terms
   - Two generic languages: $\mathcal{L}_1$ with dependent types, $\mathcal{L}_2$ with simple types
   - Can invoke each others programs freely by wrapping them in boundary crossing terms
   - Evaluation proceeds by translating programs from one language to the other
   - Languages can be very different, but breaks consistency of $\mathcal{L}_1$ if $\mathcal{L}_2$ is inconsistent

Introduction

What is interoperability?

- Two (or more) languages, $\mathcal{L}_1$ and $\mathcal{L}_2$, able to invoke each other's programs
- Dependent interoperability is when at least one of the languages is dependently-typed

Two main approaches for dependent interoperability

1. [Osera et al., 2012, Dagand et al., 2018] Using boundary crossing terms
   - Two generic languages: $\mathcal{L}_1$ with dependent types, $\mathcal{L}_2$ with simple types
   - Can invoke each others programs freely by wrapping them in boundary crossing terms
   - Evaluation proceeds by translating programs from one language to the other
   - Languages can be very different, but breaks consistency of $\mathcal{L}_1$ if $\mathcal{L}_2$ is inconsistent
2. [Stump et al., 2010, Casinghino et al., 2014] Using a box modality (with let-box eliminator)
   - Two fixed languages: $\mathcal{L}_1$ with primitive recursion, $\mathcal{L}_2$ with general recursion
   - Both are dependently-typed; $\mathcal{L}_1$ is a restricted version of $\mathcal{L}_2$
   - Arbitrary $\mathcal{L}_1$ programs can be boxed into $\mathcal{L}_2$, but only $\mathcal{L}_2$ values can be boxed into $\mathcal{L}_1$
   - Consistency of $\mathcal{L}_1$ is preserved, but languages are fixed and interactions are limited

Introduction

Goal: A dependent interoperability framework such that

1. Interoperating languages are kept abstract, with focus on allowed dependencies
   - Could be multiple dependently-typed systems (interoperability between proof assistants)
   - Dependently-typed system with arbitrary universe hierarchy
   - Simply-typed system may or may not have general recursion
2. Can be mechanized to extract verified type checkers for interoperating systems
   - Formally proven to preserve semantics of programs
3. Consistency is preserved under language interactions
   - Support interoperability involving proof assistant without sacrificing trust
   - Maintains decidability of type checking

Introduction

Goal: A dependent interoperability framework such that

1. Interoperating languages are kept abstract, with focus on allowed dependencies
   - Could be multiple dependently-typed systems (interoperability between proof assistants)
   - Dependently-typed system with arbitrary universe hierarchy
   - Simply-typed system may or may not have general recursion
2. Can be mechanized to extract verified type checkers for interoperating systems
   - Formally proven to preserve semantics of programs
3. Consistency is preserved under language interactions
   - Support interoperability involving proof assistant without sacrificing trust
   - Maintains decidability of type checking

Progress so far

- We already know how to address goals 1 and 2 based on our previous work
- Some idea on how to address goal 3 based on existing work

Generic language specifications

Goal 1: Interoperating languages are kept abstract

- May have different sort hierarchies (simple types/dependent types/polymorphism)
- May support different type constructors

Generic language specifications

Goal 1: Interoperating languages are kept abstract

- May have different sort hierarchies (simple types/dependent types/polymorphism)
- May support different type constructors

The PTS* framework

- Gives a uniform representation for several languages
  - Simply-typed: STLC, MiniML
  - Dependently-typed: LF, MLTT (cumulative or non-cumulative)
  - Polymorphic: System F, CoC
  - Inconsistent: System U, Type-in-Type

Generic language specifications

Goal 1: Interoperating languages are kept abstract

- May have different sort hierarchies (simple types/dependent types/polymorphism)
- May support different type constructors

The PTS* framework

- Gives a uniform representation for several languages
  - Simply-typed: STLC, MiniML
  - Dependently-typed: LF, MLTT (cumulative or non-cumulative)
  - Polymorphic: System F, CoC
  - Inconsistent: System U, Type-in-Type
- Language definition parameterized by sort hierarchies
- Each type constructor controlled by a relation on sorts
  - Language specifications are modular
  - Fine-grained control over features supported in a language

Definition (PTS* signatures)

