香 港 大 學
THE UNIVERSITY OF HONG KONG

华东师大可信软件
学术沙龙

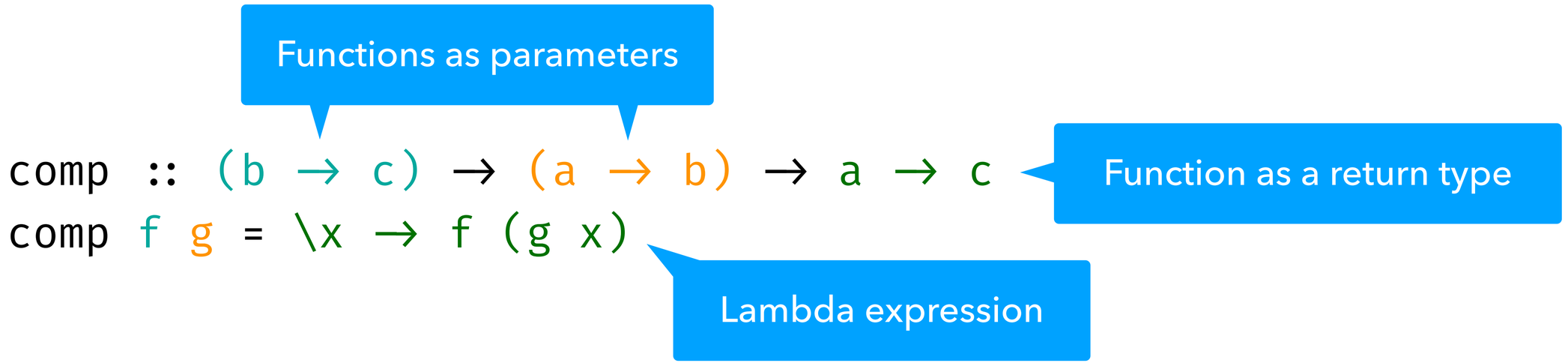
Pursuing Type Safety for First-Class Constructs

孙耀珠 | 2025-03-31

First-Class Constructs

in FP & OOP

- **Functions** are first-class in functional programming.



- Similarly, **objects** are first-class in object-oriented programming…

🤔 What about classes?

Type Safety

Well-typed programs cannot “go wrong”

$e_1 + e_2$

🤦 $e_1 = 1$ $e_2 = 2$

🤦 $e_1 = 1.1$ $e_2 = 2.2$

🤷 $e_1 = 1$ $e_2 = \text{“str”}$

🤔 $e_1 = 1$ $e_2 = 2.2$

$$\frac{e_1 : \text{Int} \quad e_2 : \text{Int}}{e_1 + e_2 : \text{Int}}$$

$$\frac{e_1 : \text{Double} \quad e_2 : \text{Double}}{e_1 + e_2 : \text{Double}}$$

$e_1 + e_2$ is not well-typed in other cases

Agenda

1. *Type-Safe Compilation of Dynamic Inheritance via Merging* ¿TOPLAS?
   - **Classes/Traits** are first-class
   - For type safety: merging rather than overriding
2. *Named Arguments as Intersections, Optional Arguments as Unions* [ESOP'25]
   - **Named arguments** are first-class
   - For type safety: rewriting call sites

First-Class Classes

in JavaScript

Mixin Pattern

Class as a parameter

Dynamic inheritance

```
function Mixin(Base) {
  return class extends Base {
    m() { return 48; }
  };
}
```

First-Class Classes

in TypeScript

This type represents an empty class.

```
type Constructor = new (...args: any[]) ⇒ {};

function Mixin<TBase extends Constructor>(Base: TBase) {
  return class extends Base {
    m(): number { return 48; }
  };
}
```

If m() exists in Base, that one will be overridden by the definition here.

Unsafe Overriding with Mixin

in TypeScript

```
function Mixin<TBase extends Constructor>(Base: TBase) {
  return class extends Base {
    m(): number { return 48; }
  };
}
class A {
  m(): string { return "foobar"; }
  n(): string { return this.m().toUpperCase(); }
}
var B = Mixin(A);
(new B).n() // Runtime Error!
```

This m() overrides that in class A.

We use A as Base.

