TFP'22


# Named Arguments as Records

Yaozhu Sun and Bruno C. d. S. Oliveira

18 March 2022

# Why do we need argument names?

## Can you tell source from destination?

- cp file1 file2 

- memcpy(array1, array2, length) 
- Array.Copy(array1, array2, length) 
- copy(array1, array2) 

- mov rax, rbx 


movq %rbx, %rax 

# Why do we need argument names?

## source in green / destination in orange

- cp file1 file2 

- memcpy(array1, array2, length) 
- Array.Copy(array1, array2, length) 
- copy(array1, array2) 
- mov rax, rbx 


movq %rbx, %rax 

copy(array1, array2)

copy(***to:*** array1, ***from:*** array2)

copy(***from:*** array2, ***to:*** array1)

# Named arguments in some languages

## Smalltalk, Objective-C, and Swift

- (n odd) ifTrue: [ "n is odd" ] ifFalse: [ "n is even" ]

- [mutableDictionary setValue:@"Asia/Hong_Kong" forKey:@"tz"]

- DateComponents(year: 2022, month: 3, day: 18, hour: 20, minute: 5)
  DateComponents(year: 2022, month: 3, day: 18)
  DateComponents(year: 2022, weekday: 6, weekOfYear: 12)

# Named arguments in some other languages

## Python and Ruby

```python
def exp(x, base=math.e):
  return base ** x

exp(10) #= exp(x=10) = 22026
exp(base=2, x=10)   #=  1024
```

```ruby
def exp(x:, base: Math::E)
  base ** x
end
exp(10)    # ArgumentError!
exp(base: 2, x: 20) #= 1024
```

# Even in natural languages :)

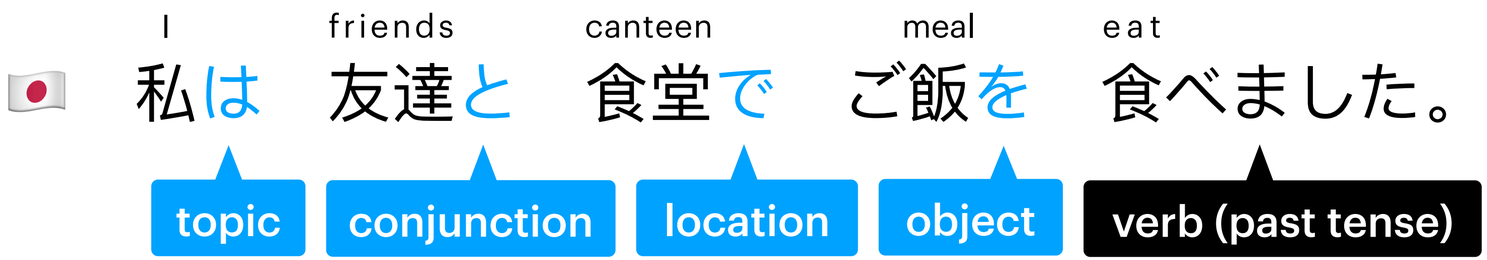
## eat(who: I, with: friends, where: canteen, what: meal)

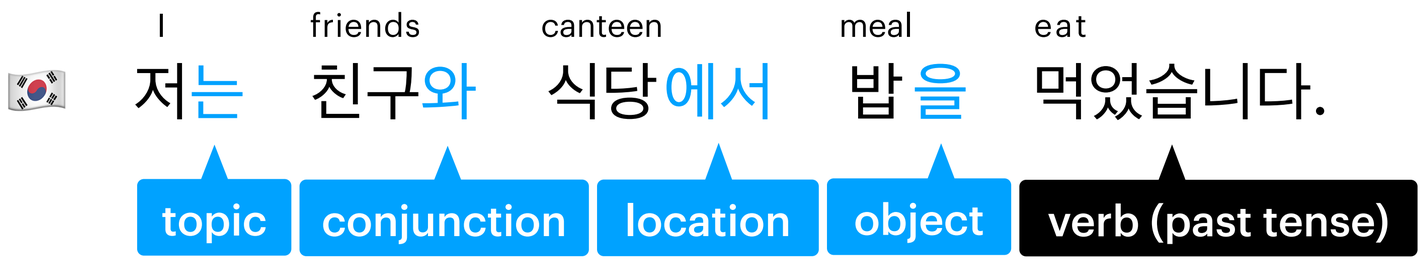
| | I | friends | canteen | meal | eat |
|---|---|---|---|---|---|
| 🇯🇵 | 私は | 友達と | 食堂で | ご飯を | 食べました。 |

| | I | friends | canteen | meal | eat |
|---|---|---|---|---|---|
| 🇰🇷 | 저는 | 친구와 | 식당에서 | 밥을 | 먹었습니다. |

