Distributive Disjoint Polymorphism for Compositional Programming

Xuan Bi [1], **Ningning Xie** [1], Bruno C. d. S. Oliveira [1], Tom Schrijvers [2]

ESOP 2019 9 April 2019

[1]The University of Hong Kong [2]KU Leuven

Motivation

Compositionality

Compositionality is a desirable property in programming designs.

- Compositionality is a key aspect of denotational semantics [Scott, 1970; Scott and Strachey, 1971].
- Compositional definitions are easy to reason and to extend.
- Programming techniques include: shallow embeddings of DSLs [Gibbons and Wu, 2014], finally tagless [Carette et al., 2009], object algebras [Oliveira and Cook, 2012].

Compositionality

- Yet, programming languages often only support simple compositional designs well.
- Our work improves on existing techniques by supporting highly modular and compositional designs.

Contributions

- A new calculus F_i^+ combining
  - disjoint intersection types
  - disjoint polymorphism
  - BCD-style distributive subtyping
- F_i^+ enables improved compositional programming designs.
- A semantic coherence proof based on canonicity relation.
- Coq proof for all metatheory except some manual proofs of decidability.
- Haskell implementation.

Language Features

Disjoint Intersection Types

- Intersection types: if `e::A` and `e::B`, then we have `e :: A & B`.

  Int & **Bool**
  (**Int** $\to$ **Int**) & (**Int** $\to$ **Bool**)

Disjoint Intersection Types

- Intersection types: if `e::A` and `e::B`, then we have `e :: A & B`.

```
Int & Bool
(Int → Int) & (Int → Bool)
```

- In many languages and calculi (e.g. Muehlboeck and Tate [2018]), intersection types do not increase the expressiveness of terms.

```
Int & Bool -- uninhabited
```

Disjoint Intersection Types

- Intersection types: if `e::A` and `e::B`, then we have `e :: A & B`.

```
Int & Bool
(Int → Int) & (Int → Bool)
```

- In many languages and calculi (e.g. Muehlboeck and Tate [2018]), intersection types do not increase the expressiveness of terms.

```
Int & Bool -- uninhabited
```

- The merge operator increases the expressiveness of terms.

```
1 ,, True :: Int & Bool
1 ,, True :: Int
1 ,, True :: Bool
```

Disjoint Intersection Types

- However, merges can introduce ambiguity:

```
1 ,, True :: Int & Bool -- evaluates to (1, True)
1 ,, True :: Int        -- evaluates to 1
1 ,, True :: Bool        -- evaluates to True
```

Disjoint Intersection Types

- However, merges can introduce ambiguity:

```
1 ,, True :: Int & Bool -- evaluates to (1, True)
1 ,, True :: Int        -- evaluates to 1
1 ,, True :: Bool        -- evaluates to True

1 ,, 2    :: Int        -- 1, or 2
```

Disjoint Intersection Types

- However, merges can introduce ambiguity:

```
1 ,, True :: Int & Bool -- evaluates to (1, True)
1 ,, True :: Int        -- evaluates to 1
1 ,, True :: Bool        -- evaluates to True

1 ,, 2    :: Int        -- 1, or 2
```

- The disjointness judgment guarantees that only two terms with disjoint types can be merged: given `e1::A` and `e2::B`, only if we have $A * B$, we can have `e1,,e2`.

```
1 ,, True :: Int & Bool -- valid as Int * Bool
1 ,, 2    :: Int & Int  -- invalid
```

Disjoint Polymorphism

- The disjointness quantification introduces a disjointness constraint for type variables:

```
Λa*Int. \x:a. (x ,, 3)
Λa. Λb*a. \x:a. \y:b. (x ,, y) -- polymorphic merge
```

Distributive subtyping

- F_i^+ features BCD-style subtyping [Barendregt et al., 1983], where intersection types distribute over arrows, records, and universal quantifications

$$(\mathsf{Int} \to \mathsf{Int}) \,\&\, (\mathsf{Int} \to \mathsf{Bool}) <: \mathsf{Int} \to (\mathsf{Int} \,\&\, \mathsf{Bool})$$

$$\{l : \mathsf{Int}\} \,\&\, \{l : \mathsf{Bool}\} <: \{l : \mathsf{Int} \,\&\, \mathsf{Bool}\}$$

$$(\forall(\alpha * \mathsf{Int}).\, \mathsf{Int}) \,\&\, (\forall(\alpha * \mathsf{Int}).\, \mathsf{Bool}) <: \forall(\alpha * \mathsf{Int}).\, \mathsf{Int} \,\&\, \mathsf{Bool}$$

Compositional Programming

Compositional Programming

To demonstrate the compositional properties of F_i^+, we use Gibbons and Wu [2014]'s shallow embeddings of parallel prefix circuits.