A PTS* *signature* is a tuple $(\mathcal{S}t, \mathcal{A}x, \mathcal{A}x^{\sqsubseteq}, \mathcal{R}u^{\Pi}, \mathcal{R}u^{\mathbb{N}})$, where:

- $\mathcal{S}t$ is a collection of *sorts*, denoted $\mathbf{s}$, that classify types;
- $\mathcal{A}x(\mathbf{s}_1, \mathbf{s}_2)$ is a binary relation of *typing axioms*;
- $\mathcal{A}x^{\sqsubseteq}(\mathbf{s}_1, \mathbf{s}_2)$ is a binary relation of *subtyping axioms*;
- $\mathcal{R}u^{\Pi}(\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3)$ is a ternary relation of *formation rules for function spaces*;
- $\mathcal{R}u^{\mathbb{N}}(\mathbf{s})$ is a predicate of *formation rules for natural numbers*.

Definition (PTS* signatures)

A PTS* *signature* is a tuple $(\mathcal{S}t, \mathcal{A}x, \mathcal{A}x^{\sqsubseteq}, \mathcal{R}u^{\Pi}, \mathcal{R}u^{\mathbb{N}})$, where:

- $\mathcal{S}t$ is a collection of *sorts*, denoted $\mathbf{s}$, that classify types;
- $\mathcal{A}x(\mathbf{s}_1, \mathbf{s}_2)$ is a binary relation of *typing axioms*;
- $\mathcal{A}x^{\sqsubseteq}(\mathbf{s}_1, \mathbf{s}_2)$ is a binary relation of *subtyping axioms*;
- $\mathcal{R}u^{\Pi}(\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3)$ is a ternary relation of *formation rules for function spaces*;
- $\mathcal{R}u^{\mathbb{N}}(\mathbf{s})$ is a predicate of *formation rules for natural numbers*.

- Parameters control rules for type formation

$$\frac{\mathcal{A}x(\mathbf{s}_1, \mathbf{s}_2)}{\Gamma \vdash \mathbf{s}_1 : \mathbf{s}_2} \qquad \frac{\mathcal{R}u^{\Pi}(\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3) \quad \Gamma \vdash A : \mathbf{s}_1 \quad \Gamma, x{:}A \vdash B : \mathbf{s}_2}{\Gamma \vdash \Pi x{:}A.B : \mathbf{s}_3} \qquad \frac{\mathcal{R}u^{\mathbb{N}}(\mathbf{s})}{\Gamma \vdash \mathsf{Nat} : \mathbf{s}}$$

Definition (PTS* signatures)

A PTS* *signature* is a tuple $(\mathcal{S}t, \mathcal{A}x, \mathcal{A}x^{\sqsubseteq}, \mathcal{R}u^{\Pi}, \mathcal{R}u^{\mathbb{N}})$, where:

- $\mathcal{S}t$ is a collection of *sorts*, denoted $\mathbf{s}$, that classify types;
- $\mathcal{A}x(\mathbf{s}_1, \mathbf{s}_2)$ is a binary relation of *typing axioms*;
- $\mathcal{A}x^{\sqsubseteq}(\mathbf{s}_1, \mathbf{s}_2)$ is a binary relation of *subtyping axioms*;
- $\mathcal{R}u^{\Pi}(\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3)$ is a ternary relation of *formation rules for function spaces*;
- $\mathcal{R}u^{\mathbb{N}}(\mathbf{s})$ is a predicate of *formation rules for natural numbers*.

- Parameters control rules for type formation

$$\frac{\mathcal{A}x(\mathbf{s}_1, \mathbf{s}_2)}{\Gamma \vdash \mathbf{s}_1 : \mathbf{s}_2} \qquad \frac{\mathcal{R}u^{\Pi}(\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3) \quad \Gamma \vdash A : \mathbf{s}_1 \quad \Gamma, x{:}A \vdash B : \mathbf{s}_2}{\Gamma \vdash \Pi x{:}A.B : \mathbf{s}_3} \qquad \frac{\mathcal{R}u^{\mathbb{N}}(\mathbf{s})}{\Gamma \vdash \mathsf{Nat} : \mathbf{s}}$$

- Rules for well-typed terms are more or less standard

$$\frac{\Gamma, x{:}A \vdash M : B \quad \Gamma \vdash \Pi x{:}A.B}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B} \qquad \frac{\Gamma \vdash M : \Pi x{:}A.B \quad \Gamma \vdash N : A \quad \Gamma \vdash \Pi x{:}A.B}{\Gamma \vdash M\,N : B[N/x]}$$

