Applicative Intersection Types

Applicative Intersection Types

- It's more like a tutorial than a research report
- Function application in terms typed with intersection types
- Not directly related to applicative functor*

McBride, C., & Paterson, R. (2008). Applicative programming with effects. *Journal of functional programming*, *18*(1), 1-13.

Table of Contents

Table of Contents

Intersection type and Merge Operator

```
1 ,, true :: Int & Bool
```

- Intersection type allows a term that can have multiple types
- Merge Operator creates a term which is an inhabitant of intersection type

λ_i and Type Directed Operational Semantics*

- λ_i is a calculus with intersection types and merge operator
- λ_i has a direct call-by-value operational semantics
- λ_i uses type annotation to guide reduction then further reduces value

Huang, Xuejing, and Bruno C. D. S. Oliveira. "A Type-Directed Operational Semantics For a Calculus with a Merge Operator." *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

λ_i and Type Directed Operational Semantics*

```
succ          :: Int → Int
not           :: Bool → Bool
succ ,, not   :: (Int → Int) & (Bool → Bool)
```

```
(succ ,, not) : (Int → Int) ~→ succ
(succ ,, not) : (Bool → Bool) ~→ not
```

```
(succ ,, not) : (Int → Bool)  ~→ type check error!
```

Table of Contents

Limitations in λ_i

```
((succ ,, not) : Bool → Bool) true
⟿ not true
⟿ false
```

```
(λf. f 1 : (Int → Int) → Int) ((succ ,, not) : Int → Int)
⟿ (λf. f 1 : (Int → Int) → Int) succ
⟿ succ 1
⟿ 2
```

Limitations in λ_i

```
((succ ,, not) : Bool → Bool) true
~→ not true
~→ false
```

we already knew the information from arguments!!

```
(λf. f 1 : (Int → Int) → Int) ((succ ,, not) : Int → Int)
~→ (λf. f 1 : (Int → Int) → Int) succ
~→ succ 1
~→ 2
```

we already knew the information from function!!

Programmers' Intuition

```
(succ ,, not) true
~~> not true
~~> false
```

automatically pick during compilation

```
(λf. f 1 : (Int → Int) → Int) (succ ,, not)
~~> (λf. f 1 : (Int → Int) → Int) succ
~~> succ 1
~~> 2
```

Table of Contents

Ideas

```
(succ ,, not) 3

     Int
```

- instead of using type annotation, we obtain types from arguments*

Xie, Ningning, and Bruno C. D. S. Oliveira. "Let Arguments Go First." *European Symposium on Programming*. Springer, Cham, 2018.

Bi-Directional Type Checking

- Infer mode : $\Gamma \vdash e \Rightarrow A$
- Check mode: $\Gamma \vdash e \Leftarrow A$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \rightarrow B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\ e_2 \Rightarrow B}\ \text{T-App}$$

Application mode: $\Gamma \mid \Psi \vdash e \Rightarrow A$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \rightarrow B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\ e_2 \Rightarrow B}\ \text{T-App}$$

Becomes

Ψ is a argument stack!

$$\frac{\Gamma \vdash e_2 \Rightarrow A \qquad \Gamma \mid \Psi, A \vdash e_1 \Rightarrow A \rightarrow B}{\Gamma \vdash e_1\ e_2 \Rightarrow B}\ \text{T-App}$$

Case Study

type check without type reconstruction!

$$\frac{\Gamma \vdash e_2 \Rightarrow A \qquad \Gamma \mid \Psi, A \vdash e_1 \Rightarrow A \to B}{\Gamma \vdash e_1\ e_2 \Rightarrow B}\ \text{T-App}$$

```
   1 ⇒ Int             Int ⊢ (λx. x) ⇒ Int → Int
----------------------------------------------------- T-App
          (λx. x) 1 ⇒ Int
```

Case Study

$$\frac{\Gamma \vdash e_2 \Rightarrow A \qquad \Gamma \mid \Psi, A \vdash e_1 \Rightarrow A \to B}{\Gamma \vdash e_1\ e_2 \Rightarrow B}\ \text{T-App}$$

```
true ⇒ Bool          Bool ⊢ (succ ,, not) ⇒ Bool → Bool
----------------------------------------------------------- T-App
             (succ ,, not) true ⇒ Bool
```