- The finally tagless encoding [Carette et al., 2009] in Haskell
- The encoding in SEDEL [Bi and Oliveira, 2018], a source language built on top of F_i^+

Compositional Programming

Two questions to answer:

- ☞ How can we compose multiple interpretations?
- ☞ How can we compose dependent interpretations?

Parallel Prefix Circuit

The circuit DSL represents networks that map a number of inputs $x_1, \ldots, x_n$ onto the same number of outputs $y_1, \ldots, y_n$, where $y_i = x_1 \oplus x_2 \oplus \ldots \oplus x_i$.

Conceptually,

```
identity :: Int → C
fan      :: Int → C
beside   :: C → C → C
above    :: C → C → C
stretch  :: [Int] → C → C
```

Parallel Prefix Circuit

The circuit DSL represents networks that map a number of inputs $x_1, \ldots, x_n$ onto the same number of outputs $y_1, \ldots, y_n$, where $y_i = x_1 \oplus x_2 \oplus \ldots \oplus x_i$.

Conceptually,

```
identity :: Int → C
fan      :: Int → C
beside   :: C → C → C
above    :: C → C → C
stretch  :: [Int] → C → C
```

`identity 4`

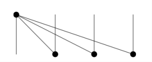
$y_i = x_i$

Parallel Prefix Circuit

The circuit DSL represents networks that map a number of inputs $x_1, \ldots, x_n$ onto the same number of outputs $y_1, \ldots, y_n$, where $y_i = x_1 \oplus x_2 \oplus \ldots \oplus x_i$.

Conceptually,

```
identity :: Int → C
fan      :: Int → C
beside   :: C → C → C
above    :: C → C → C
stretch  :: [Int] → C → C
```



`fan 4`

$y_1 = x_1$

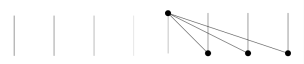
$y_i = x_1 \oplus x_i$

Parallel Prefix Circuit

The circuit DSL represents networks that map a number of inputs $x_1, \ldots, x_n$ onto the same number of outputs $y_1, \ldots, y_n$, where $y_i = x_1 \oplus x_2 \oplus \ldots \oplus x_i$.

Conceptually,

```
identity :: Int → C
fan      :: Int → C
beside   :: C → C → C
above    :: C → C → C
stretch  :: [Int] → C → C
```



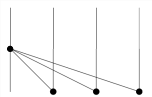
```
beside (identity 4)
       (fan 4)
```

Parallel Prefix Circuit

The circuit DSL represents networks that map a number of inputs $x_1, \ldots, x_n$ onto the same number of outputs $y_1, \ldots, y_n$, where $y_i = x_1 \oplus x_2 \oplus \ldots \oplus x_i$.

Conceptually,

```
identity :: Int → C
fan      :: Int → C
beside   :: C → C → C
above    :: C → C → C
stretch  :: [Int] → C → C
```



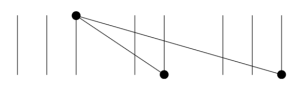
```
above (identity 4)
      (fan 4)
```

Parallel Prefix Circuit

The circuit DSL represents networks that map a number of inputs $x_1, \ldots, x_n$ onto the same number of outputs $y_1, \ldots, y_n$, where $y_i = x_1 \oplus x_2 \oplus \ldots \oplus x_i$.

