Multi-language semantics

Antoine Gaulin

McGill University

November 26, 2024

Introduction

We want to assign meaning to programs from multiple languages simultaneously.

Introduction

We want to assign meaning to programs from multiple languages simultaneously.

Applications

- **Interoperability**
  Modern software are written in multiple languages that are allowed to interact.

Challenges

- Different languages have different features, providing different safety guarantees.
  - Type hierarchy (simple, dependent, polymorphic).
  - Substructural rules.
  - Effects.

  How to ensure each language's safety remains in the presence of interoperability?

Introduction

We want to assign meaning to programs from multiple languages simultaneously.

Applications

- **Interoperability**
  Modern software are written in multiple languages that are allowed to interact.
- **Compiler verification**
  Compilers transform programs in a sequence of intermediate language.

Challenges

- Different languages have different features, providing different safety guarantees.
  - Type hierarchy (simple, dependent, polymorphic).
  - Substructural rules.
  - Effects.

  How to ensure each language's safety remains in the presence of interoperability?
- How to ensure the semantics of input and output programs match?

Interoperability

Main approaches

1. Unsafe
   - Trust the programmer; don't type-check.
   - Run each program according to its own language's semantics.

INTEROPERABILITY

MAIN APPROACHES

1. Unsafe
   - Trust the programmer; don't type-check.
   - Run each program according to its own language's semantics.
2. Monolithic
   - Merge all the languages into a big language.
   - Use complicated annotations to control what program can use what features.
   - [Trifonov and Shao, 1999] Three languages with different effects.
   - [Ou et al., 2004] Simply-typed and a dependently-typed language.

Interoperability

Main approaches

1. Unsafe
   - Trust the programmer; don't type-check.
   - Run each program according to its own language's semantics.
2. Monolithic
   - Merge all the languages into a big language.
   - Use complicated annotations to control what program can use what features.
   - [Trifonov and Shao, 1999] Three languages with different effects.
   - [Ou et al., 2004] Simply-typed and a dependently-typed language.
3. Multi-language
   - Keep all languages separated.
   - Interactions restricted to special boundary-crossing terms.
   - [Matthews and Findler, 2009] Untyped and simply-typed languages.
   - [Osera et al., 2012] Simply-types and (first-order) dependently-typed languages.
   - [Scherer et al., 2018] Unrestricted and linear languages.

Interoperability

Dependent interoperability [Osera et al., 2012]

We have two languages:

$$
\begin{array}{llll}
\lambda^{\rightarrow} \text{ Programs} & s & ::= & x \mid \mathbf{c} \mid \lambda x{:}S.s \mid s_1\ s_2 \mid \mathsf{SD}^{S}_{T} t \\
 & & & \mid\ (\,)\ \mid \langle s_1, s_2 \rangle \mid \pi_1 s \mid \pi_2 s \\
\lambda^{\rightarrow} \text{ Types} & S & ::= & \mathbf{a} \mid \mathsf{Unit} \mid S_1 \rightarrow S_2 \mid S_1 \times S_2 \\
\lambda^{\rightarrow} \text{ Kinds} & L & ::= & \mathsf{type}
\end{array}
$$

Simply-typed $\lambda^{\rightarrow}$

$$
\begin{array}{llll}
\lambda^{\cong} \text{ Programs} & t & ::= & v \mid \mathbf{c} \mid \lambda v{:}T.t \mid t_1\ t_2 \mid \mathsf{DS}^{T}_{S} s \\
 & & & \mid\ (\,)\ \mid \langle t_1, t_2 \rangle \mid \pi_1 t \mid \pi_1 t \\
\lambda^{\cong} \text{ Types} & T & ::= & \mathbf{a} \mid \mathsf{Unit} \mid \Pi v{:}T_1.T_2 \mid \Sigma v{:}T_1.T_2 \\
\lambda^{\cong} \text{ Kinds} & K & ::= & \mathsf{type} \mid \Pi x{:}T.K
\end{array}
$$

Dependently-typed $\lambda^{\cong}$

Interoperability

Dependent interoperability [Osera et al., 2012]

