香 港 大 學
THE UNIVERSITY OF HONG KONG

ESOP'25

Named Arguments as Intersections, Optional Arguments as Unions

Yaozhu Sun and Bruno C. d. S. Oliveira

6 May 2025

Why Argument Names Matter

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 Argument Names Matter

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 the Early Days

Smalltalk and its followers

Smalltalk

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

Objective-C

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

Swift

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

Named Arguments Nowadays

Python vs Ruby

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

exp(base=2, x=10)      #= 1024
exp(x=10)              #= e^10
exp(10, 2)             #= 1024

args = { "base": 2, "x": 10 }
exp(**args)            #= 1024
```

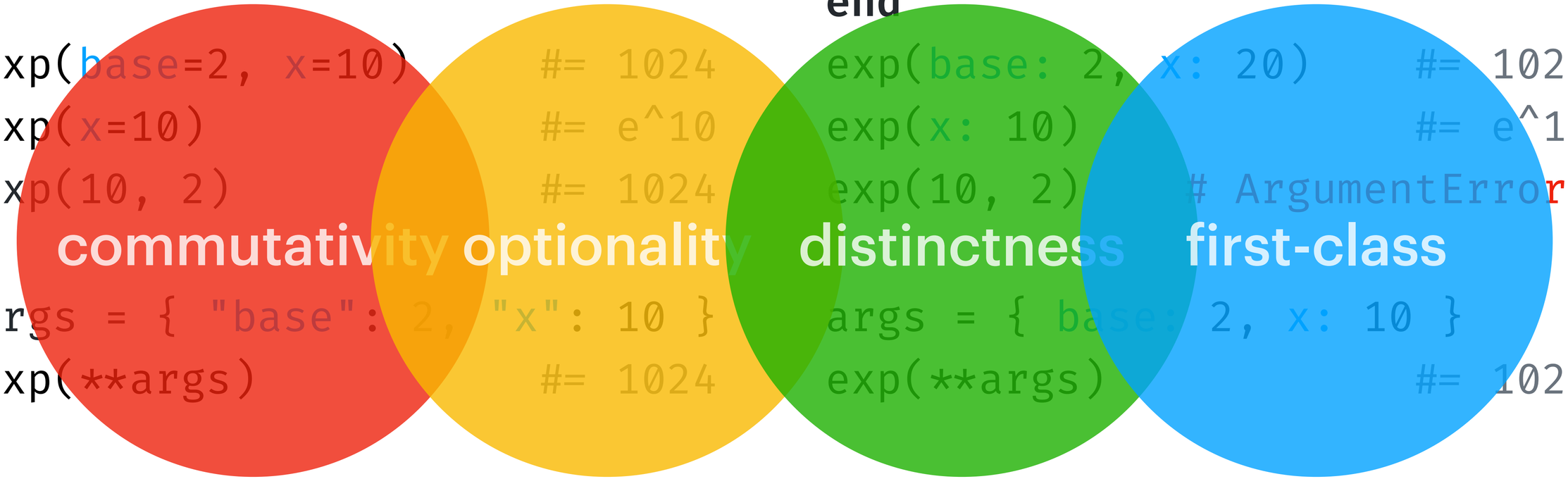
```ruby
def exp(x:, base: Math::E)
  base ** x
end
exp(base: 2, x: 20)    #= 1024
exp(x: 10)             #= e^10
exp(10, 2)    # ArgumentError!

args = { base: 2, x: 10 }
exp(**args)            #= 1024
```

Named Arguments Nowadays

Python vs Ruby

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

exp(base=2, x=10)      #= 1024
exp(x=10)              #= e^10
exp(10, 2)             #= 1024

args = { "base": 2, "x": 10 }
exp(**args)            #= 1024
```

```ruby
def exp(x:, base: Math::E)
    base ** x
end

exp(base: 2, x: 20)    #= 1024
exp(x: 10)             #= e^10
exp(10, 2)             # ArgumentError!

args = { base: 2, x: 10 }
exp(**args)            #= 1024
```