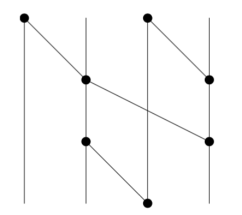
Conceptually,

```
identity :: Int → C
fan      :: Int → C
beside   :: C → C → C
above    :: C → C → C
stretch  :: [Int] → C → C
```



```
stretch [3,2,3]
        (fan 3)
```

Parallel Prefix Circuit



Brent-Kung circuit of width 4:

```
above (beside (fan 2) (fan 2))
  (above (stretch [2, 2] (fan 2))
    (beside (beside (identity 1) (fan 2))
      (identity 1)))
```

Parallel Prefix Circuit

For the purpose of presentation, we only focus on:

```
identity :: Int → C
beside   :: C → C → C
above    :: C → C → C
```

Parallel Prefix Circuit

```
class Circuit c where
  identity :: Int → c
  beside   :: c → c → c
  above    :: c → c → c
```

In Finally tagless, DSL is defined as Haskell type classes.

Parallel Prefix Circuit

```haskell
class Circuit c where
  identity :: Int → c
  beside   :: c → c → c
  above    :: c → c → c
```

In Finally tagless, DSL is defined as Haskell type classes.

In SEDEL, DSL is defines as polymorphic record types.

```
type Circuit[C] = {
  identity : Int → C,
  beside : C → C → C,
  above : C → C → C
}
```

Parallel Prefix Circuit

```
data Width = W { width :: Int }
instance Circuit Width where
  identity n   = W n
  beside c1 c2 = W (width c1 + width c2)
  above c1 c2  = c1
```

In Finally tagless, interpretation is defined as Haskell instances.

Parallel Prefix Circuit

```
data Width = W { width :: Int }
instance Circuit Width where
  identity n    = W n
  beside c1 c2 = W (width c1 + width c2)
  above c1 c2   = c1
```

In Finally tagless, interpretation is defined as Haskell instances.

In SEDEL, interpretation is defined as record terms.

```
type Width = { width : Int };
language1 : Circuit[Width] = {
  identity (n : Int) = { width = n },
  beside   (c1 : Width) (c2 : Width) = { width = c1.width +
    c2.width },
  above    (c1 : Width) (c2 : Width) = { width = c1.width }
}
```

Parallel Prefix Circuit

```
data Depth = D { depth :: Int }
instance Circuit Depth where
  identity n   = D 0
  beside c1 c2 = D (max (depth c1) (depth c2))
  above c1 c2  = D (depth c1 + depth c2)
```

In Finally tagless, interpretation is defined as Haskell instances.

In SEDEL, interpretation is defined as record terms.

```
type Depth = { depth : Int };
language2 : Circuit[Depth] = {
  identity (n : Int) = { depth = 0 },
  beside   (c1 : Depth) (c2 : Depth) = { depth = max c1.depth
    c2.depth},
  above    (c1 : Depth) (c2 : Depth) = { depth = c1.depth +
    c2.depth}
}
```

Parallel Prefix Circuit

```
type DCircuit = ∀ c. Circuit c ⇒ c

brentKung :: DCircuit =
  above (beside (fan 2) (fan 2)) (above (stretch [2,2] (fan 2))
      (beside (beside (identity 1) (fan 2)) (identity 1)))
e1 :: Width = brentKung
e2 :: Depth = brentKung
```

With polymorphism, we can define a type that support multiple interpretations

---

```
type DCircuit = { accept : ∀ C. Circuit[C] → C };
brentKung : DCircuit = {
  accept C l = l.above (l.beside (l.fan 2) (l.fan 2))
    (l.above (l.stretch (cons 2 (cons 2 nil)) (l.fan 2))
      (l.beside (l.beside (l.identity 1) (l.fan 2))
    (l.identity 1))) };
e1 = brentKung.accept Width language1;
e2 = brentKung.accept Depth language2;
```

Parallel Prefix Circuit

☞ How can we compose multiple interpretations?

Parallel Prefix Circuit

```
instance (Circuit c1, Circuit c2) ⇒ Circuit (c1, c2) where
 identity n  = (identity n, identity n)
 beside c1 c2= (beside (fst c1)(fst c2),beside (snd c1)(snd c2))
 above  c1 c2= (above (fst c1) (fst c2),above (snd c1) (snd c2))

e3 :: (Width, Depth)
e3 = brentKung
```

☞ How can we compose multiple interpretations?