We have two languages:

$$\begin{array}{llcl} \lambda^{\rightarrow}\text{ Programs} & s & ::= & x \mid \mathbf{c} \mid \lambda x{:}S.s \mid s_1\ s_2 \mid \mathsf{SD}^S_T t \\ & & & \mid\ (\,)\ \mid \langle s_1, s_2\rangle \mid \pi_1 s \mid \pi_2 s \\ \lambda^{\rightarrow}\text{ Types} & S & ::= & \mathbf{a} \mid \mathsf{Unit} \mid S_1 \rightarrow S_2 \mid S_1 \times S_2 \\ \lambda^{\rightarrow}\text{ Kinds} & L & ::= & \mathsf{type} \end{array}$$

Simply-typed $\lambda^{\rightarrow}$

$$\begin{array}{llcl} \lambda^{\cong}\text{ Programs} & t & ::= & v \mid \mathbf{c} \mid \lambda v{:}T.t \mid t_1\ t_2 \mid \mathsf{DS}^T_S s \\ & & & \mid\ (\,)\ \mid \langle t_1, t_2\rangle \mid \pi_1 t \mid \pi_1 t \\ \lambda^{\cong}\text{ Types} & T & ::= & \mathbf{a} \mid \mathsf{Unit} \mid \Pi v{:}T_1.T_2 \mid \Sigma v{:}T_1.T_2 \\ \lambda^{\cong}\text{ Kinds} & K & ::= & \mathsf{type} \mid \Pi x{:}T.K \end{array}$$

Dependently-typed $\lambda^{\cong}$

- Boundary crossing terms
  - $\mathsf{SD}^S_T t$ allows using a dependent program $t : T$ as a simple program of type S.
  - $\mathsf{DS}^T_S s$ allows using a simple program $s : S$ as a dependent program of type T.

INTEROPERABILITY

DEPENDENT INTEROPERABILITY [OSERA ET AL., 2012]

We have two languages:

$\lambda^{\rightarrow}$ Programs	s	$::=$	$x \mid \mathbf{c} \mid \lambda x{:}S.s \mid s_1\ s_2 \mid \mathsf{SD}^S_T t$
			$\mid () \mid \langle s_1, s_2 \rangle \mid \pi_1 s \mid \pi_2 s$
$\lambda^{\rightarrow}$ Types	S	$::=$	$\mathbf{a} \mid \mathsf{Unit} \mid S_1 \rightarrow S_2 \mid S_1 \times S_2$
$\lambda^{\rightarrow}$ Kinds	L	$::=$	type

Simply-typed $\lambda^{\rightarrow}$

$\lambda^{\cong}$ Programs	t	$::=$	$v \mid \mathbf{c} \mid \lambda v{:}T.t \mid t_1\ t_2 \mid \mathsf{DS}^T_S s$
			$\mid () \mid \langle t_1, t_2 \rangle \mid \pi_1 t \mid \pi_1 t$
$\lambda^{\cong}$ Types	T	$::=$	$\mathbf{a} \mid \mathsf{Unit} \mid \Pi v{:}T_1.T_2 \mid \Sigma v{:}T_1.T_2$
$\lambda^{\cong}$ Kinds	K	$::=$	$\mathsf{type} \mid \Pi x{:}T.K$

Dependently-typed $\lambda^{\cong}$

- Boundary crossing terms
  - $\mathsf{SD}^S_T t$ allows using a dependent program $t : T$ as a simple program of type S.
  - $\mathsf{DS}^T_S s$ allows using a simple program $s : S$ as a dependent program of type T.
- Variables x and v are different from each others, and share a context.