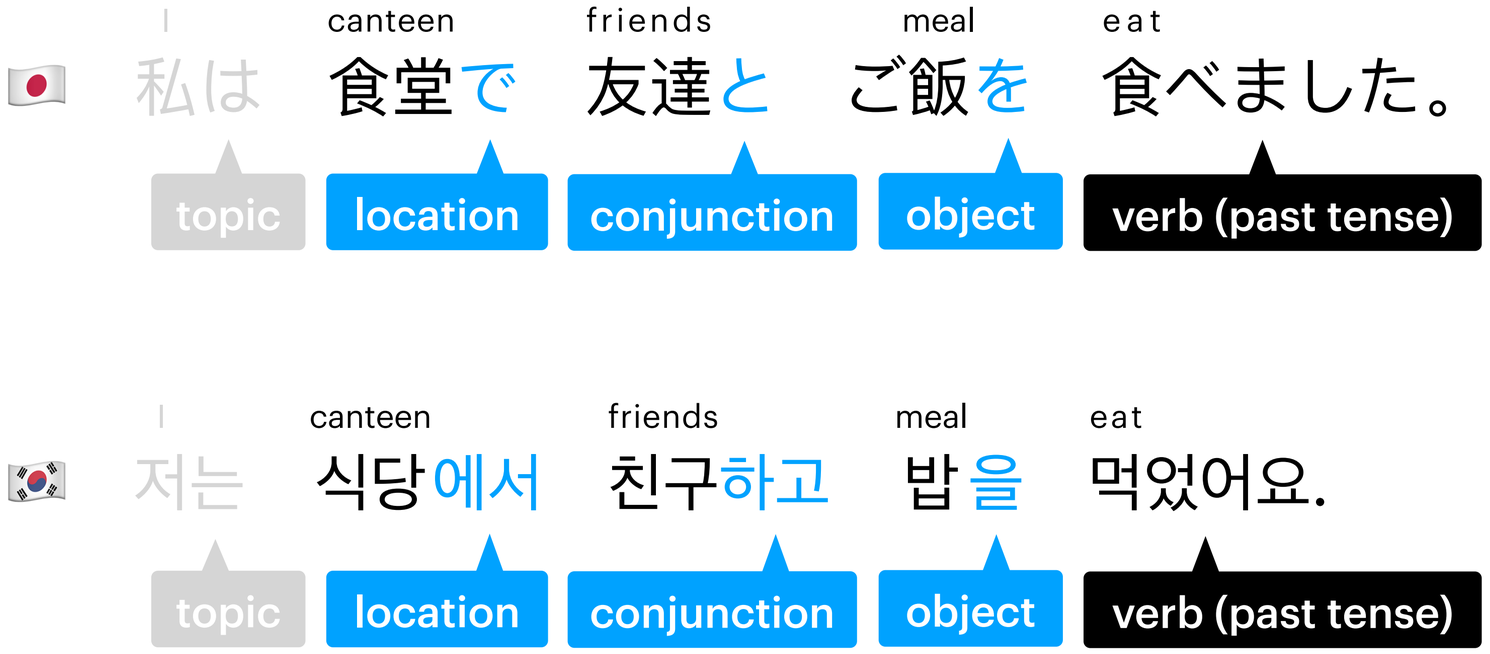


Even in Natural Languages :)

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

Even in Natural Languages :)

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



Named and Optional Arguments

in Python

Default value for an optional argument

```
class App: # from a web server library
  def run(self, host, port, debug = False):
    assert isinstance(debug, bool) # actual code omitted

args = { "host": "0.0.0.0", "port": 80, "debug": True }
app.run(**args)
```

Named arguments from a variable

Named and Optional Arguments

in Python/mypy

```
class App: # from a web server library
  def run(self, host: str, port: int, debug: bool = False):
    assert isinstance(debug, bool) # actual code omitted


type Args = { "host": str, "port": int, "debug": NotRequired[bool] }
args: Args = { "host": "0.0.0.0", "port": 80, "debug": True }
app.run(**args)
```

The type of args is a more precise `TypedDict`, instead of the inferred `dict[str,object]`.

Type Unsafety
in Python/mypy

```
class App: # from a web server library
  def run(self, host: str, port: int, debug: bool = False):
    assert isinstance(debug, bool) # actual code omitted

def f(args: { "host": str, "port": int, "debug": str }) \
          → { "host": str, "port": int }:
  return args
```