Parallel Prefix Circuit

```
instance (Circuit c1, Circuit c2) ⇒ Circuit (c1, c2) where
 identity n  = (identity n, identity n)
 beside c1 c2= (beside (fst c1)(fst c2),beside (snd c1)(snd c2))
 above  c1 c2= (above (fst c1) (fst c2),above (snd c1) (snd c2))

e3 :: (Width, Depth)
e3 = brentKung
```

☞ How can we compose multiple interpretations?

```
language3 : Circuit[Width & Depth] = language1 ,, language2;
e3 = brentKung.accept (Width & Depth) language3;
```

Parallel Prefix Circuit

```
instance (Circuit c1, Circuit c2) ⇒ Circuit (c1, c2) where
 identity n  = (identity n, identity n)
 beside c1 c2= (beside (fst c1)(fst c2),beside (snd c1)(snd c2))
 above  c1 c2= (above (fst c1) (fst c2),above (snd c1) (snd c2))

e3 :: (Width, Depth)
e3 = brentKung
```

☞ How can we compose multiple interpretations?

```
language3 : Circuit[Width & Depth] = language1 ,, language2;
e3 = brentKung.accept (Width & Depth) language3;
```

```
Circuit[Width] & Circuit[Depth] <: Circuit[Width & Depth]
```

Parallel Prefix Circuit

☞ How can we compose dependent interpretations ?

Parallel Prefix Circuit

```
data WellSized = WS { wS :: Bool, ox :: Width }
instance Circuit WellSized where
 identity n   = WS True (identity n)
 beside c1 c2 = WS (wS c1 && wS c2) (beside (ox c1) (ox c2))
 above c1 c2  = WS (wS c1 && wS c2 && width (ox c1) == width
    (ox c2)) (above (ox c1) (ox c2))

e4 :: WellSized = brenkKung
```

☞ How can we compose dependent interpretations ?

Parallel Prefix Circuit

```
data WellSized = WS { wS :: Bool, ox :: Width }
instance Circuit WellSized where
 identity n   = WS True (identity n)
 beside c1 c2 = WS (wS c1 && wS c2) (beside (ox c1) (ox c2))
 above c1 c2  = WS (wS c1 && wS c2 && width (ox c1) == width
    (ox c2)) (above (ox c1) (ox c2))

e4 :: WellSized = brenkKung
```

☞ How can we compose dependent interpretations ?

```
type WellSized = { wS : Bool };
language4 = {
  identity (n : Int) = { wS = true },
  above (c1 : WellSized & Width) (c2 : WellSized & Width) =
    { wS = c1.wS && c2.wS && c1.width == c2.width },
  beside (c1:WellSized) (c2:WellSized) = {wS= c1.wS && c2.wS}
}
e4 = brentKung.accept (WellSized & Width) (language1 ,,
    language4)
```

Parallel Prefix Circuit

- For multiple interpretations, SEDEL encoding simply compose existing components;
- For dependent interpretations, SEDEL encoding only needs the new interpretation and simply compose it with existing ones.

Parallel Prefix Circuit

- For multiple interpretations, SEDEL encoding simply compose existing components;
- For dependent interpretations, SEDEL encoding only needs the new interpretation and simply compose it with existing ones.

☞ F_i^+ improves on existing techniques by supporting highly modular and compositional designs!

Declarative Type System

Syntax of F_i^+

Types	A, B, C	$::=$	$\mathsf{Int} \mid \top \mid \bot \mid A \to B \mid A \,\&\, B$
		$\mid$	$\{l : A\} \mid \alpha \mid \forall(\alpha * A).\, B$
Expressions	E	$::=$	$x \mid i \mid \top \mid \lambda x.\, E \mid E_1\, E_2 \mid E_1 \,,,\, E_2$
		$\mid$	$E : A \mid \{l = E\} \mid E.l$
		$\mid$	$\Lambda(\alpha * A).\, E \mid E\, A$
Term contexts	Γ	$::=$	$\bullet \mid \Gamma, x : A$
Type contexts	Δ	$::=$	$\bullet \mid \Delta, \alpha * A$