Interoperability

Dependent interoperability [Osera et al., 2012]

We have two languages:

$$\begin{array}{llcl} \lambda^{\rightarrow} \text{ Programs} & s & ::= & x \mid \mathbf{c} \mid \lambda x{:}S.s \mid s_1\ s_2 \mid \mathsf{SD}^S_T t \\ & & & \mid \ () \mid \langle s_1, s_2\rangle \mid \pi_1 s \mid \pi_2 s \\ \lambda^{\rightarrow} \text{ Types} & S & ::= & \mathbf{a} \mid \mathsf{Unit} \mid S_1 \rightarrow S_2 \mid S_1 \times S_2 \\ \lambda^{\rightarrow} \text{ Kinds} & L & ::= & \mathsf{type} \end{array}$$

Simply-typed $\lambda^{\rightarrow}$

$$\begin{array}{llcl} \lambda^{\cong} \text{ Programs} & t & ::= & v \mid \mathbf{c} \mid \lambda v{:}T.t \mid t_1\ t_2 \mid \mathsf{DS}^T_S s \\ & & & \mid \ () \mid \langle t_1, t_2\rangle \mid \pi_1 t \mid \pi_1 t \\ \lambda^{\cong} \text{ Types} & T & ::= & \mathbf{a} \mid \mathsf{Unit} \mid \Pi v{:}T_1.T_2 \mid \Sigma v{:}T_1.T_2 \\ \lambda^{\cong} \text{ Kinds} & K & ::= & \mathsf{type} \mid \Pi x{:}T.K \end{array}$$

Dependently-typed $\lambda^{\cong}$

- Boundary crossing terms
  - $\mathsf{SD}^S_T t$ allows using a dependent program $t : T$ as a simple program of type S.
  - $\mathsf{DS}^T_S s$ allows using a simple program $s : S$ as a dependent program of type T.
- Variables x and v are different from each others, and share a context.
- Constants **c**, **a** are shared inductive definitions.
  - Each type **a** has a simple and a dependent kind.
  - Each constructor **c** has a simple and a dependent type.
  - WLOG, every constant takes exactly one argument.

Interoperability

Dependent interoperability [Osera et al., 2012]

For boundary-crossing terms to work, we need to translate types.

$$\frac{}{\mathsf{Unit} \Leftrightarrow \mathsf{Unit}} \qquad \frac{(\mathbf{a} : K) \in \mathsf{Sig}}{\mathbf{a} \Leftrightarrow \mathbf{a}\ t} \qquad \frac{S_1 \Leftrightarrow T_1 \qquad S_2 \Leftrightarrow T_2}{S_1 \to S_2 \Leftrightarrow \Pi v{:}T_1.T_2}$$

Interoperability

Dependent interoperability [Osera et al., 2012]

For boundary-crossing terms to work, we need to translate types.

$$\frac{}{\mathsf{Unit} \Leftrightarrow \mathsf{Unit}} \qquad \frac{(\mathbf{a} : K) \in \mathsf{Sig}}{\mathbf{a} \Leftrightarrow \mathbf{a}\ t} \qquad \frac{S_1 \Leftrightarrow T_1 \qquad S_2 \Leftrightarrow T_2}{S_1 \to S_2 \Leftrightarrow \Pi v{:}T_1.T_2}$$

Typing just check that the two types are related:

$$\frac{\Gamma \Vdash t : T \qquad S \Leftrightarrow T}{\Gamma \Vdash \mathsf{SD}^S_T t : S} \qquad \frac{\Gamma \Vdash s : S \qquad \Gamma \Vdash T : \mathsf{type} \qquad S \Leftrightarrow T}{\Gamma \Vdash \mathsf{DS}^T_S s : T}$$

Interoperability

Dependent interoperability [Osera et al., 2012]

To evaluate boundary-crossing, we need to somehow translate arguments to constructors.

Interoperability

Dependent interoperability [Osera et al., 2012]

To evaluate boundary-crossing, we need to somehow translate arguments to constructors.

- Osera et al. [2012] assumes the user will provide translation functions for all constructors. Given $\mathbf{c} : S \to S'$ and $\mathbf{c} : \Pi v{:}T.T'$, define:

  $$\mathsf{argToS}_{\mathbf{c}} : T \to S \qquad\qquad \mathsf{argToD}_{\mathbf{c}} : S \to T$$

  Note these functions do not exist in either $\lambda^{\to}$ or $\lambda^{\cong}$.