Key examples of PTS*

- MiniML, a simply-typed language with recursion of natural numbers
  - One sort $\star \in \mathcal{S}t$, no axiom, $\mathcal{R}u^{\Pi}(\star,\star,\star)$, and $\mathcal{R}u^{\mathbb{N}}(\star)$
  - No dependent types: e.g. $\Pi x{:}\mathsf{Nat}.\star$ ill-formed because we do not have $\star : \star$
  - Adding axiom $\mathcal{A}x(\star,\star)$ produces the inconsistent Type-in-Type

Key examples of PTS*

- MiniML, a simply-typed language with recursion of natural numbers
  - One sort $\star \in \mathcal{S}t$, no axiom, $\mathcal{R}u^{\Pi}(\star,\star,\star)$, and $\mathcal{R}u^{\mathbb{N}}(\star)$
  - No dependent types: e.g. $\Pi x{:}\mathsf{Nat}.\star$ ill-formed because we do not have $\star : \star$
  - Adding axiom $\mathcal{A}x(\star,\star)$ produces the inconsistent Type-in-Type
- LF, a dependently-typed language with not recursion
  - Two sorts $\star, \Box \in \mathcal{S}t$, $\mathcal{A}x(\star,\Box)$, no subtyping axioms
  - $\mathcal{R}u^{\Pi}(\star,\Box,\Box)$ allows formation of dependent types, $\mathcal{R}u^{\Pi}(\star,\star,\star)$ is usual function space
  - $\mathcal{R}u^{\mathbb{N}}$ is an empty relation, so natural numbers cannot be formed (feature modularity)

Key examples of PTS*

- MiniML, a simply-typed language with recursion of natural numbers
  - One sort $\star \in \mathcal{S}t$, no axiom, $\mathcal{R}u^{\Pi}(\star,\star,\star)$, and $\mathcal{R}u^{\mathbb{N}}(\star)$
  - No dependent types: e.g. $\Pi x{:}\mathsf{Nat}.\star$ ill-formed because we do not have $\star : \star$
  - Adding axiom $\mathcal{A}x(\star,\star)$ produces the inconsistent Type-in-Type
- LF, a dependently-typed language with not recursion
  - Two sorts $\star, \Box \in \mathcal{S}t$, $\mathcal{A}x(\star,\Box)$, no subtyping axioms
  - $\mathcal{R}u^{\Pi}(\star,\Box,\Box)$ allows formation of dependent types, $\mathcal{R}u^{\Pi}(\star,\star,\star)$ is usual function space
  - $\mathcal{R}u^{\mathbb{N}}$ is an empty relation, so natural numbers cannot be formed (feature modularity)
- Variants of MLTT, a full dependent type theory with recursion over natural numbers
  - Infinite sort hierarchy $\mathcal{S}t = \{\mathsf{Type}_i \mid i \in \mathsf{Nat}\}$, with $\mathcal{A}x(\mathsf{Type}_i, \mathsf{Type}_{i+1})$ for all $i \in \mathsf{Nat}$
  - $\mathcal{R}u^{\Pi}(\mathsf{Type}_i, \mathsf{Type}_j, \mathsf{Type}_{\max(i,j)})$ for all $i, j \in \mathsf{Nat}$, and $\mathcal{R}u^{\mathbb{N}}(\mathsf{Type}_0)$
  - Control cumulativity: $\mathcal{A}x^{\sqsubseteq}(\mathsf{Type}_i, \mathsf{Type}_{i+1})$ is cumulative, empty $\mathcal{A}x^{\sqsubseteq}$ is non-cumulative

Generic proofs

- Theorems are stated parametrically in the PTS* signature
- Proofs apply to all PTS* instances
- e.g. Presupposition: In any PTS*, if $\Gamma \vdash M \approx M' : A$, then $\Gamma \vdash M : A$ and $\Gamma \vdash M' : A$

Generic proofs

- Theorems are stated parametrically in the PTS* signature
- Proofs apply to all PTS* instances
- e.g. Presupposition: In any PTS*, if $\Gamma \vdash M \approx M' : A$, then $\Gamma \vdash M : A$ and $\Gamma \vdash M' : A$