Semantics

$\boxed{A <: B}$ *(selected rules for subtyping)*

$$\frac{B_1 <: B_2 \qquad A_2 <: A_1}{\forall(\alpha * A_1).\, B_1 <: \forall(\alpha * A_2).\, B_2}$$

$$\frac{}{(A_1 \to A_2) \,\&\, (A_1 \to A_3) <: A_1 \to A_2 \,\&\, A_3}$$

$$\frac{}{\{l : A\} \,\&\, \{l : B\} <: \{l : A \,\&\, B\}}$$

$$\frac{}{(\forall(\alpha * A).\, B_1) \,\&\, (\forall(\alpha * A).\, B_2) <: \forall(\alpha * A).\, B_1 \,\&\, B_2}$$

Semantics

$\boxed{\Delta \vdash A * B}$ *(selected rules for disjointness)*

$$\frac{\Delta \vdash A_1 * B \qquad \Delta \vdash A_2 * B}{\Delta \vdash A_1 \,\&\, A_2 * B}$$

$$\frac{\Delta, \alpha * A_1 \,\&\, A_2 \vdash B_1 * B_2}{\Delta \vdash \forall(\alpha * A_1).\, B_1 * \forall(\alpha * A_2).\, B_2} \qquad \frac{(\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash \alpha * B}$$

Semantics

$\boxed{\Delta; \Gamma \vdash E \Leftrightarrow A}$ *(selected rules for typing)*

$$\frac{\Delta; \Gamma \vdash E_1 \Rightarrow A_1 \qquad \Delta; \Gamma \vdash E_2 \Rightarrow A_2 \qquad \Delta \vdash A_1 * A_2}{\Delta; \Gamma \vdash E_1 \,,, \, E_2 \Rightarrow A_1 \,\&\, A_2}$$

$$\frac{\Delta \vdash A \qquad \Delta, \alpha * A; \Gamma \vdash E \Rightarrow B}{\Delta; \Gamma \vdash \Lambda(\alpha * A).\, E \Rightarrow \forall(\alpha * A).\, B}$$

$$\frac{\Delta; \Gamma \vdash E \Rightarrow \forall(\alpha * B).\, C \qquad \Delta \vdash A * B}{\Delta; \Gamma \vdash E\, A \Rightarrow [A/\alpha]C}$$

Dynamic Semantics

- Type-directed translation into a target calculus F_{co}, which extends System F with products and coercions

Types	τ	$::=$	$\mathsf{Int} \mid \langle\rangle \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \alpha \mid \forall\alpha.\,\tau$
Terms	e	$::=$	$x \mid i \mid \langle\rangle \mid \lambda x.\, e \mid e_1\, e_2 \mid \langle e_1, e_2\rangle$
		$\mid$	$\Lambda\alpha.\, e \mid e\,\tau \mid co\, e$
Coercions	co	$::=$	$\mathsf{id} \mid co_1 \circ co_2 \mid \mathsf{top} \mid \mathsf{bot} \mid co_1 \to co_2$
		$\mid$	$\langle co_1, co_2\rangle \mid \pi_1 \mid \pi_2$
		$\mid$	$co_\forall \mid \mathsf{dist}_\to \mid \mathsf{top}_\to \mid \mathsf{top}_\forall \mid \mathsf{dist}_\forall$
Term contexts	Ψ	$::=$	$\bullet \mid \Psi, x : \tau$
Type contexts	Φ	$::=$	$\bullet \mid \Phi, \alpha$

Coherence Issue

The Problem

- During type-directed translation, intersections elaborate to pairs:

$$\Delta; \Gamma \vdash 1 \,,,\, \mathsf{True} \Rightarrow \mathsf{Int} \,\&\, \mathsf{Bool} \rightsquigarrow \langle 1, \mathsf{True} \rangle$$

$$\Delta; \Gamma \vdash (1 \,,,\, \mathsf{True}) : \mathsf{Int} \Rightarrow \mathsf{Int} \rightsquigarrow \pi_1 \langle 1, \mathsf{True} \rangle$$

$$\Delta; \Gamma \vdash (1 \,,,\, \mathsf{True}) : \mathsf{Bool} \Rightarrow \mathsf{Bool} \rightsquigarrow \pi_2 \langle 1, \mathsf{True} \rangle$$

$$\Delta; \Gamma \vdash 1 : \mathsf{Int} \,\&\, \mathsf{Int} \Rightarrow \mathsf{Int} \,\&\, \mathsf{Int} \rightsquigarrow \langle 1, 1 \rangle$$