# Even in natural languages :)

eat(who: I, with: friends, where: canteen, what: meal)



# Combining commutativity and currying

## The OCaml way

- OLabl extends OCaml with labeled and optional arguments, formalized in a label-selective λ-calculus[1].
- All features of OLabl are merged into OCaml 3, with subtle differences[2].
- **[Q1]** How to deal with out-of-order applications to labeled arguments?
- **[Q2]** How to distinguish between a partially applied function and a fully applied one using default values?


1. Jun P. Furuse and Jacques Garrigue. A Label-Selective Lambda-Calculus with Optional Arguments and its Compilation Method. 1995.
2. Jacques Garrigue. Labeled and Optional Arguments for Objective Caml. In JSSST SIG-PPL Workshop 2001.

# Optional parameters are options

$\tau$ option = None | Some of $\tau$

optional & named

required & positional

```
let exp ?(base = 2.71828) x = base ** x
(* val exp : ?base:float → float → float = <fun> *)

exp 10.0                (*= 22026. *)
exp 10.0 ~base:2.0      (*=  1024. *)
(exp 10.0) ~base:2.0    (* TypeError! *)

let exp ?base x =
  let base = match base with None → 2.71828 | Some b → b in
  base ** x
(* val exp : ?base:float → float → float = <fun> *)
```

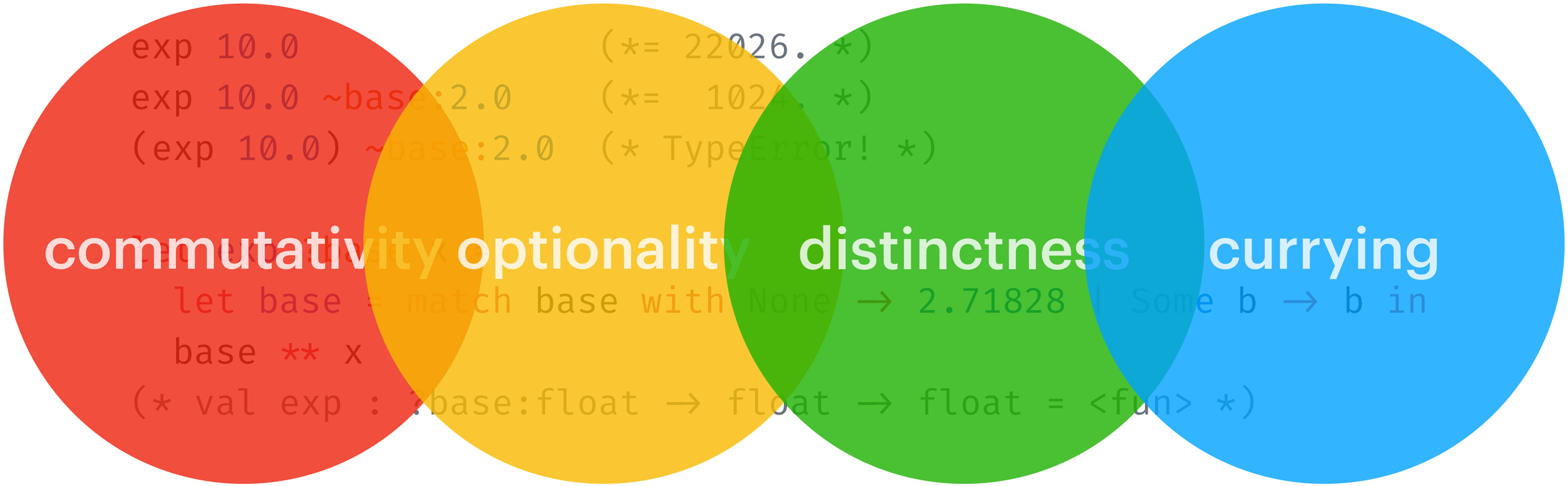
# Optional parameters are options

$\tau$ option = None | Some of $\tau$

optional & named

required & positional

```
let exp ?(base = 2.71828) x = base ** x
(* val exp : ?base:float → float → float = <fun> *)
```



# If a function is uncurried...

copy(*to:* array1, *from:* array2)