INTEROPERABILITY

DEPENDENT INTEROPERABILITY [OSERA ET AL., 2012]

To evaluate boundary-crossing, we need to somehow translate arguments to constructors.

- Osera et al. [2012] assumes the user will provide translation functions for all constructors. Given $\mathbf{c} : S \to S'$ and $\mathbf{c} : \Pi v{:}T.T'$, define:

  $$\mathsf{argToS}_{\mathbf{c}} : T \to S \qquad\qquad \mathsf{argToD}_{\mathbf{c}} : S \to T$$

  Note these functions do not exist in either $\lambda^{\to}$ or $\lambda^{\cong}$.
- These translation can be expressed as ornaments [Dagand and McBride, 2014].

INTEROPERABILITY

DEPENDENT INTEROPERABILITY [OSERA ET AL., 2012]

To evaluate boundary-crossing, we need to somehow translate arguments to constructors.

- Osera et al. [2012] assumes the user will provide translation functions for all constructors. Given $\mathbf{c} : S \to S'$ and $\mathbf{c} : \Pi v{:}T.T'$, define:

$$\mathsf{argToS}_{\mathbf{c}} : T \to S \qquad \mathsf{argToD}_{\mathbf{c}} : S \to T$$

  Note these functions do not exist in either $\lambda^{\to}$ or $\lambda^{\cong}$.

- These translation can be expressed as ornaments [Dagand and McBride, 2014].

```
inductive list : type =
| Empty : list
| Cons : nat → list → list
```

```
inductive list : nat → type =
| Empty : list zero
| Cons : Πn:nat.nat → list n → list (succ n)
```

```
type ornament list-length (n : nat) : list ⇒ list n
list-length zero Empty ⇒ Empty
list-length (succ n) (Cons m l) ⇒  Cons n (translate-nat m) (list-length n l)
```

Interoperability

Dependent interoperability [Osera et al., 2012]

Now, let us look at the evaluation rule for boundary crossing on the simple side:

$$\frac{\mathsf{argToS}_{\mathbf{c}}\ t = s}{\mathsf{SD}^{S'}_{T'}(\mathbf{c}\ t) \longrightarrow \mathbf{c}\ s}$$

Interoperability

Dependent interoperability [Osera et al., 2012]

Now, let us look at the evaluation rule for boundary crossing on the simple side:

$$\frac{\mathsf{argToS}_{\mathbf{c}}\ t = s}{\mathsf{SD}^{S'}_{T'}(\mathbf{c}\ t) \longrightarrow \mathbf{c}\ s}$$

$$\frac{}{\mathsf{SD}^{S \to S'}_{\Pi v:T.T'}(\lambda v{:}T.t) \longrightarrow \lambda x{:}S.\mathsf{SD}^{S'}_{[\mathsf{DS}^{T}_{S} x/v]T'}((\lambda v{:}T.t)\ (\mathsf{DS}^{T}_{S} x))}$$

Interoperability

Dependent interoperability [Osera et al., 2012]

Now, let us look at the evaluation rule for boundary crossing on the simple side:

$$\frac{\mathsf{argToS}_{\mathbf{c}}\ t = s}{\mathsf{SD}^{S'}_{T'}(\mathbf{c}\ t) \longrightarrow \mathbf{c}\ s}$$

$$\frac{}{\mathsf{SD}^{S \to S'}_{\Pi v:T.T'}(\lambda v{:}T.t) \longrightarrow \lambda x{:}S.\mathsf{SD}^{S'}_{[\mathsf{DS}^{T}_{S} x/v]T'}((\lambda v{:}T.t)\ (\mathsf{DS}^{T}_{S} x))}$$

For crossing on the dependent side, rules are more or less symmetric:

$$\frac{\mathsf{argToD}_{\mathbf{c}}\ s = t \qquad (\mathbf{c} : \Pi v{:}T.T'') \in \mathsf{Sig} \qquad [t/v]T'' \cong T'}{\mathsf{DS}^{T'}_{S'}(\mathbf{c}\ s) \longrightarrow \mathbf{c}\ t}$$

Interoperability

Dependent interoperability [Osera et al., 2012]