The Problem

- During type-directed translation, intersections elaborate to pairs:

$$\Delta; \Gamma \vdash 1\,,,\,\mathsf{True} \Rightarrow \mathsf{Int}\,\&\,\mathsf{Bool} \rightsquigarrow \langle 1, \mathsf{True}\rangle$$

$$\Delta; \Gamma \vdash (1\,,,\,\mathsf{True}) : \mathsf{Int} \Rightarrow \mathsf{Int} \rightsquigarrow \pi_1\,\langle 1, \mathsf{True}\rangle$$

$$\Delta; \Gamma \vdash (1\,,,\,\mathsf{True}) : \mathsf{Bool} \Rightarrow \mathsf{Bool} \rightsquigarrow \pi_2\,\langle 1, \mathsf{True}\rangle$$

$$\Delta; \Gamma \vdash 1 : \mathsf{Int}\,\&\,\mathsf{Int} \Rightarrow \mathsf{Int}\,\&\,\mathsf{Int} \rightsquigarrow \langle 1, 1\rangle$$

- There can be multiple translations for one typing derivation:

$$\Delta; \Gamma \vdash (1 : \mathsf{Int}\,\&\,\mathsf{Int}) : \mathsf{Int} \Rightarrow \mathsf{Int} \rightsquigarrow \pi_1\,\langle 1, 1\rangle$$

$$\Delta; \Gamma \vdash (1 : \mathsf{Int}\,\&\,\mathsf{Int}) : \mathsf{Int} \Rightarrow \mathsf{Int} \rightsquigarrow \pi_2\,\langle 1, 1\rangle$$

Coherence

- Proof Strategy: semantic coherence
- We define a heterogeneous logical relation, called canonicity $(v_1, v_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!]$.

Canonicity Relation

$$
\begin{array}{rcl}
(v_1, v_2) \in \mathcal{V}[\![\mathsf{Int}; \mathsf{Int}]\!] & \triangleq & \exists i.\ v_1 = v_2 = i \\
(v_1, v_2) \in \mathcal{V}[\![\{l : A\}; \{l : B\}]\!] & \triangleq & (v_1, v_2) \in \mathcal{V}[\![A; B]\!] \\
(v_1, v_2) \in \mathcal{V}[\![A_1 \to B_1; A_2 \to B_2]\!] & \triangleq & \forall (v_2', v_1') \in \mathcal{V}[\![A_2; A_1]\!].\ (v_1\ v_1', v_2\ v_2') \in \mathcal{E}[\![B_1; B_2]\!] \\
(\langle v_1, v_2 \rangle, v_3) \in \mathcal{V}[\![A \,\&\, B; C]\!] & \triangleq & (v_1, v_3) \in \mathcal{V}[\![A; C]\!] \wedge (v_2, v_3) \in \mathcal{V}[\![B; C]\!] \\
(v_3, \langle v_1, v_2 \rangle) \in \mathcal{V}[\![C; A \,\&\, B]\!] & \triangleq & (v_3, v_1) \in \mathcal{V}[\![C; A]\!] \wedge (v_3, v_2) \in \mathcal{V}[\![C; B]\!] \\
(v_1, v_2) \in \mathcal{V}[\![\forall (\alpha * A_1).\ B_1; \forall (\alpha * A_2).\ B_2]\!] & \triangleq & \forall \bullet \vdash t * A_1 \,\&\, A_2.\ (v_1\ |t|, v_2\ |t|) \in \mathcal{E}[\![[t/\alpha]B_1; [t/\alpha]B_2]\!] \\
(v_1, v_2) \in \mathcal{V}[\![A; B]\!] & \triangleq & \text{true} \quad \text{otherwise} \\
(e_1, e_2) \in \mathcal{E}[\![A; B]\!] & \triangleq & \exists v_1, v_2.\ e_1 \longrightarrow^* v_1 \wedge e_2 \longrightarrow^* v_2\ \wedge (v_1, v_2) \in \mathcal{V}[\![A; B]\!]
\end{array}
$$

- The relation should relate values originating from non-disjoint intersection types, and is thus heterogeneous