Unsafe Overriding with Mixin

in TypeScript

```
function Mixin<TBase extends Constructor>(Base: TBase) {
  return class extends Base {
    m(): number { return 48; }
  };
}
class A {
  m(): string { return "foobar"; }
  n(): string { return this.m().toUpperCase(); }
}
var B = Mixin(A);
(new B).n() // Runtime Error!
```

This m() overrides that in class A.

Here m() is expected to return a string, but the overridden one returns a number.

We use A as Base.

Type unsafe!

🤔 Add some constraints to Base?

Unsafe Overriding with Constrained Mixin

This type represents a class with n() returning a string.

```
type GConstructor<T = {}> = new (..args: any[]) ⇒ T;

type HasStringN = GConstructor<{ n: () ⇒ string }>;

function Mixin<TBase extends HasStringN>(Base: TBase) {
  return class extends Base {
    m(): number { return parseInt(this.n()); }
  };
}

var B = Mixin(A);
(new B).n() // Runtime Error!
```

This can be helpful to express dependencies.

But this doesn't help to prevent unsafe overriding of m()!

🤔 Wanna express absence of a method…

Another Look at First-Class Classes

in TypeScript

- **Nested classes** for free:

```
class A {
  Nested = class { …… };
  NestedWithParams(x, y) { return class { …… }; }
  newNested() { return new this.NestedWithParams(0, 0); }
}
```

This class is dynamically bound, and it refers to the one in class B now.

- Like virtual methods, nested classes can be **virtual**:

```
class B extends A {
  NestedWithParams(x, y) { return class { /* different impl */ }; }
}
```

Solving the Expression Problem

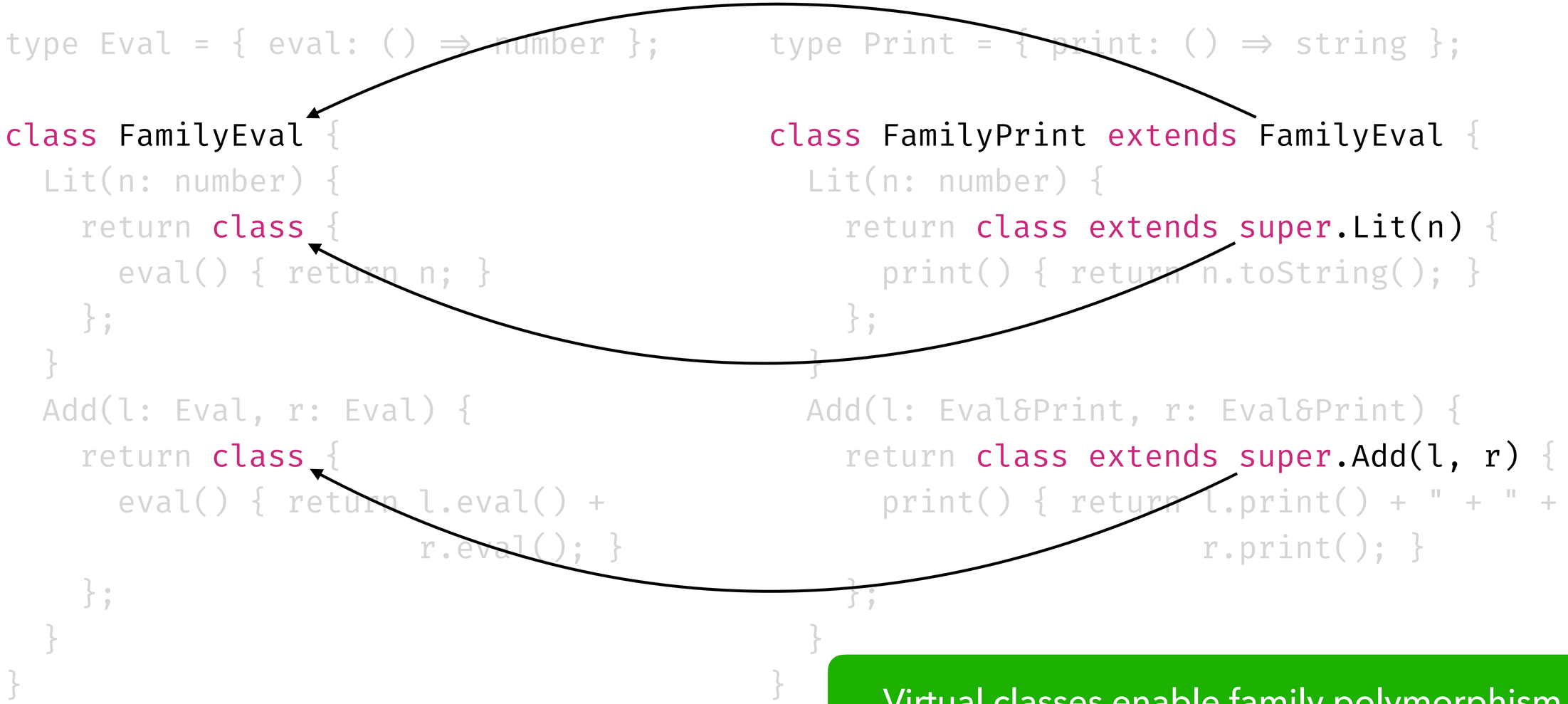
in TypeScript

```
type Eval = { eval: () ⇒ number };

class FamilyEval {
  Lit(n: number) {
    return class {
      eval() { return n; }
    };
  }
  Add(l: Eval, r: Eval) {
    return class {
      eval() { return l.eval() +
                      r.eval(); }
    };
  }
}
```

```
type Print = { print: () ⇒ string };

class FamilyPrint extends FamilyEval {
  Lit(n: number) {
    return class extends super.Lit(n) {
      print() { return n.toString(); }
    };
  }
  Add(l: Eval&Print, r: Eval&Print) {
    return class extends super.Add(l, r) {
      print() { return l.print() + " + " +
                      r.print(); }
    };
  }
}
```