Now, let us look at the evaluation rule for boundary crossing on the simple side:

$$\frac{\mathsf{argToS}_{\mathbf{c}}\ t = s}{\mathsf{SD}^{S'}_{T'}(\mathbf{c}\ t) \longrightarrow \mathbf{c}\ s}$$

$$\frac{}{\mathsf{SD}^{S \to S'}_{\Pi v:T.T'}(\lambda v{:}T.t) \longrightarrow \lambda x{:}S.\mathsf{SD}^{S'}_{[\mathsf{DS}^{T}_{S} x/v]T'}((\lambda v{:}T.t)\ (\mathsf{DS}^{T}_{S} x))}$$

For crossing on the dependent side, rules are more or less symmetric:

$$\frac{\mathsf{argToD}_{\mathbf{c}}\ s = t \qquad (\mathbf{c} : \Pi v{:}T.T'') \in \mathsf{Sig} \qquad [t/v]T'' \cong T'}{\mathsf{DS}^{T'}_{S'}(\mathbf{c}\ s) \longrightarrow \mathbf{c}\ t}$$

$$\frac{}{\mathsf{DS}^{\Pi v:T.T'}_{S \to S'} \lambda x{:}S.s \longrightarrow \lambda v{:}T.\mathsf{DS}^{T'}_{S'}((\lambda x{:}S.s)\ (\mathsf{SD}^{S}_{T} v))}$$

Interoperability

Dependent interoperability [Osera et al., 2012]

In this setting, we can prove the usual properties in the usual way:

- Substitution lemmas
- Type preservation
- Progress
- Canonical forms

$\vdots$

Interoperability

Dependent interoperability [Osera et al., 2012]

In this setting, we can prove the usual properties in the usual way:

- Substitution lemmas
- Type preservation
- Progress
- Canonical forms

$\vdots$

But we need some properties of argToS and argToD :

- Respect substitutions: $\mathsf{argToD}_{\mathbf{c}}\ [s/x]s' = [s/x]\mathsf{argToD}_{\mathbf{c}}\ s'$
- Respect evaluation: If $s \longrightarrow s'$, then $\mathsf{argToD}_{\mathbf{c}}\ s \longrightarrow \mathsf{argToD}_{\mathbf{c}}\ s'$

$\vdots$

Compiler verification

Another application of multi-language semantics is for compiler verification.

Compiler verification

Another application of multi-language semantics is for compiler verification.

Overview

1. Define source language $\mathcal{S}$ and target language $\mathcal{T}$.
2. Add boundary-crossing terms.
3. Use boundary-crossing to prove equivalence between source programs and target programs.

Compiler verification

Another application of multi-language semantics is for compiler verification.

Overview

1. Define source language $\mathcal{S}$ and target language $\mathcal{T}$.
2. Add boundary-crossing terms.
3. Use boundary-crossing to prove equivalence between source programs and target programs.

- Perconti and Ahmed [2014] applies this principle to a two-pass compiler.
  - Source language is System F (F).
  - First pass does closure conversion (C).
  - Second pass adds memory allocation (A).

  Boundary crossing allowed between F and C, and between C and A.
  Crossing between F and A can be done by combining the other crossings.

Compiler verification

Another application of multi-language semantics is for compiler verification.

Overview

1. Define source language $\mathcal{S}$ and target language $\mathcal{T}$.
2. Add boundary-crossing terms.
3. Use boundary-crossing to prove equivalence between source programs and target programs.

- Perconti and Ahmed [2014] applies this principle to a two-pass compiler.
  - Source language is System F (F).
  - First pass does closure conversion (C).
  - Second pass adds memory allocation (A).

  Boundary crossing allowed between F and C, and between C and A.
  Crossing between F and A can be done by combining the other crossings.
- Patterson et al. [2017] compiles a System F-style language to typed assembly.

Compiler verification

Another application of multi-language semantics is for compiler verification.

Overview

1. Define source language $\mathcal{S}$ and target language $\mathcal{T}$.
2. Add boundary-crossing terms.
3. Use boundary-crossing to prove equivalence between source programs and target programs.