Type Unsafety
in Python/mypy

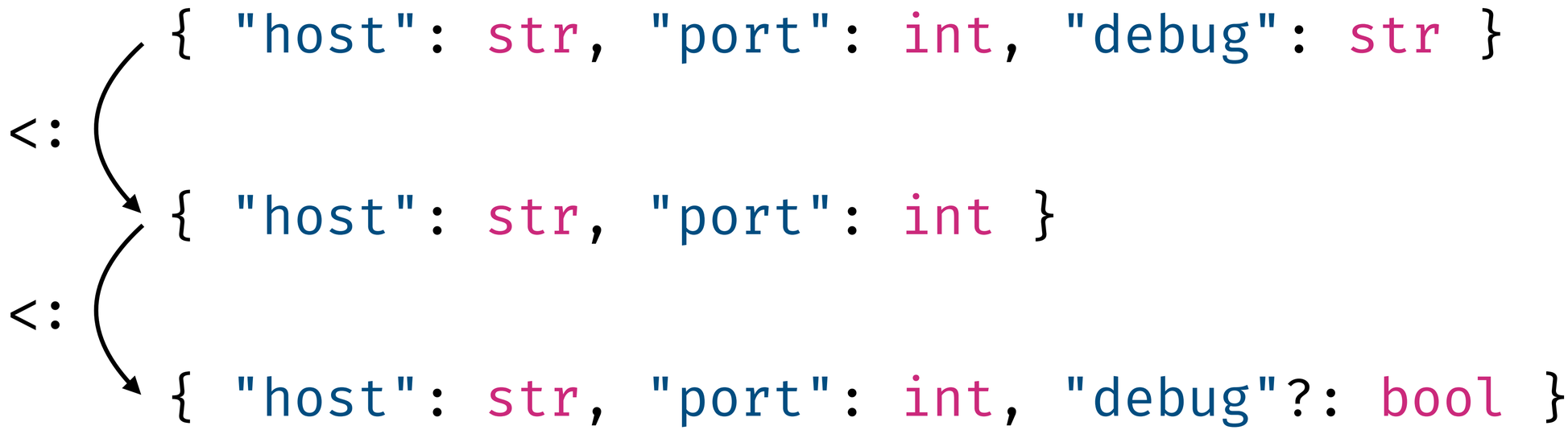
```
class App: # from a web server library
  def run(self, host: str, port: int, debug: bool = False):
    assert isinstance(debug, bool)

def f(args: { "host": str, "port": int, "debug": str }) \
          → { "host": str, "port": int }:
  return args

type Args = { "host": str, "port": int }
args: Args = f({ "host": "0.0.0.0", "port": 80, "debug": "Oops!" })
app.run(**args)
```

Runtime error because “debug” is not boolean!

The key “debug” with a wrong type is forgotten in the static type “Args”.

The call type-checks because “debug” is optional.

Questionable Subsumption Chain



Remodeling Named and Optional Arguments

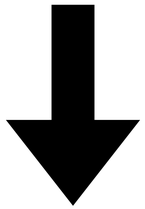
based on intersection and union types

```
def run(host: str, port: int, debug?: bool)
```

Remodeling Named and Optional Arguments

based on intersection and union types

```
def run(host: str, port: int, debug?: bool)
```



```
def run(args: { host: str } & { port: int } & { debug: bool|None })
```

