Applicative Intersection Types

January 10, 2023

Xu Xue (MPhil Candidate)

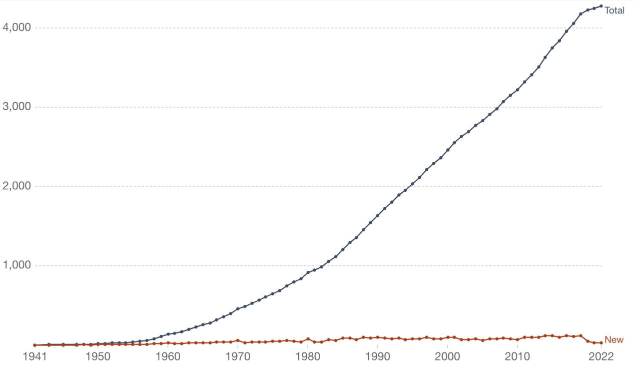
The University of Hong Kong

The Trend is ...

New languages keep being invented!

The Trend is ...

New languages keep being invented!



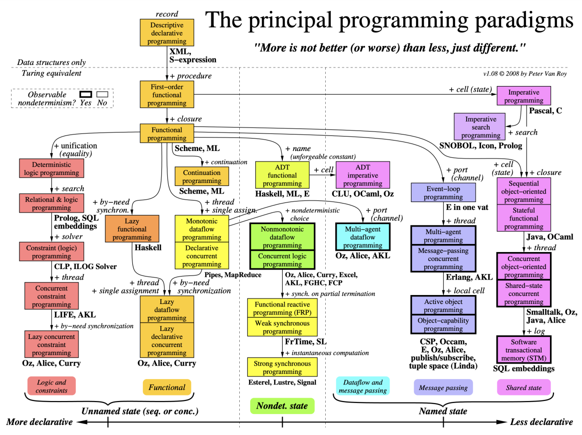
credit: pldb.com

The Trend is ...

New features keep being discovered!

The Trend is ...

New features keep being discovered!



The Problem is ...

- New languages are prototyped *by a small/core calculus*;
- New features are often studied *in an isolated environment*;

And...

The Problem is ...

- New languages are prototyped *by a small/core calculus*;
- New features are often studied *in an isolated environment*;

And...

- Features are not orthogonal;
- Languages are not designed at once.

A general framework

- contains wide features;
- retains simplicity;
- has extensibility;
- and enjoys good properties,

A general framework

- contains wide features;
- retains simplicity;
- has extensibility;
- and enjoys good properties,

is *desired* by language designers and implementors.

Intersection Types is a nice fit

*"our goal is to use **intersections** and unions as general mechanisms for encoding language features, so we really should do it in full generality, or not at all..."*[1]

[1] Jana Dunfield. "Elaborating intersection and union types". In: *Journal of Functional Programming* 24.2-3 (2014), pp. 133–165.

Intersection Types

- A term e having the type $A \& B$ means e has both A and B.

[2] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. "Functional characters of solvable terms". In: *Mathematical Logic Quarterly* 27.2-6 (1981), pp. 45–58.

Intersection Types

- A term e having the type $A \,\&\, B$ means e has both A and B.
- Originally introduced by Coppo et al.[2], it allows $\lambda x.\, x\, x$ to be typed $((A \to B) \,\&\, A) \to B$.

[2] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. “Functional characters of solvable terms”. In: *Mathematical Logic Quarterly* 27.2-6 (1981), pp. 45–58.

Intersection Types

- A term e having the type $A \mathbin{\&} B$ means e has both A and B.
- Originally introduced by Coppo et al.[2], it allows $\lambda x.\, x\, x$ to be typed $((A \to B) \mathbin{\&} A) \to B$.
- In languages like TypeScript, the intersection types are explicitly inhabitated.

```
interface Name { name: string; }
interface ID { id: number; }
type Person = Name & ID
let e : Person = { id: 42, name: 'Alice'};
```

[2] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. "Functional characters of solvable terms". In: *Mathematical Logic Quarterly* 27.2-6 (1981), pp. 45–58.

Merge Operator[4]

- $e_1 \,,\, e_2$ means it can be used as e_1 or e_2.