- Perconti and Ahmed [2014] applies this principle to a two-pass compiler.
  - Source language is System F (F).
  - First pass does closure conversion (C).
  - Second pass adds memory allocation (A).

  Boundary crossing allowed between F and C, and between C and A.
  Crossing between F and A can be done by combining the other crossings.
- Patterson et al. [2017] compiles a System F-style language to typed assembly.

How do we know that compilation preserves semantics?

Compiler verification

Contextual equivalence

How do we know that compilation (of open programs) preserves semantics?

- For pure programs, it is straightforward since we can look at programs in isolation.
- If there are effects, especially memory allocations, we need to consider the surroundings of programs. Evaluation contexts capture this idea.

Compiler verification

Contextual equivalence

How do we know that compilation (of open programs) preserves semantics?

- For pure programs, it is straightforward since we can look at programs in isolation.
- If there are effects, especially memory allocations, we need to consider the surroundings of programs. Evaluation contexts capture this idea.

Evaluation contexts

- An *evaluation context* is a program with a single hole in it.

$$\text{Evaluation context} \quad \mathcal{C} \quad ::= \quad \diamond \mid \lambda x{:}A.\mathcal{C} \mid \mathcal{C}\ M \mid M\ \mathcal{C} \mid \ldots$$

Applying an evaluation context, $\mathcal{C}[M]$, to a term fills the hole with that term.

How do we know that compilation (of open programs) preserves semantics?

- For pure programs, it is straightforward since we can look at programs in isolation.
- If there are effects, especially memory allocations, we need to consider the surroundings of programs. Evaluation contexts capture this idea.

Evaluation contexts

- An *evaluation context* is a program with a single hole in it.

$$\text{Evaluation context} \quad \mathcal{C} \quad ::= \quad \diamond \mid \lambda x{:}A.\mathcal{C} \mid \mathcal{C}\; M \mid M\; \mathcal{C} \mid \ldots$$

  Applying an evaluation context, $\mathcal{C}[M]$, to a term fills the hole with that term.

- In the multi-language setting, we need mutually-defined evaluation contexts for every language.

$$\begin{array}{lcl} \mathcal{S}\ \text{Evaluation context} \quad \mathcal{C} & ::= & \ldots \mid \mathsf{CrossTo}_{\mathcal{S}}\mathcal{C} \\ \mathcal{T}\ \text{Evaluation context} \quad \mathcal{C} & ::= & \ldots \mid \mathsf{CrossTo}_{\mathcal{T}}\mathcal{C} \end{array}$$

  $\Rightarrow$ Programs from every language can be passed to evaluation context from every language.

COMPILER VERIFICATION

CONTEXTUAL EQUIVALENCE

Now, to prove correctness of compiler, we need two new judgments:

1. $\boxed{\Gamma \Vdash (s{:}S) \rightsquigarrow (t{:}T)}$ – $\mathcal{S}$ term s compiles to t.
   - Intuitively, t is the fully evaluated $\mathsf{CrossTo}_{\mathcal{T}} s$.

Compiler verification

Contextual equivalence

Now, to prove correctness of compiler, we need two new judgments:

1. $\boxed{\Gamma \Vdash (s{:}S) \rightsquigarrow (t{:}T)}$ – $\mathcal{S}$ term s compiles to t.
   - Intuitively, t is the fully evaluated $\mathsf{CrossTo}_{\mathcal{T}}s$.
2. $\boxed{\Gamma \Vdash (s{:}S) \approx (t{:}T)}$ – $\mathcal{S}$ term s and $\mathcal{T}$ term t are contextually equivalent.
   - For all $\mathcal{S}$ evaluation context $\mathcal{C}$, $\Gamma \Vdash \mathcal{C}[s] \equiv \mathcal{C}[\mathsf{CrossTo}_{\mathcal{S}}t] : S$, and
   - For all $\mathcal{T}$ evaluation context $\mathcal{C}$, $\Gamma \Vdash \mathcal{C}[\mathsf{CrossTo}_{\mathcal{T}}s] \equiv \mathcal{C}[t] : T$.