Conditional proofs

- Some theorems are started only for PTS* signatures with some additional properties
- e.g. Predicativity: There is a well-founded order $\mathbf{s}_1 < \mathbf{s}_2$ over $\mathcal{S}t$ that is respected by relations
  - If $\mathcal{A}x(\mathbf{s}_1, \mathbf{s}_2)$, then $\mathbf{s}_1 < \mathbf{s}_2$; if $\mathcal{R}u^{\Pi}(\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3)$, then $\mathbf{s}_1, \mathbf{s}_2 \leq \mathbf{s}_3$
  - Every predicative PTS* signature generates a normalizing system (proved with NbE)
  - Corollary: Every predicative PTS* signature generates a consistent type theory

Goal 2: Mechanize and extract verified type checkers

- Implement type checker in Rocq, extract to OCaml
  - Rocq functions must be total, i.e. terminating on every input
- Key result: Decidability of type checking
  - Ensures we can write a correct, terminating algorithm in Rocq

Goal 2: Mechanize and extract verified type checkers

- Implement type checker in Rocq, extract to OCaml
  - Rocq functions must be total, i.e. terminating on every input
- Key result: Decidability of type checking
  - Ensures we can write a correct, terminating algorithm in Rocq

The McPTS* infrastructure

- Mechanization of the PTS* framework
  - Definition of PTS*, with basic properties proven generically
  - Normalization for predicative signatures
  - Algorithm for functional signatures (i.e. all relations are partial functions)
- Decidability of type checking follows from normalization
  - Main challenge: deciding equality of terms
  - Easy for normal forms: just compare syntactically (as long as sort equality is decidable)

Preserving consistency under interoperability

Goal 3: Consistency is preserved under language interactions

- Say $\mathcal{L}_1$ is a consistent PTS^* and $\mathcal{L}_2$ is any PTS^*
- Want to extend $\mathcal{L}_1$ with the ability to use some $\mathcal{L}_2$ programs, without breaking consistency

Preserving consistency under interoperability

Goal 3: Consistency is preserved under language interactions

- Say $\mathcal{L}_1$ is a consistent PTS^* and $\mathcal{L}_2$ is any PTS^*
- Want to extend $\mathcal{L}_1$ with the ability to use some $\mathcal{L}_2$ programs, without breaking consistency
- First obstacle: PTS^* is minimalistic, no useful form of inconsistency
  - Only primitive recursion, which enforces structurally smaller recursive calls
  - Only pure computations

Preserving consistency under interoperability

Goal 3: Consistency is preserved under language interactions

- Say $\mathcal{L}_1$ is a consistent PTS^* and $\mathcal{L}_2$ is any PTS^*
- Want to extend $\mathcal{L}_1$ with the ability to use some $\mathcal{L}_2$ programs, without breaking consistency
- First obstacle: PTS^* is minimalistic, no useful form of inconsistency
  - Only primitive recursion, which enforces structurally smaller recursive calls
  - Only pure computations

Extending PTS^* with more practical features

- Features that may break consistency in a useful way
  - General recursion
  - Effects

Preserving consistency under interoperability

Goal 3: Consistency is preserved under language interactions

- Say $\mathcal{L}_1$ is a consistent PTS^* and $\mathcal{L}_2$ is any PTS^*
- Want to extend $\mathcal{L}_1$ with the ability to use some $\mathcal{L}_2$ programs, without breaking consistency
- First obstacle: PTS^* is minimalistic, no useful form of inconsistency
  - Only primitive recursion, which enforces structurally smaller recursive calls
  - Only pure computations

Extending PTS^* with more practical features

- Features that may break consistency in a useful way
  - General recursion
  - Effects
- More type constructors: tuples, disjoin sums, general inductive types (future work), etc.
  - Some programs can make sense only in one of the interoperating languages
- Substructurality (Rafael)
  - Can also be controlled with parameters

Preserving consistency under interoperability

Two forms of interoperability

Boundary crossing approach: $\mathcal{L}_1$ directly uses a $\mathcal{L}_2$ program M

- M must be restricted so it makes sense in the setting of $\mathcal{L}_1$
  - e.g. If M is non-terminating, using it breaks $\mathcal{L}_1$'s consistency
  - Need a way to isolate fragments of $\mathcal{L}_2$ consistent with $\mathcal{L}_1$
- May need to reconstruct dependencies
  - e.g. $\mathcal{L}_2$ has simple lists, but $\mathcal{L}_1$ has vectors (indexed by length)
  - Need to translate the representation to recover the length
  - Not obvious in general