# If a function is uncurried...

copy({ *to:* array1, *from:* array2 })

# If a function is uncurried...

copy({ *from*: array2, *to*: array1 })

# How to handle optional fields in a record?

## Workarounds in Haskell

```haskell
data Args = Args { base :: Double
                 , x    :: Double }
```

```haskell
exp :: Args → Double
exp Args {..} = base ** x
```

# How to handle optional fields in a record?

## Workarounds in Haskell

```haskell
data Args = Args { base :: Double
                 , x    :: Double }


defaultArgs :: Args
defaultArgs = Args { base = 2.7, x = 1 }

exp defaultArgs { base = 2, x = 10 }
```

```haskell
exp :: Args → Double
exp Args {..} = base ** x
```

# How to handle optional fields in a record?

## Workarounds in Haskell

```haskell
data Args = Args { base :: Double
                 , x    :: Double }


defaultArgs :: Args
defaultArgs = Args { base = 2.7, x = 1 }

exp defaultArgs { base = 2, x = 10 }
```

```haskell
exp :: Args → Double
exp Args {..} = base ** x


exp' :: (Args → Args) → Double
exp' update = exp (update defaultArgs)

exp' (\args → args { base = 2, x = 10 })
```

# How to handle optional fields in a record?

## Workarounds in Haskell

```haskell
data Args = Args { base :: Double
                 , x    :: Double }


defaultArgs :: Args
defaultArgs = Args { base = 2.7, x = 1 }

exp defaultArgs { base = 2, x = 10 }


newtype ReqArgs = ReqArgs { x :: Double }

mkArgs :: ReqArgs → Args
mkArgs ReqArgs {..} = Args { base = 2.7, .. }

exp (mkArgs ReqArgs { x = 10 }) { base = 2 }
```

```haskell
exp :: Args → Double
exp Args {..} = base ** x


exp' :: (Args → Args) → Double
exp' update = exp (update defaultArgs)

exp' (\args → args { base = 2, x = 10 })
```

# How to handle optional fields in a record?

## Workarounds in Haskell

```haskell
data Args = Args { base :: Double
                 , x    :: Double }


defaultArgs :: Args
defaultArgs = Args { base = 2.7, x = 1 }

exp defaultArgs { base = 2, x = 10 }


newtype ReqArgs = ReqArgs { x :: Double }

mkArgs :: ReqArgs → Args
mkArgs ReqArgs {..} = Args { base = 2.7, .. }

exp (mkArgs ReqArgs { x = 10 }) { base = 2 }
```

```haskell
exp :: Args → Double
exp Args {..} = base ** x


exp' :: (Args → Args) → Double
exp' update = exp (update defaultArgs)

exp' (\args → args { base = 2, x = 10 })


newtype OptArgs = OptArgs { base :: Double }

exp'' :: (OptArgs → Args) → Double
exp'' update = exp (update OptArgs { base = 2.7 })

exp'' (\OptArgs {..} →
        Args { x = 10, .. } { base = 2 })
```

# How to handle optional fields in a record?

## Workarounds in Haskell

```haskell
data Args = Args { base :: Double
                 , x    :: Double }


defaultArgs :: Args
defaultArgs = Args { base = 2.7, x = 1 }

exp defaultArgs { base = 2, x = 10 }


newtype ReqArgs = ReqArgs { x :: Double }

mkArgs :: ReqArgs → Args
mkArgs ReqArgs {..} = Args { base = 2.7, .. }

exp (mkArgs ReqArgs { x = 10 }) { base = 2 }
```

```haskell
exp :: Args → Double
exp Args {..} = base ** x


exp' :: (Args → Args) → Double
exp' update = exp (update defaultArgs)

exp' (\args → args { base = 2, x = 10 })


newtype OptArgs = OptArgs { base :: Double }

exp'' :: (OptArgs → Args) → Double
exp'' update = exp (update OptArgs { base = 2.7 })

exp'' (\OptArgs {..} →
         Args { x = 10, .. } { base = 2 })
```

**Too much boilerplate code!**

# Our approach via desugaring

**source code**

```
exp { x: Double; base: Double = 2.7 } = base ** x
```

# Our approach via desugaring

**source code**

```
exp { x: Double; base: Double = 2.7 } = base ** x
```

**desugared code (option 1: failable projections)**

```
exp = λargs: { x: Double | base?: Double }.
  let base = args.base ? 2.7 in let x = args.x in base ** x
```