[3] Bidirectional typing, $\Gamma \vdash e \Leftrightarrow A$, and $\Leftrightarrow ::= \Leftarrow | \Rightarrow$. $\Leftarrow$ is to check; $\Rightarrow$ is to infer.

[4] Jana Dunfield. "Elaborating intersection and union types". In: *Journal of Functional Programming* 24.2-3 (2014), pp. 133–165.

Merge Operator[4]

- $e_1 ,, e_2$ means it can be used as e_1 or e_2.
- Force intersection types to be *explicitly* introduced and inhabitated.
- Typing for merge is [3]

$$\frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B}{\Gamma \vdash e_1 ,, e_2 \Rightarrow A \,\&\, B}\ \textsc{T-Mrg}$$

[3] Bidirectional typing, $\Gamma \vdash e \Leftrightarrow A$, and $\Leftrightarrow ::= \Leftarrow | \Rightarrow$. $\Leftarrow$ is to check; $\Rightarrow$ is to infer.

[4] Jana Dunfield. "Elaborating intersection and union types". In: *Journal of Functional Programming* 24.2-3 (2014), pp. 133–165.

Merge Operator[4]

- $e_1 ,, e_2$ means it can be used as e_1 or e_2.
- Force intersection types to be *explicitly* introduced and inhabitated.
- Typing for merge is [3]

$$\textsc{T-Mrg}\quad \frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B}{\Gamma \vdash e_1 ,, e_2 \Rightarrow A \,\&\, B}$$

- Merge operator adds expressive power and enables many applications.

[3] Bidirectional typing, $\Gamma \vdash e \Leftrightarrow A$, and $\Leftrightarrow ::= \Leftarrow \mid \Rightarrow$. $\Leftarrow$ is to check; $\Rightarrow$ is to infer.

[4] Jana Dunfield. "Elaborating intersection and union types". In: *Journal of Functional Programming* 24.2-3 (2014), pp. 133–165.

Extensible Records[5]

- Records can be represented by *syntactic sugar of merge operator*.
- $\{x = e_1, y = e_2, z = e_3\}$ can be viewed as $\{x = e_1\}, , \{y = e_2\}, , \{z = e_3\}$.

[5] Luca Cardelli and John C Mitchell. "Operations on records". In: *Mathematical structures in computer science* 1.1 (1991), pp. 3–48.

Extensible Records[5]

- Records can be represented by *syntactic sugar of merge operator*.
- $\{x = e_1, y = e_2, z = e_3\}$ can be viewed as $\{x = e_1\}, , \{y = e_2\}, , \{z = e_3\}$.
- Record width subtyping *for free*.

$$\{l_i : T_i\}^{i=1..n..n+k} <: \{l_i : T_i\}^{1..n}$$

is subsumed by

$$\{l_1 : A\} \,\&\, \{l_2 : B\} <: \{l_1 : A\}$$

is subsumed by

$$A \,\&\, B <: A$$

[5] Luca Cardelli and John C Mitchell. "Operations on records". In: *Mathematical structures in computer science* 1.1 (1991), pp. 3–48.

Record Projection

- Record Projection is standard.

$$(\{x = e_1\} ,, \{y = e_2\}).x \hookrightarrow e_1$$

$$(\{x = e_1\} ,, \{y = e_2\}).y \hookrightarrow e_2$$

- Record Concatenation is simply merging.

$$(\{x = e_1\} ,, \{y = e_2\}) ,, \{z = e_3\}$$

Overloaded Functions[6]

- Function implementation *varies* depending on the types of arguments.

[6] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. "A calculus for overloaded functions with subtyping". In: *Information and Computation* 117.1 (1995), pp. 115–135.

Overloaded Functions[6]

- Function implementation *varies* depending on the types of arguments.
- Consider Haskell's `show` function.

```
show :: Show a => a -> String
instance Show Int where
  show = showInt
instance Show Bool where
  show = showBool
-- instance will be selected according to the argument type
show 1 ↪ showInt 1 ↪ "1"
show true ↪ showBool true ↪ "true"
```

- `show` can be defined as `showInt,,showBool`

[6] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. "A calculus for overloaded functions with subtyping". In: *Information and Computation* 117.1 (1995), pp. 115–135.