Canonicity Relation

$$
\begin{array}{rcl}
(v_1, v_2) \in \mathcal{V}[\![\mathsf{Int}; \mathsf{Int}]\!] & \triangleq & \exists i.\, v_1 = v_2 = i \\
(v_1, v_2) \in \mathcal{V}[\![\{l : A\}; \{l : B\}]\!] & \triangleq & (v_1, v_2) \in \mathcal{V}[\![A; B]\!] \\
(v_1, v_2) \in \mathcal{V}[\![A_1 \to B_1; A_2 \to B_2]\!] & \triangleq & \forall (v_2', v_1') \in \mathcal{V}[\![A_2; A_1]\!].\, (v_1\ v_1', v_2\ v_2') \in \mathcal{E}[\![B_1; B_2]\!] \\
(\langle v_1, v_2 \rangle, v_3) \in \mathcal{V}[\![A \,\&\, B; C]\!] & \triangleq & (v_1, v_3) \in \mathcal{V}[\![A; C]\!] \wedge (v_2, v_3) \in \mathcal{V}[\![B; C]\!] \\
(v_3, \langle v_1, v_2 \rangle) \in \mathcal{V}[\![C; A \,\&\, B]\!] & \triangleq & (v_3, v_1) \in \mathcal{V}[\![C; A]\!] \wedge (v_3, v_2) \in \mathcal{V}[\![C; B]\!] \\
(v_1, v_2) \in \mathcal{V}[\![\forall (\alpha * A_1).\, B_1; \forall (\alpha * A_2).\, B_2]\!] & \triangleq & \forall \bullet \vdash t * A_1 \,\&\, A_2.\ (v_1\ |t|, v_2\ |t|) \in \mathcal{E}[\![[t/\alpha]B_1; [t/\alpha]B_2]\!] \\
(v_1, v_2) \in \mathcal{V}[\![A; B]\!] & \triangleq & \text{true} \quad \text{otherwise} \\
(e_1, e_2) \in \mathcal{E}[\![A; B]\!] & \triangleq & \exists v_1, v_2.\, e_1 \longrightarrow^* v_1 \wedge e_2 \longrightarrow^* v_2 \ \wedge (v_1, v_2) \in \mathcal{V}[\![A; B]\!]
\end{array}
$$

- The relation should relate values originating from non-disjoint intersection types, and is thus heterogeneous
- We must consider $(v_1, v_2) \in \mathcal{V}[\![\mathsf{Int}; \alpha]\!]$, where we need to substitute the type variables; but then the relation is ill-formed

Canonicity Relation

$$
\begin{array}{rcl}
(v_1, v_2) \in \mathcal{V}[\![\mathsf{Int}; \mathsf{Int}]\!] & \triangleq & \exists i.\, v_1 = v_2 = i \\
(v_1, v_2) \in \mathcal{V}[\![\{l : A\}; \{l : B\}]\!] & \triangleq & (v_1, v_2) \in \mathcal{V}[\![A; B]\!] \\
(v_1, v_2) \in \mathcal{V}[\![A_1 \to B_1; A_2 \to B_2]\!] & \triangleq & \forall (v_2', v_1') \in \mathcal{V}[\![A_2; A_1]\!].\, (v_1\ v_1', v_2\ v_2') \in \mathcal{E}[\![B_1; B_2]\!] \\
(\langle v_1, v_2 \rangle, v_3) \in \mathcal{V}[\![A \,\&\, B; C]\!] & \triangleq & (v_1, v_3) \in \mathcal{V}[\![A; C]\!] \wedge (v_2, v_3) \in \mathcal{V}[\![B; C]\!] \\
(v_3, \langle v_1, v_2 \rangle) \in \mathcal{V}[\![C; A \,\&\, B]\!] & \triangleq & (v_3, v_1) \in \mathcal{V}[\![C; A]\!] \wedge (v_3, v_2) \in \mathcal{V}[\![C; B]\!] \\
(v_1, v_2) \in \mathcal{V}[\![\forall(\alpha * A_1).\, B_1; \forall(\alpha * A_2).\, B_2]\!] & \triangleq & \forall \bullet \vdash t * A_1 \,\&\, A_2.\ (v_1\,|t|, v_2\,|t|) \in \mathcal{E}[\![[t/\alpha]B_1; [t/\alpha]B_2]\!] \\
(v_1, v_2) \in \mathcal{V}[\![A; B]\!] & \triangleq & \mathsf{true} \quad \text{otherwise} \\
(e_1, e_2) \in \mathcal{E}[\![A; B]\!] & \triangleq & \exists v_1, v_2.\, e_1 \longrightarrow^* v_1 \wedge e_2 \longrightarrow^* v_2 \ \wedge (v_1, v_2) \in \mathcal{V}[\![A; B]\!]
\end{array}
$$