Compiler verification
Contextual equivalence

Now, to prove correctness of compiler, we need two new judgments:

1. $\boxed{\Gamma \Vdash (s{:}S) \rightsquigarrow (t{:}T)}$ – $\mathcal{S}$ term s compiles to t.
   - Intuitively, t is the fully evaluated $\mathsf{CrossTo}_{\mathcal{T}} s$.
2. $\boxed{\Gamma \Vdash (s{:}S) \approx (t{:}T)}$ – $\mathcal{S}$ term s and $\mathcal{T}$ term t are contextually equivalent.
   - For all $\mathcal{S}$ evaluation context $\mathcal{C}$, $\Gamma \Vdash \mathcal{C}[s] \equiv \mathcal{C}[\mathsf{CrossTo}_{\mathcal{S}} t] : S$, and
   - For all $\mathcal{T}$ evaluation context $\mathcal{C}$, $\Gamma \Vdash \mathcal{C}[\mathsf{CrossTo}_{\mathcal{T}} s] \equiv \mathcal{C}[t] : T$.

Theorem (Correctness of compilation)

If $\Gamma \Vdash (s{:}S) \rightsquigarrow (t{:}T)$, *then* $\Gamma \Vdash (s{:}S) \approx (t{:}T)$.

Conclusion

Recap

- There are three approaches to language interoperability.
  - The unsafe approach ignores all typing (bad).
  - The monolithic approach merges all language into one (impractical).
  - The multi-language approach extends each language with boundary-crossing terms.
- We looked into the inner workings of multi-languages with dependent interoperability.
- We discussed an application of multi-languages for compiler verification.

Conclusion

Recap

- There are three approaches to language interoperability.
  - The unsafe approach ignores all typing (bad).
  - The monolithic approach merges all language into one (impractical).
  - The multi-language approach extends each language with boundary-crossing terms.
- We looked into the inner workings of multi-languages with dependent interoperability.
- We discussed an application of multi-languages for compiler verification.

Limitations of multi-language approach

- Unclear if it scales to more than two languages.
- Not general; depends heavily on language-specific features.
- Not grounded on logic.

Pierre-Évariste Dagand and Conor McBride. Transporting functions across ornaments. *J. Funct. Program.*, 24(2-3):316–383, 2014. doi: 10.1017/S0956796814000069. URL `https://doi.org/10.1017/S0956796814000069`.

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`.

Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In Jean-Jacques Lévy, Ernst W. Mayr, and John C. Mitchell, editors, *Exploring New Frontiers of Theoretical Informatics, IFIP 18th World Computer Congress, TC1 3rd International Conference on Theoretical Computer Science (TCS2004), 22-27 August 2004, Toulouse, France*, volume 155 of *IFIP*, pages 437–450. Kluwer/Springer, 2004. doi: 10.1007/1-4020-8141-3_34. URL `https://doi.org/10.1007/1-4020-8141-3_34`.

Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. Funtal: reasonably mixing a functional language with assembly. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 495–509. ACM, 2017. doi: 10.1145/3062341.3062347. URL `https://doi.org/10.1145/3062341.3062347`.

James T. Perconti and Amal Ahmed. Verifying an open compiler using multi-language semantics. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of*

Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings, volume 8410 of *Lecture Notes in Computer Science*, pages 128–148. Springer, 2014. doi: 10.1007/978-3-642-54833-8_8. URL https://doi.org/10.1007/978-3-642-54833-8_8.

Gabriel Scherer, Max S. New, Nick Rioux, and Amal Ahmed. FabULous interoperability for ML and a linear language. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10803 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2018. doi: 10.1007/978-3-319-89366-2_8. URL https://doi.org/10.1007/978-3-319-89366-2_8.

Valery Trifonov and Zhong Shao. Safe and principled language interoperation. In S. Doaitse Swierstra, editor, *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings*, volume 1576 of *Lecture Notes in Computer Science*, pages 128–146. Springer, 1999. doi: 10.1007/3-540-49099-X_9. URL https://doi.org/10.1007/3-540-49099-X_9.