Overloaded Application

- Overloaded Application is standard.

  ```
  show : (Int -> String) & (Bool -> String)
  show = showInt,,showBool
  show 1 ↪ showInt 1 ↪ "1"
  show true ↪ showBool true ↪ "true"
  ```

- Adding overloading instances is simply by merging.

  ```
  newShow = show,,showDouble
  ```

Return type Overloading[7]

- Function implementation varies depending on the surrounding contexts.

[7] Koar Marntirosian et al. “Resolution as Intersection Subtyping via Modus Ponens”. In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020).

Return type Overloading[7]

- Function implementation varies depending on the surrounding contexts.
- Consider Haskell's `read` function

```
read :: Read a => String -> a
instance Read Int where
  read = readInt
instance Read Bool where
  read = readBool
-- instance will be selected according to surrounding contexts
succ (read "1") ↪ succ (readInt "1") ↪ 2
not (read "true") ↪ succ (readBool "1") ↪ false
```

[7] Koar Marntirosian et al. "Resolution as Intersection Subtyping via Modus Ponens". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020).

Return type Overloading[7]

- Function implementation varies depending on the surrounding contexts.
- Consider Haskell's `read` function

```
read :: Read a => String -> a
instance Read Int where
  read = readInt
instance Read Bool where
  read = readBool
-- instance will be selected according to surrounding contexts
succ (read "1") ↪ succ (readInt "1") ↪ 2
not (read "true") ↪ succ (readBool "1") ↪ false
```

- Calculi with merge operator can do in a similar way.

```
read = readInt,,readBool
```

[7] Koar Marntirosian et al. "Resolution as Intersection Subtyping via Modus Ponens". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020).

Nested Composition[8]

- It reflects *distributivity* of intersection types at the term level.

$$\{l : A\} \,\&\, \{l : B\} <: \{l : A \,\&\, B\} \quad \text{S-Distri-Rcd}$$

$$(A \rightarrow B) \,\&\, (A \rightarrow C) <: A \rightarrow (B \,\&\, C) \quad \text{S-Distri-Arr}$$

- Results extracted from <u>nested</u> terms will be <u>composed</u> when eliminating terms created by the merge operator.

[8] Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. "The essence of nested composition". In: *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.

Nested Composition via Projection and Application

- For records

$$(\{x = e_1\},,\{x = e_2\}).x \hookrightarrow e_1,,e_2$$

Nested Composition via Projection and Application

- For records

$$(\{x = e_1\},, \{x = e_2\}).x \hookrightarrow e_1,, e_2$$

- For overloaded functions

$$f : Int \longrightarrow Int \longrightarrow Int$$
$$g : Int \longrightarrow Bool \longrightarrow Bool$$
$$(f,, g)\, 1 \hookrightarrow (f\, 1),, (g\, 1)$$

Nested Composition via Projection and Application

- For records

$$(\{x = e_1\} ,, \{x = e_2\}).x \hookrightarrow e_1 ,, e_2$$

- For overloaded functions

$$f : Int \to Int \to Int$$
$$g : Int \to Bool \to Bool$$
$$(f ,, g)\, 1 \hookrightarrow (f\, 1) ,, (g\, 1)$$

- Both cases are "unnatural"
  since we allow repeated labels and ambiguous overloaded application.

Goodness of Nested Composition

- *[Nested record composition]* Key feature of *Compositional Programming*[9].
    - solves the Expression Problem naturally.
    - models forms of family polymorphism.

[9] Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. "Compositional Programming". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43.3 (2021), pp. 1–61.

Goodness of Nested Composition

- *[Nested record composition]* Key feature of *Compositional Programming*[9].
  - solves the Expression Problem naturally.
  - models forms of family polymorphism.
- *[Nested function composition]* It enables *first-class curried overloaded functions*.
  - overloaded functions are default curried;
  - we can abstract and return overloaded functions in a flexible way;
  - it's a novel and interesting finding in this work.

[9] Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. "Compositional Programming". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43.3 (2021), pp. 1–61.