Case Study

we need an <u>algorithm</u> to pick a type from intersection type!

$$\frac{\Gamma \vdash e_2 \Rightarrow A \qquad \Gamma \mid \Psi, A \vdash e_1 \Rightarrow A \rightarrow B}{\Gamma \vdash e_1\ e_2 \Rightarrow B}\ \text{T-App}$$

```
true ⇒ Bool         Bool ⊢ (succ ,, not) ⇒ Bool → Bool
----------------------------------------------------- T-App
             (succ ,, not) true ⇒ Bool
```

Application Subtyping

$$\frac{}{\varnothing \vdash A \leq A}\ \text{AS-Refl}$$

$$\frac{C \leq A \qquad \Psi \vdash B \leq D}{\Psi, C \vdash A \to B \leq C \to D}\ \text{AS-Fun}$$

$$\frac{\Psi, C \vdash A \leq D}{\Psi, C \vdash A \,\&\, B \leq D}\ \text{AS-L}$$

$$\frac{\Psi, C \vdash B \leq D}{\Psi, C \vdash A \,\&\, B \leq D}\ \text{AS-R}$$

Application Subtyping

$$\frac{}{\varnothing \vdash A \leq A}\ \text{AS-Refl}$$

$$\frac{C \leq A \qquad \Psi \vdash B \leq D}{\Psi, C \vdash A \rightarrow B \leq C \rightarrow D}\ \text{AS-Fun}$$

```
-------------- Sub-Refl  ---------------------- As-Refl
Bool <: Bool                . ⊢ Bool <: Bool
------------------------------------------------ As-Fun
Bool ⊢ Bool → Bool <: Bool → Bool
----------------------------------------------------------- AS-R
Bool ⊢ (Int → Int) & (Bool → Bool) <: Bool → Bool
```

Metatheory of Application Subtyping

- ($\Psi \to B$) If $\Psi = A_1, A_2, \ldots, A_n$, then $\Psi \to B$ means the arrow type $A_n \to \ldots \to A_2 \to A_1 \to B$
- ($\Psi \vdash$ <: **to** <:) If $\Psi \vdash A <: B$, then $A <: B$
- (<: **to** $\Psi \vdash$ <:) If $A <: \Psi \to B_1$, then $\exists B_2, \Psi \vdash A <: \Psi \to B_2$ and $B_2 <: B_1$

Case Study

$$\frac{\Gamma \mid \Psi \vdash e_1 ,, e_2 \Rightarrow B \qquad \Psi, A \vdash B \leq C}{\Gamma \mid \Psi, A \vdash e_1 ,, e_2 \Rightarrow C} \text{ T-Merge-Pick}$$

```
                    ⊢ (succ ,, not) ⇒ (Int → Int) & (Bool → Bool)
               Bool ⊢ (Int → Int) & (Bool → Bool) ⇒ Bool → Bool
              ------------------------------------------------------ T-Merge-Pick
true ⇒ Bool         Bool ⊢ (succ ,, not) ⇒ Bool → Bool
--------------------------------------------------------------------- T-App
                     (succ ,, not) true ⇒ Bool
```

Table of Contents

Challenge I: Ambiguity

```
Int ⊢ (Int → Int) <: (Int → Int)
------------------------------------------------------ As-L
Int ⊢ (Int → Int) & (Int → Bool) <: (Int → Int)


Int ⊢ (Int → Bool) <: (Int → Bool)
------------------------------------------------------ As-R
Int ⊢ (Int → Int) & (Int → Bool) <: (Int → Bool)
```

Challenge I: Ambiguity

type with same inputs should be rejected!

```
Int ⊢ (Int → Int) <: (Int → Int)
--------------------------------------------------- As-L
Int ⊢ (Int → Int) & (Int → Bool) <: (Int → Int)


Int ⊢ (Int → Bool) <: (Int → Bool)
--------------------------------------------------- As-R
Int ⊢ (Int → Int) & (Int → Bool) <: (Int → Bool)
```