Preserving consistency under interoperability

Two forms of interoperability

Boundary crossing approach: $\mathcal{L}_1$ directly uses a $\mathcal{L}_2$ program M

- M must be restricted so it makes sense in the setting of $\mathcal{L}_1$
  - e.g. If M is non-terminating, using it breaks $\mathcal{L}_1$'s consistency
  - Need a way to isolate fragments of $\mathcal{L}_2$ consistent with $\mathcal{L}_1$
- May need to reconstruct dependencies
  - e.g. $\mathcal{L}_2$ has simple lists, but $\mathcal{L}_1$ has vectors (indexed by length)
  - Need to translate the representation to recover the length
  - Not obvious in general

Modal approach: $\mathcal{L}_1$ is used for meta-programming with a boxed $\mathcal{L}_2$ program M

- Think of $\mathcal{L}_2$ programs as suspended from $\mathcal{L}_1$'s perspective
  - No issue with non-termination since boxed programs don't run
- Types in $\mathcal{L}_1$ can depend on objects from $\mathcal{L}_2$
  - Need to decide conversion in $\mathcal{L}_1$, and thus equality of $\mathcal{L}_2$ programs appearing in $\mathcal{L}_1$ types
  - Still need to isolate fragments of $\mathcal{L}_2$ with decidable equality

Isolating fragments in the PTS* framework

- Fix a PTS* signature $\mathcal{P}_1 = (\mathcal{S}t_1, \mathcal{A}x_1, \mathcal{A}x_1^{\sqsubseteq}, \mathcal{R}u_1^{\Pi}, \mathcal{R}u_1^{\mathbb{N}})$
- A signature $\mathcal{P}_2 = (\mathcal{S}t_2, \mathcal{A}x_2, \mathcal{A}x_2^{\sqsubseteq}, \mathcal{R}u_2^{\Pi}, \mathcal{R}u_2^{\mathbb{N}})$ generates a fragment of $\mathcal{P}_1$ if:
  - There is a (injective?) function $f : \mathcal{S}t_2 \to \mathcal{S}t_1$;
  - If $\mathcal{A}x_2(\mathbf{s}_1, \mathbf{s}_2)$, then $\mathcal{A}x_1(f(\mathbf{s_1}), f(\mathbf{s_2}))$;
  - If $\mathcal{A}x_2^{\sqsubseteq}(\mathbf{s}_1, \mathbf{s}_2)$, then $\mathcal{A}x_1^{\sqsubseteq}(f(\mathbf{s}_1), f(\mathbf{s}_2))$;
  - If $\mathcal{R}u_2^{\Pi}(\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3)$ then $\mathcal{R}u_1^{\Pi}(f(\mathbf{s}_1), f(\mathbf{s}_2), f(\mathbf{s}_3))$;
  - If $\mathcal{R}u_2^{\mathbb{N}}(\mathbf{s})$, then $\mathcal{R}u_1^{\mathbb{N}}(f(\mathbf{s}))$.
- Any such function f generates a sound embedding of $\mathcal{P}_2$ into $\mathcal{P}_1$