Challenges in Type Inference

In traditional calculi, we have the following typing rule for application:

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

This does not apply to case `show 1`, where

$$\frac{\Gamma \vdash show \Rightarrow A \,\&\, B \qquad \Gamma \vdash 1 \Leftarrow ?}{\Gamma \vdash show\, 1 \Rightarrow ?} \text{ T-App}$$

Challenges in Type Inference

A direct method is to:

1. assume we have the argument type A;
2. assume the type of function to be a intersection of function types:

$$(A_1 \longrightarrow B_1) \,\&\, (A_2 \longrightarrow B_2) \,\&\, \ldots \,\&\, (A_n \longrightarrow B_n)$$

Challenges in Type Inference

A direct method is to:

1. assume we have the argument type A;
2. assume the type of function to be a intersection of function types:

$$(A_1 \rightarrow B_1) \,\&\, (A_2 \rightarrow B_2) \,\&\, \ldots \,\&\, (A_n \rightarrow B_n)$$

3. then iterate intersection types by comparing the argument type A and input type A_i;

Challenges in Type Inference

A direct method is to:

1. assume we have the argument type A;
2. assume the type of function to be a intersection of function types:

$$(A_1 \longrightarrow B_1) \,\&\, (A_2 \longrightarrow B_2) \,\&\, \ldots \,\&\, (A_n \longrightarrow B_n)$$

3. then iterate intersection types by comparing the argument type A and input type A_i;
4. compose the outputs as the result type

Challenges in Dynamic Semantics

A direct method is to:

1. assume the overloaded function to be a merge of functions,

Challenges in Dynamic Semantics

A direct method is to:

1. assume the overloaded function to be a merge of functions,
2. then select correct instances according to the types.

Challenges in Dynamic Semantics

A direct method is to:

1. assume the overloaded function to be a merge of functions,
2. then select correct instances according to the types.
    - call-by-value strategy
    - type-dependent semantics

Distributivity Breaks the Assumptions

```
pshow : Unit -> (Int -> String) & (Bool -> String)
pshow = λx. show
pshow unit 1 ↪ "1"
pshow unit true ↪ "true"
```

Distributivity Breaks the Assumptions

```
pshow : Unit -> (Int -> String) & (Bool -> String)
pshow = λx. show
pshow unit 1 ↪ "1"
pshow unit true ↪ "true"
```

- `pshow` is **not** a merge of functions (wrapped in a lambda);
- its type is **not** a intersection of function types;
- it's still treated as an overloaded function.

Re-interpret Subtyping

We can have two interpretations of $A <: B \rightarrow C$:

- Suppose A, B and C are given, we tell whether the subtyping holds.

$$(Int \rightarrow String) \,\&\, (Bool \rightarrow String) <: Int \rightarrow String$$

- Suppose A and B are given, we infer the result type C[10].

$$(Int \rightarrow String) \,\&\, (Bool \rightarrow String) <: Int \rightarrow ?$$

[10] which is also the type of overloaded application.

Applicative Subtyping

$A \ll S$ is a specialized subtyping used to infer the type of applications and projections [11].

$$
\begin{aligned}
A_1 \to A_2 \ll B &= A_2 && \textit{when } B <: A_1 && (1)\\
A_1 \to A_2 \ll B &= . && \textit{when } \neg(B <: A_1) && (2)\\
\{l = A\} \ll l &= A && && (3)\\
\{l_1 = A\} \ll l_2 &= . && \textit{when } l_1 \neq l_2 && (4)\\
A_1 \,\&\, A_2 \ll S &= (A_1 \ll S) \circledcirc (A_2 \ll S) && && (5)\\
A \ll S &= . && \textit{otherwise} && (6)
\end{aligned}
$$

[11] $S ::= A \mid l$, Selector S is either type A or label l

Examples of Applicative Subtyping

```
show 1
```

$$
\begin{aligned}
& (Int \to String) \,\&\, (Bool \to String) \ll Int \\
\text{by } (5) \hookrightarrow\ & (Int \to String) \ll Int \circledcirc (Bool \to String) \ll Int \\
\text{by } (1)\ (2) \hookrightarrow\ & String \circledcirc .
\end{aligned}
$$

```
read "1"
```

$$
\begin{aligned}
& (String \to Int) \,\&\, (String \to Bool) \ll String \\
\text{by } (5) \hookrightarrow\ & (String \to Int) \ll String \circledcirc (String \to Bool) \ll String \\
\text{by } (1) \hookrightarrow\ & Int \circledcirc Bool
\end{aligned}
$$