# Our approach via desugaring

**source code**

```
exp { x: Double; base: Double = 2.7 } = base ** x
```

**desugared code (option 1: failable projections)**

```
exp = λargs: { x: Double | base?: Double }.
  let base = args.base ? 2.7 in let x = args.x in base ** x
```

**desugared code (option 2: open-bindings)**

```
exp = λargs: { x: Double | base?: Double }.
  let base = 2.7 in open args in base ** x
```

# Our approach via desugaring

**source code**

```
exp { x: Double; base: Double = 2.7 } = base ** x
```

**desugared code (option 1: failable projections)**

```
exp = λargs: { x: Double | base?: Double }.
  let base = args.base ? 2.7 in let x = args.x in base ** x
```

**desugared code (option 2: open-bindings)**

```
exp = λargs: { x: Double | base?: Double }.
  let base = 2.7 in open args in base ** x
```

**use cases**

```
exp { x = 10.0; base = 2.0 }
exp { base = 2.0; x = 10.0 }
```

```
exp { x = 10.0 }
```

width subtyping

```
exp { x = 10.0; base = true }
```

type error! ✘

# Our approach via desugaring

**source code**

```
exp { x: Double; base: Double = 2.7 } = base ** x
```

**desugared code (option 1: failable projections)**

```
exp = λargs: { x: Double | base?: Double }.
  let base = args.base ? 2.7 in let x = args.x in base ** x
```

**desugared code (option 2: open-bindings)**

```
exp = λargs: { x: Double | base?: Double }.
  let base = 2.7 in open args in base ** x
```

**use cases**

```
let x = 10.0 in let base = 2.0 in
exp { base = 2.0; x = 10.0 }
```

```
exp { x = 10.0 }
exp { x = 10.0; base = true }
```

width subtyping

type error! ✘

# Based on $\lambda_{<:}$ with record subtyping

→ <: Top — *Based on $\lambda_{\to}$ (9-1)*

*Syntax*

| | | | |
|---|---|---|---|
| t | ::= | | *terms:* |
| | | x | *variable* |
| | | λx:T.t | *abstraction* |
| | | t t | *application* |
| v | ::= | | *values:* |
| | | λx:T.t | *abstraction value* |
| T | ::= | | *types:* |
| | | Top | *maximum type* |
| | | T→T | *type of functions* |
| Γ | ::= | | *contexts:* |
| | | ∅ | *empty context* |
| | | Γ, x:T | *term variable binding* |

*Evaluation* — $t \longrightarrow t'$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-AppAbs)}$$

*Subtyping* — $S <: T$

$$S <: S \quad \text{(S-Refl)}$$

$$\frac{S <: U \qquad U <: T}{S <: T} \quad \text{(S-Trans)}$$

$$S <: \text{Top} \quad \text{(S-Top)}$$

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \quad \text{(S-Arrow)}$$

*Typing* — $\Gamma \vdash t : T$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-Var)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-App)}$$

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \quad \text{(T-Sub)}$$

**Figure 15-1: Simply typed lambda-calculus with subtyping ($\lambda_{<:}$)**

→ {} — *Extends $\lambda_{\to}$ (9-1)*

*New syntactic forms*

| | | | |
|---|---|---|---|
| t | ::= | ... | *terms:* |
| | | $\{l_i{=}t_i\ ^{i\in 1..n}\}$ | *record* |
| | | t.l | *projection* |
| v | ::= | ... | *values:* |
| | | $\{l_i{=}v_i\ ^{i\in 1..n}\}$ | *record value* |
| T | ::= | ... | *types:* |
| | | $\{l_i{:}T_i\ ^{i\in 1..n}\}$ | *type of records* |

*New evaluation rules* — $t \longrightarrow t'$

$$\{l_i{=}v_i\ ^{i\in 1..n}\}.l_j \longrightarrow v_j \quad \text{(E-ProjRcd)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1.l \longrightarrow t_1'.l} \quad \text{(E-Proj)}$$

$$\frac{t_j \longrightarrow t_j'}{\{l_i{=}v_i\ ^{i\in 1..j-1}, l_j{=}t_j, l_k{=}t_k\ ^{k\in j+1..n}\} \longrightarrow \{l_i{=}v_i\ ^{i\in 1..j-1}, l_j{=}t_j', l_k{=}t_k\ ^{k\in j+1..n}\}} \quad \text{(E-Rcd)}$$