Solution I: Ambiguity

$$\frac{\Psi, C \vdash A \leq D \qquad \textit{not}\ (B\ \textit{in}\ \textit{NextInputs}(A))}{\Psi, C \vdash A \,\&\, B \leq D}\ \text{AS-L}$$

$$\frac{\Psi, C \vdash B \leq D \qquad \textit{not}\ (A\ \textit{in}\ \textit{NextInputs}(B))}{\Psi, C \vdash A \,\&\, B \leq D}\ \text{AS-R}$$

Challenge II: Semantics

$$\frac{\Gamma \mid \Psi \vdash e_1 ,, e_2 \Rightarrow B \qquad \Psi, A \vdash B \leq C}{\Gamma \mid \Psi, A \vdash e_1 ,, e_2 \Rightarrow C}\ \text{T-Merge-Pick}$$

in some process of function application

Semantics should mimic the typing

Solution II: Semantics

$$\frac{ptype(vl) \vdash ptype(v_1,,v_2) \leq ptype(v_1) \qquad v_1\ vl \rightsquigarrow e}{v_1,,v_2\ vl \rightsquigarrow e}\ \text{App-Merge-L}$$

$$\frac{ptype(vl) \vdash ptype(v_1,,v_2) \leq ptype(v_2) \qquad v_2\ vl \rightsquigarrow e}{v_1,,v_2\ vl \rightsquigarrow e}\ \text{App-Merge-R}$$

Challenges III: Syntax and value

- Value must carry some type annotation to preserve in typed reduction
- Annotating value would trigger typed reduction
- We need information of types in reduction
- Narrowing lemmas has conflicts with unannotated lambda

$$\frac{v \rightsquigarrow A\ v'}{v : A \rightsquigarrow v'}\ \text{Step-Anno-V}$$

Solution III: Syntax and value

No

$$p := T \mid N \mid \lambda x\,.\,x \mid p_1, , p_2$$
$$v := p : A \mid \lambda x\,.\,x$$

No

$$p := T \mid N \mid \lambda x\,.\,x$$
$$v := p : A \mid v_1, , v_2$$
$$r := v \mid \lambda x\,.\,x$$

Yes!

$$p := N \mid \lambda x : A\,.\,x$$
$$v := p : A \mid v_1, , v_2$$

Solution III: Value

- `1 : Top` behaves as a term Top
- `λx : Int . x` is not a value
- `λx : Int . x : (Int → Int)` is a value
- `1 : Int ,, true : Bool` is a value

Challenge IV: Meta-theory

- Current lemmas between Application Subtyping and Subtyping is not strong enough
- $\Gamma \mid \Psi \vdash e \Rightarrow A$ is generalized from $\Gamma \vdash e \Rightarrow A$, but properties between two formally and intuitively are still undiscovered
- Preservation / Progress re-statement

Unresolved IV: Meta-theory

- If $\Gamma \mid \Psi, A \vdash e \Leftrightarrow A \to B$, then $\Gamma \mid \Psi \vdash e \Leftarrow A \to B$
- If $\Gamma \mid \varnothing \vdash v \Rightarrow A$ and $\Psi \vdash A <: B$, then $\Gamma \mid \Psi \vdash v \Rightarrow B$

Table of Contents

Meta-theorist's Perspective

- It's a kind of type inference using bidirectional typing
- It combines application mode and check mode in λ_i

Meta-theorist's Perspective

- It's a kind of type inference using bidirectional typing
- It combines application mode and check mode in λ_i

retain the possibility of writing type annotations!

Programmer's Perspective

- It reduces some unnecessary typing annotations
- It enables function overloading in a type-safe manner
- It improves efficiency in some sense

Programmer's Perspective

```
succ ,, not 4 -- type checks
succ ,, not 'c' -- doesn't type check
(f: Int → Int) ,, (g : Int → Bool) 1 -- ambiguity error
```

- It reduces some unnecessary typing annotations
- It enables function overloading in a type-safe manner
- It improves efficiency in some sense **Less runtime dispatch!**

Thanks for listening,

and hope you enjoy!