Composition Operators

One version that implements *nested composition semantics* [12].

$$
\begin{aligned}
\bullet \circledcirc \bullet &= \bullet \\
A_1 \circledcirc \bullet &= A_1 \\
\bullet \circledcirc A_2 &= A_2 \\
A_1 \circledcirc A_2 &= A_1 \,\&\, A_2
\end{aligned}
$$

[12] We have another version of the operator which models the overloading semantics

Examples (applying nested composition semantics)

$$
\begin{aligned}
(\mathit{Int} \to \mathit{String}) \,\&\, (\mathit{Bool} \to \mathit{String}) &\ll \mathit{Int} &&= \mathit{String} \\
(\mathit{String} \to \mathit{Int}) \,\&\, (\mathit{String} \to \mathit{Bool}) &\ll \mathit{String} &&= \mathit{Int} \,\&\, \mathit{Bool} \\
\{x : \mathit{String}\} \,\&\, \{y : \mathit{String}\} &\ll y &&= \mathit{String}
\end{aligned}
$$

Let arguments go "together"

We infer both the type of function (merges) and argument together and then compute.

$$\frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B \qquad \boxed{A \ll B = C}}{\Gamma \vdash e_1\, e_2 \Rightarrow C}\ \textsc{T-App}$$

Examples (applying nested composition semantics)

We assume Γ is $f : I \rightarrow I \rightarrow I, g : I \rightarrow B \rightarrow B$.[13]

$$\dfrac{\dfrac{\Gamma \vdash (f\,,,\,g) \Rightarrow (I \rightarrow I \rightarrow I) \,\&\, (I \rightarrow B \rightarrow B) \qquad \Gamma \vdash 2 \Rightarrow I}{\Gamma \vdash (f\,,,\,g)\; 2 \Rightarrow \boxed{(I \rightarrow I) \,\&\, (B \rightarrow B)}}\;\textsc{T-App} \qquad \Gamma \vdash \mathit{true} \Rightarrow B}{\Gamma \vdash (f\,,,\,g)\; 2\; \mathit{true} \Rightarrow \boxed{B}}\;\textsc{T-App}$$

1. $f\,,,\,g$
2. $(f\,,,\,g)\; 2$
3. $(f\,,,\,g)\; 2\; \mathit{true}$

[13] I stands for *Int*, B stands for *Bool*.

Metatheory

$$
\begin{aligned}
(Int \to String) \,\&\, (Bool \to String) &\ll Int &&= String \\
(String \to Int) \,\&\, (String \to Bool) &\ll String &&= Int \,\&\, Bool \\
\{x : String\} \,\&\, \{y : String\} &\ll y &&= String
\end{aligned}
$$

$$
\begin{aligned}
(Int \to String) \,\&\, (Bool \to String) &<: Int \to String \\
(String \to Int) \,\&\, (String \to Bool) &<: String \to Int \,\&\, Bool \\
\{x : String\} \,\&\, \{y : String\} &<: \{y : String\}
\end{aligned}
$$

Metatheory

Lemma (Soundness (Function))

If $A \ll B = C$, *then* $A <: B \to C$.

Lemma (Completeness (Function))

If $A <: B \to C$, *then* $\exists D, A \ll B = D \wedge D <: C$.

Calculi Syntax

$$
\begin{array}{ll}
\text{Expressions} & e ::= x \mid i \mid e : A \mid e_1\, e_2 \mid \lambda x\,.e : A \to B \mid e_1 ,, e_2 \mid \{l = e\} \mid e.l \\
\text{Raw Values} & p ::= i \mid \lambda x\,.e : A \to B \\
\text{Values} & v ::= \boxed{p : A^o \mid v_1 ,, v_2 \mid \{l = v\}} \\
\text{Contexts} & \Gamma ::= \cdot \mid \Gamma, x : A
\end{array}
$$

- Values carry extra annotations as runtime types;
- The dispatching is based on runtime types;
- The restriction on runtime types settles a canonical form of overloaded functions.