Optional Arguments as Unions

Named Arguments as Intersections

Intersection and Union Types

via Venn diagram

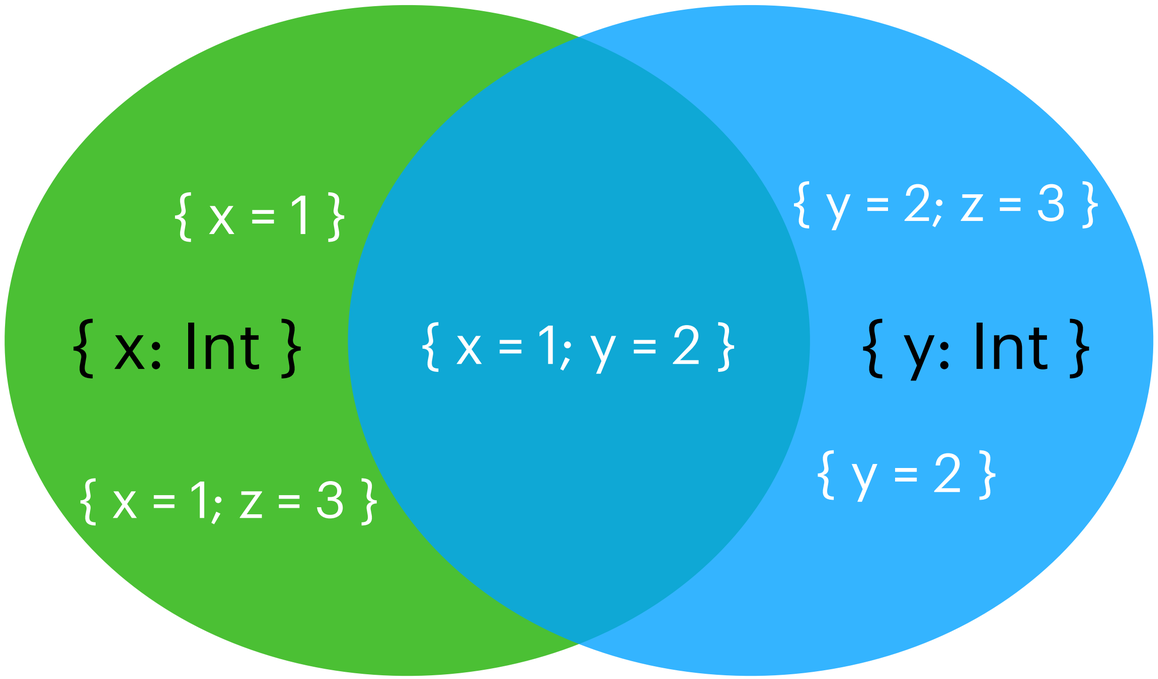
Intersection types: A & B

- { x = 1; y = 2 } : { x: Int } & { y: Int }

  i.e. { x: Int; y: Int }

Union types: A | B

- { x = 1 } : { x: Int } | { y: Int }
- { y = 2 } : { x: Int } | { y: Int }
- ……



Merge, Disjointness, and Switch-Case

in CP

Merge operator: e_1 , e_2

- record concatenation: { x = 1 } , { y = 2 } = { x = 1; y = 2 }
- function overloading, trait composition, ...

Type disjointness

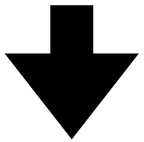
- to avoid semantic ambiguity: ({ x = 1 } , { x = 2 }).x = ?? (ill-typed!)
- to make merging commutative: e_1 , e_2 = e_2 , e_1

Type-based switch-case: switch e_0 as x case A ⇒ e_1 case B ⇒ e_2

Translating the Function

in CP

```
run { host: String; port: Int; debug: Bool = false } =
  -- function body
```

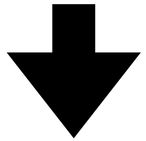


```
run (args: { host: String } & { port: Int } & { debug: Bool|Null }) =
  let host = args.host in
  let port = args.port in
  let debug = switch args.debug as d case Bool ⇒ d
                                   case Null ⇒ false in
  -- function body
```

Rewriting the Call Site

in CP

```
f(args: { host: String; port: Int; debug: String })
      : { host: String; port: Int } = args;
args = f { host = "0.0.0.0"; port = 80; debug = "Oops!" };

run args -- args : { host: String; port: Int }
```



```
run { host = args.host; port = args.port; debug = null }
```

Rewrite the potentially forgotten “debug”.

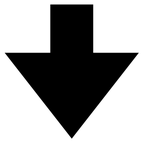
writing the Call Site
in CP

f removes “debug” because of coercive subtyping in CP.

```
f(args: { host: String; port: Int; debug: String })
      : { host: String; port: Int } = args;
args = f { host = "0.0.0.0"; port = 80; debug = "Oops!" };

run args -- args : { host: String; port: Int }
```

Rewrite the potentially forgotten “debug“.

```
run { host = args.host; port = args.port; debug = null }
```



The merge operator (record concatenation)

```
run (args , { debug = null })
```

Actual (and more efficient) core code.

Takeaways

for optional arguments

- $\{\text{required} : A; \text{optional}? : B\} \neq \{\text{required} : A\}$
  - the former is a **subtype** because it contains more information that:
    - "optional" can be absent, but if it's present, it must have type B.
- Correspondingly, $\{\text{optional} = \text{null}\}$ is explicitly added in a core term if "optional" is missing **statically**.
  - In essence, we implement Python's ** operator as per the static type.

Named Arguments in Different Languages

	Smalltalk	Python	Ruby	Racket	OCaml	C#	Scala	Dart	Swift	CP
Commutativity	○	●	●	●	●	●	●	●	○	●
Optionality	○	●	●	●	●	●	●	●	●	●
Currying	○	○	○	○	●	○	○	○	○	○
Distinctness	*n/a*	○	●	●	●	○	○	●	●	●
First-class value	○	●	●	◐	○	○	○	○	○	●
Static typing	○	○	○	○	●	●	●	●	●	●
Soundness proof	○	○	○	○	●	○	○	○	○	●

n/a: Smalltalk does not support positional arguments at all.