Isolating fragments in the PTS* framework

- Fix a PTS* signature $\mathcal{P}_1 = (\mathcal{S}t_1, \mathcal{A}x_1, \mathcal{A}x_1^{\sqsubseteq}, \mathcal{R}u_1^{\Pi}, \mathcal{R}u_1^{\mathbb{N}})$
- A signature $\mathcal{P}_2 = (\mathcal{S}t_2, \mathcal{A}x_2, \mathcal{A}x_2^{\sqsubseteq}, \mathcal{R}u_2^{\Pi}, \mathcal{R}u_2^{\mathbb{N}})$ generates a fragment of $\mathcal{P}_1$ if:
  - There is a (injective?) function $f : \mathcal{S}t_2 \to \mathcal{S}t_1$;
  - If $\mathcal{A}x_2(\mathbf{s}_1, \mathbf{s}_2)$, then $\mathcal{A}x_1(f(\mathbf{s_1}), f(\mathbf{s_2}))$;
  - If $\mathcal{A}x_2^{\sqsubseteq}(\mathbf{s}_1, \mathbf{s}_2)$, then $\mathcal{A}x_1^{\sqsubseteq}(f(\mathbf{s}_1), f(\mathbf{s}_2))$;
  - If $\mathcal{R}u_2^{\Pi}(\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3)$ then $\mathcal{R}u_1^{\Pi}(f(\mathbf{s}_1), f(\mathbf{s}_2), f(\mathbf{s}_3))$;
  - If $\mathcal{R}u_2^{\mathbb{N}}(\mathbf{s})$, then $\mathcal{R}u_1^{\mathbb{N}}(f(\mathbf{s}))$.
- Any such function f generates a sound embedding of $\mathcal{P}_2$ into $\mathcal{P}_1$
- Given $M \in \mathcal{L}_2$ for interoperability with $\mathcal{L}_1$
  - M can cross the boundary if it belongs to a common fragment of $\mathcal{L}_1$ and $\mathcal{L}_2$
  - An $\mathcal{L}_1$ type can depend on an $\mathcal{L}_2$ term M if it belongs to a normalizing fragment of $\mathcal{L}_2$

Conclusion

Recap

- Three goals for a generic dependent interoperability framework:
  1. Interoperating languages are kept abstract
  2. Amenable to mechanization and extraction of verified compilers for interoperating systems
  3. Consistency is preserved under language interactions

Conclusion

Recap

- Three goals for a generic dependent interoperability framework:
  1. Interoperating languages are kept abstract
  2. Amenable to mechanization and extraction of verified compilers for interoperating systems
  3. Consistency is preserved under language interactions
- For goals 1 and 2, we build on our previous work
  - PTS* framework for generic language definitions
  - McPTS* for extraction of generic verified type-checkers

Conclusion

Recap

- Three goals for a generic dependent interoperability framework:
  1. Interoperating languages are kept abstract
  2. Amenable to mechanization and extraction of verified compilers for interoperating systems
  3. Consistency is preserved under language interactions
- For goals 1 and 2, we build on our previous work
  - PTS* framework for generic language definitions
  - McPTS* for extraction of generic verified type-checkers
- For goal 3, a key challenge is identifying which program is safe for interoperability
  - PTS* is well-suited for isolating safe fragments

Conclusion

Recap

- Three goals for a generic dependent interoperability framework:
  1. Interoperating languages are kept abstract
  2. Amenable to mechanization and extraction of verified compilers for interoperating systems
  3. Consistency is preserved under language interactions
- For goals 1 and 2, we build on our previous work
  - PTS* framework for generic language definitions
  - McPTS* for extraction of generic verified type-checkers
- For goal 3, a key challenge is identifying which program is safe for interoperability
  - PTS* is well-suited for isolating safe fragments

Todo list

- Extend PTS* with more practical features
  - General recursion and effects, maybe more type constructors
- Extend PTS* with the actual interoperability mechanism
  - Boundary-crossing vs modality
  - Implement fragment isolation stuff

Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. *SIGPLAN Not.*, 49(1):33–45, January 2014. ISSN 0362-1340. doi: 10.1145/2578855.2535883. URL https://doi.org/10.1145/2578855.2535883.

Pierre-Évariste Dagand, Nicolas Tabareau, and Éric Tanter. Foundations of dependent interoperability. *J. Funct. Program.*, 28:e9, 2018. doi: 10.1017/S0956796818000011. URL https://doi.org/10.1017/S0956796818000011.

Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *ACM Trans. Program. Lang. Syst.*, 31(3):12:1–12:44, 2009. doi: 10.1145/1498926.1498930. URL https://doi.org/10.1145/1498926.1498930.

Peter-Michael Osera, Vilhelm Sjöberg, and Steve Zdancewic. Dependent interoperability. In Koen Claessen and Nikhil Swamy, editors, *Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA, January 24, 2012*, pages 3–14. ACM, 2012. doi: 10.1145/2103776.2103779. URL https://doi.org/10.1145/2103776.2103779.

Aaron Stump, Vilhelm Sjöberg, and Stephanie Weirich. Termination casts: A flexible approach to termination with general recursion. *Electronic Proceedings in Theoretical Computer Science*, 43:76–93, December 2010. ISSN 2075-2180. doi: 10.4204/eptcs.43.6. URL http://dx.doi.org/10.4204/EPTCS.43.6.