Operational Semantics

$$\textsc{Step-App}\quad \frac{(v_1 \bullet v_2) \hookrightarrow e}{v_1\, v_2 \longmapsto e}$$

$$\textsc{Step-Prj}\quad \frac{(v \bullet l) \hookrightarrow v'}{v.l \longmapsto v'}$$

Applicative Dispatching[14]

$\boxed{(v \bullet vl) \hookrightarrow e}$ *(Applicative Dispatching)*

$$\textsc{App-Lam}\quad \frac{v \longmapsto_A v'}{((\lambda x.\, e : A \to B) : C \to D \bullet v) \hookrightarrow e[x \mapsto v'] : D}$$

$$\textsc{App-Proj}\quad \frac{}{(\{l = v\} \bullet l) \hookrightarrow v}$$

$$\textsc{App-Mrg-L}\quad \frac{\langle v_2 \rangle \ll \langle vl \rangle = . \qquad (v_1 \bullet vl) \hookrightarrow e}{((v_1 \,,, v_2) \bullet vl) \hookrightarrow e}$$

$$\textsc{App-Mrg-R}\quad \frac{\langle v_1 \rangle \ll \langle vl \rangle = . \qquad (v_2 \bullet vl) \hookrightarrow e}{((v_1 \,,, v_2) \bullet vl) \hookrightarrow e}$$

$$\textsc{App-Mrg-P}\quad \frac{\langle v_1 \rangle \ll \langle vl \rangle \neq . \qquad \langle v_2 \rangle \ll \langle vl \rangle \neq . \qquad (v_1 \bullet vl) \hookrightarrow e_1 \qquad (v_2 \bullet vl) \hookrightarrow e_2}{((v_1 \,,, v_2) \bullet vl) \hookrightarrow e_1 \,,, e_2}$$

[14] $\langle v \rangle$ extracts the runtime type of v

Type Soundness and Determinism[15]

Theorem (Preservation)

If $\cdot \vdash e \Leftrightarrow A$ *and* $e \longmapsto e'$, *then* $\cdot \vdash e' \Leftarrow A$.

Theorem (Progress)

If $\cdot \vdash e \Leftrightarrow A$, *then* e *is a value or* $\exists e', e \longmapsto e'$.

Theorem (Determinism)

If e *is well-typed,* $e \longmapsto e_1$ *and* $e \longmapsto e_2$, *then* $e_1 = e_2$.

[15] held only in calculus with disjointness

Interpreter Implementation

- Statically typed;
- A dialect of Lisp;
- 382 Lines of Racket Code:
  - S-expression parsing included;
  - Contract-based runtime check;

Language Tour (1/3)

```
;; simple literals
42 42.2 #t #f

;; lambda abstraction
(λ (x : int) x int)

;; function application
((λ (x : int) x int) 1)
;; => (: 1 int)

;; annotate a "value" can force a downcast/upcast
(: (: 1 int)
   (& int int)) ;; => duplicate a number
;; => (m (: 1 int) (: 1 int))
```

Language Tour (2/3)

```
;; merge two values
(m 1 #t)

;; merge two functions
(m (λ (x : int) x int)
   (λ (x : bool) x bool))

;; merged function can be applied
((m (λ (x : int) x int)
    (λ (x : bool) x bool))
 1)
;; => (: 1 int)
```

Language Tour (3/3)

```
;; use int+ to add integers
(int+ 1 3)

;; use flo+ to add floats
(flo+ 1.0 2.1)

;; overload int+ and flo+ to create a polymorphic "double" function
((m (λ (x : int) (int+ x x) int)
    (λ (x : float) (flo+ x x) float))
 1)
;; => (: 2 int)
```

Conlusion

- Applicative Subtyping & Applicative Dispatching
  - Three Variants of Subtyping
  - Sound/Complete Lemmas
- Formalisation of Two Calculi Design
  - Type Sound Calculus with an Unrestricted Merge Operator
  - Deterministic Calculus with a Disjoint Merge Operator
- Coq Formalisation & Interpreter Implementation
  - https://github.com/juniorxxue/applicative-intersection

Future Work

- Application Mode
  - Alternative to Applicative Subtyping;
- Bidirectional Typing
  - Recover the advantage of check-mode;
- "Best-Match" Evaluation Strategy
- Compile to Racket
  - Static Type Checking and Resolution using Macro System