- The relation should relate values originating from non-disjoint intersection types, and is thus heterogeneous
- We must consider $(v_1, v_2) \in \mathcal{V}[\![\mathsf{Int}; \alpha]\!]$, where we need to substitute the type variables; but then the relation is ill-formed
- For it to be well-formed, we restrict to the predicative subset of the type system

More in the paper

- Details about canonicity relation and coherence proof
- A complete and sound algorithmic type system
- Type-safety of F_{co}, and elaboration soundness of F_i^+ to F_{co}
- Haskell implementation

More in the paper

- Details about canonicity relation and coherence proof
- A complete and sound algorithmic type system
- Type-safety of F_{co}, and elaboration soundness of F_i^+ to F_{co}
- Haskell implementation

☞ Except some manual proofs of decidability, all proofs have been mechanically formalized in the Coq proof assistant!

Related Work and Conclusion

Related Work

	$\lambda_{,,}$	λ_i	$\lambda_\wedge^\vee$	λ_i^+	F_i	F_i^+
Disjointness	○	●	○	●	●	●
Unrestricted intersections	●	○	●	●	○	●
BCD subtyping	○	○	●	●	○	●
Polymorphism	○	○	○	○	●	●
Coherence	○	◑	○	●	◑	●
Bottom type	○	○	●	○	○	●

$\lambda_{,,}$ [Dunfield, 2014] λ_i [Oliveira et al., 2016] $\lambda_\wedge^\vee$ [Blaauwbroek, 2017]
λ_i^+ [Bi et al., 2018] F_i [Alpuim et al., 2017]

Conclusion

- F_i^+ is a type-safe and coherent calculus
- F_i^+ has disjoint intersection types, BCD subtyping and parametric polymorphism
- F_i^+ improves the state-of-art of compositional designs

References

João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. 2017. Disjoint Polymorphism. In *European Symposium on Programming (ESOP)*.

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A filter lambda model and the completeness of type assignment. *The journal of symbolic logic* 48, 04 (1983), 931–940.

Xuan Bi and Bruno C. d. S. Oliveira. 2018. Typed First-Class Traits. In *European Conference on Object-Oriented Programming (ECOOP)*.

Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2018. The Essence of Nested Composition. In *European Conference on Object-Oriented Programming (ECOOP)*.

Lasse Blaauwbroek. 2017. *On the Interaction Between Unrestricted Union and Intersection Types and Computational Effects*. Master's thesis. Technical University Eindhoven.

Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 05 (2009), 509.

Joshua Dunfield. 2014. Elaborating intersection and union types. *Journal of Functional Programming (JFP)* 24, 2-3 (2014), 133–165.

Jeremy Gibbons and Nicolas Wu. 2014. Folding domain-specific languages: deep and shallow embeddings (functional Pearl). In *ICFP*. ACM, 339–347.

Fabian Muehlboeck and Ross Tate. 2018. Empowering union and intersection types with integrated subtyping. In *OOPSLA*.

Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses. In *European Conference on Object-Oriented Programming (ECOOP)*.

Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint intersection types. In *International Conference on Functional Programming (ICFP)*.

Dana Scott. 1970. *Outline of a mathematical theory of computation*. Oxford University Computing Laboratory, Programming Research Group.

Dana S Scott and Christopher Strachey. 1971. *Toward a mathematical semantics for computer languages*. Vol. 1. Oxford University Computing Laboratory, Programming Research Group.

Q & A

- Thank you for listening!
- Find more about me: http://xnning.github.io
- Scan me for full paper:

Back up slides

Finally tagless

```haskell
{-# LANGUAGE ConstraintKinds #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TypeApplications #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE UndecidableInstances #-}
```