*New typing rules* — $\Gamma \vdash t : T$

$$\frac{\text{for each } i \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i{=}t_i\ ^{i\in 1..n}\} : \{l_i{:}T_i\ ^{i\in 1..n}\}} \quad \text{(T-Rcd)}$$

$$\frac{\Gamma \vdash t_1 : \{l_i{:}T_i\ ^{i\in 1..n}\}}{\Gamma \vdash t_1.l_j : T_j} \quad \text{(T-Proj)}$$

**Figure 15-2: Records (same as Figure 11-7)**

→ {} <: — *Extends $\lambda_{<:}$ (15-1) and simple record rules (15-2)*

*New subtyping rules* — $S <: T$

$$\{l_i{:}T_i\ ^{i\in 1..n+k}\} <: \{l_i{:}T_i\ ^{i\in 1..n}\} \quad \text{(S-RcdWidth)}$$

$$\frac{\text{for each } i \quad S_i <: T_i}{\{l_i{:}S_i\ ^{i\in 1..n}\} <: \{l_i{:}T_i\ ^{i\in 1..n}\}} \quad \text{(S-RcdDepth)}$$

$$\frac{\{k_j{:}S_j\ ^{j\in 1..n}\} \text{ is a permutation of } \{l_i{:}T_i\ ^{i\in 1..n}\}}{\{k_j{:}S_j\ ^{j\in 1..n}\} <: \{l_i{:}T_i\ ^{i\in 1..n}\}} \quad \text{(S-RcdPerm)}$$

**Figure 15-3: Records and subtyping**

# Minimal extension: optional fields

1. One new type: $\{\overline{\ell_i : T_i} \mid \overline{\ell_j ? : T_j}\}$
2. One new typing rule:
$$\frac{\Gamma \vdash e \Rightarrow T \qquad \{\overline{\ell_i : T_i}; \overline{\ell_j : T_j}\} <: T <: \{\overline{\ell_i : T_i}\}}{\Gamma \vdash e \Leftarrow \{\overline{\ell_i : T_i} \mid \overline{\ell_j ? : T_j}\}}$$
3. *No* subtyping rule concerning optional fields

# Option 1: failable projections

1. One new term: $e_1.\ell\,?\,e_2$
2. One new typing rule:
$$\frac{\Gamma \vdash e_1 \Rightarrow \{\overline{\ell_i : T_i} \mid \overline{\ell_j? : T_j}; \ell? : T; \overline{\ell_{j'}? : T_{j'}}\} \qquad \Gamma \vdash e_2 \Rightarrow T}{\Gamma \vdash e_1.\ell\,?\,e_2 \Rightarrow T}$$
3. Two new computation rules:
$$\{\overline{\ell_i = v_i}; \ell = v; \overline{\ell_j = v_j}\}.\ell\,?\,e_2 \longrightarrow v$$
$$\frac{\ell \notin \{\overline{\ell_i}\}}{\{\overline{\ell_i = v_i}\}.\ell\,?\,e_2 \longrightarrow e_2}$$

# Option 2: open-bindings

1. One new term: $\text{open } e_1 \text{ in } e_2$
2. One new typing rule: $$\frac{\Gamma, \overline{\ell_j : T_j} \vdash e_1 \Rightarrow \{\overline{\ell_i : T_i} \mid \overline{\ell_j ? : T_j}\} \qquad \Gamma, \overline{\ell_j : T_j}, \overline{\ell_i : T_i} \vdash e_2 : T'}{\Gamma, \overline{\ell_j : T_j} \vdash \text{open } e_1 \text{ in } e_2 \Rightarrow T'}$$
3. One new computation rule: $\Delta \vdash \text{open } \{\overline{\ell_i = v_i}\} \text{ in } e_2 \longrightarrow \text{let } \overline{\ell_i = v_i} \text{ in } e_2$

valuation environment

# Conclusion

- Named and optional arguments are widely supported in ***object-oriented programming languages*** but are hardly formalized.
- Garrigue et al. formalized a label-selective λ-calculus for ***OCaml*** which combines commutativity and currying, but it is non-trivial to be integrated with other sophisticated λ-calculi.
- It is well known in ***Haskell*** that named arguments can be encoded as records, but it requires a lot of boilerplate code to support both required and optional arguments.
- We presented a minimal extension to $\boldsymbol{\lambda_{<:}}$ that supports optional arguments via desugaring. Metatheories are currently under construction.

# Thank you!