◐: Racket's support for first-class named arguments is limited and forbids commutativity.

Formalization

Types $\mathscr{A}, \mathscr{B} ::= \mathbb{Z}$

Normal function | $(\mathscr{A}) \to \mathscr{B}$

Function w/ named parameters | $\{\mathscr{P}\} \to \mathscr{B}$

(First-class) named arguments | $\{\mathscr{K}\}$

Expressions $\epsilon ::= n \mid x \mid \epsilon_1\ \epsilon_2$

Normal function | $\lambda(x : \mathscr{A})\ .\ \epsilon$

Function w/ named parameters | $\lambda\{\rho\}\ .\ \epsilon$

(First-class) named arguments | $\{\kappa\}$

UAENA (source)

Types $A, B ::= \top \mid \bot \mid \mathbf{Null} \mid \mathbb{Z}$

Normal function | $A \to B$

Single-field record | $\{\ell : A\}$

Intersection and union | $A \wedge B \mid A \vee B$

Expressions $e ::= \{\} \mid \mathbf{null} \mid n \mid x \mid e_1\ e_2$

Normal function | $\lambda x : A\ .\ e : B$

Single-field record | $\{\ell : A = e\} \mid e\ .\ \ell$

Merge | $e_1, e_2 \mid \mathbf{switch}\ e_0\ \mathbf{as}\ x$

Type-based switch-case

$\mathbf{case}\ A \Rightarrow e_1$

$\mathbf{case}\ B \Rightarrow e_2$

λ_{iu} (core)

Formalization

Types $\mathscr{A}, \mathscr{B} ::= \mathbb{Z}$

- Normal function | $(\mathscr{A}) \to$
- Function w/ named parameters | $\{\mathscr{P}\} \to$
- (First-class) named arguments | $\{\mathscr{K}\}$

Types $A, B ::= \top \mid \bot \mid \mathbf{Null} \mid \mathbb{Z}$

- mal function | $A \to B$
- e-field record | $\{\ell : A\}$
- on and union | $A \wedge B \mid A \vee B$

Type translation

$|\mathscr{A}| = A$

Expressions $\epsilon ::= n \mid x \mid \epsilon_1\ \epsilon_2$

- Normal function | $\lambda(x : \mathscr{A})$
- Function w/ named parameters | $\lambda\{\rho\}\ .\ \epsilon$
- (First-class) named arguments | $\{\kappa\}$

Expressions $e ::= \{\} \mid \mathbf{null} \mid n \mid x \mid e_1\ e_2$

- rmal function | $\lambda x : A\ .\ e : B$
- e-field record | $\{\ell : A = e\} \mid e\ .\ \ell$
- Merge | $e_1, e_2 \mid \mathbf{switch}\ e_0\ \mathbf{as}\ x$
  $\mathbf{case}\ A \Rightarrow e_1$
  $\mathbf{case}\ B \Rightarrow e_2$

Type-based switch-case

Elaboration

$\Delta \vdash \epsilon : \mathscr{A} \rightsquigarrow e$

UAENA (source)

λ_{iu} (core)

Proving Type Safety

via elaboration semantics

1. Core type soundness:

   For core typing $\Gamma \vdash e : A$ and reduction $e \longrightarrow e'$, we have

   [Progress] *If* $\cdot \vdash e : A$*, then either* e *is a value or* $\exists e', e \longrightarrow e'$.

   [Preservation] *If* $\Gamma \vdash e : A$ *and* $e \longrightarrow e'$*, then* $\Gamma \vdash e' : A$.

2. Elaboration type soundness: *If* $\Delta \vdash \epsilon : \mathscr{A} \rightsquigarrow e$*, then* $|\Delta| \vdash e : |\mathscr{A}|$.

More in the Paper...

- Complete code to reproduce the type-safety issues in **Python** (mypy) and **Ruby** (Steep & Sorbet);
- A working example of fractals with named and optional arguments in **CP**.
- **Rocq** (formerly Coq) formalization of the elaboration from UAENA to λ_{iu} and the type-safety proof thereof.
- Detailed comparisons with **OCaml**, **Scala**, **Racket**, **Haskell**, etc.

Both proof and impl. are open source:

https://github.com/yzyzsun/lambda-iu  https://github.com/yzyzsun/CP-next


Q&A