Solving the Expression Problem

in TypeScript



```
type Eval = { eval: () ⇒ number };

class FamilyEval {
  Lit(n: number) {
    return class {
      eval() { return n; }
    };
  }
  Add(l: Eval, r: Eval) {
    return class {
      eval() { return l.eval() +
                      r.eval(); }
    };
  }
}
```

```
type Print = { print: () ⇒ string };

class FamilyPrint extends FamilyEval {
  Lit(n: number) {
    return class extends super.Lit(n) {
      print() { return n.toString(); }
    };
  }
  Add(l: Eval&Print, r: Eval&Print) {
    return class extends super.Add(l, r) {
      print() { return l.print() + " + " +
                       r.print(); }
    };
  }
}
```

Virtual classes enable family polymorphism.

Solving the Expression Problem

with Mixin, in TypeScript

```
function MixinNeg<TBase extends Constructor>(Base: TBase) {
  return class extends Base {
    Neg(e: Eval&Print) {
      return class {
        eval()  { return -e.eval(); }
        print() { return "-(" + e.print() + ")"; }
      };
    }
  };
}

var FamilyNeg = MixinNeg(FamilyWithDifferentTypeOfEvalPrint);
```

The extension of Neg is decoupled from FamilyEval or FamilyPrint.

Type unsafe!

Discussions

First Principles

- **Implicit overriding is dangerous**, both for type safety and semantics (e.g. *fragile base class problem*).
  - **Trait** model requires resolving conflicts **explicitly**.
- With family polymorphism, virtual classes are not overridden but **merged**.
  - Generalizing to virtual methods (or even fields):
    trait members are merged if they have the same name but **disjoint types**;
    non-disjoint types imply conflicts and thus require explicit resolution.
- In short, our language (CP) employs a **trait-like model with merging**.

Type-Safe Merging with Mixin

in CP

```
mixin (TBase * { m: Int }) (base: Trait<TBase>) =
  trait [this: TBase] inherits base ⇒ { m = 48 };
```

Dynamic inheritance (via merging)

```
mkA = trait [this: { m: String; n: String }] ⇒ {
  m = "foobar";
  n = toUpperCase this.m;
};

o = new mixin @{ m: String; n: String } mkA;
-- { m = "foobar"; n = "FOOBAR"; m = 48 }
```

Two “m” fields coexist because they have disjoint types (String * Int).

Unambiguous Merging with Mixin

in CP

```
mixin (TBase * { m: String }) (base: Trait<TBase>) =
  trait [this, TBase] inherits base ⇒ { m = "φουμπαρ" };
```

We express absence by disjointness!

```
mkA = trait [this: { m: String; n: String }] ⇒ {
  m = "foobar";
  n = toUpperCase this.m;
};

o = new mixin @{ m: String; n: String } mkA;
-- Type Error!
```

Two “m” fields conflict because
they have non-disjointness type (~~String * String~~).

Recall the fragile base class problem.

Solving the Expression Problem

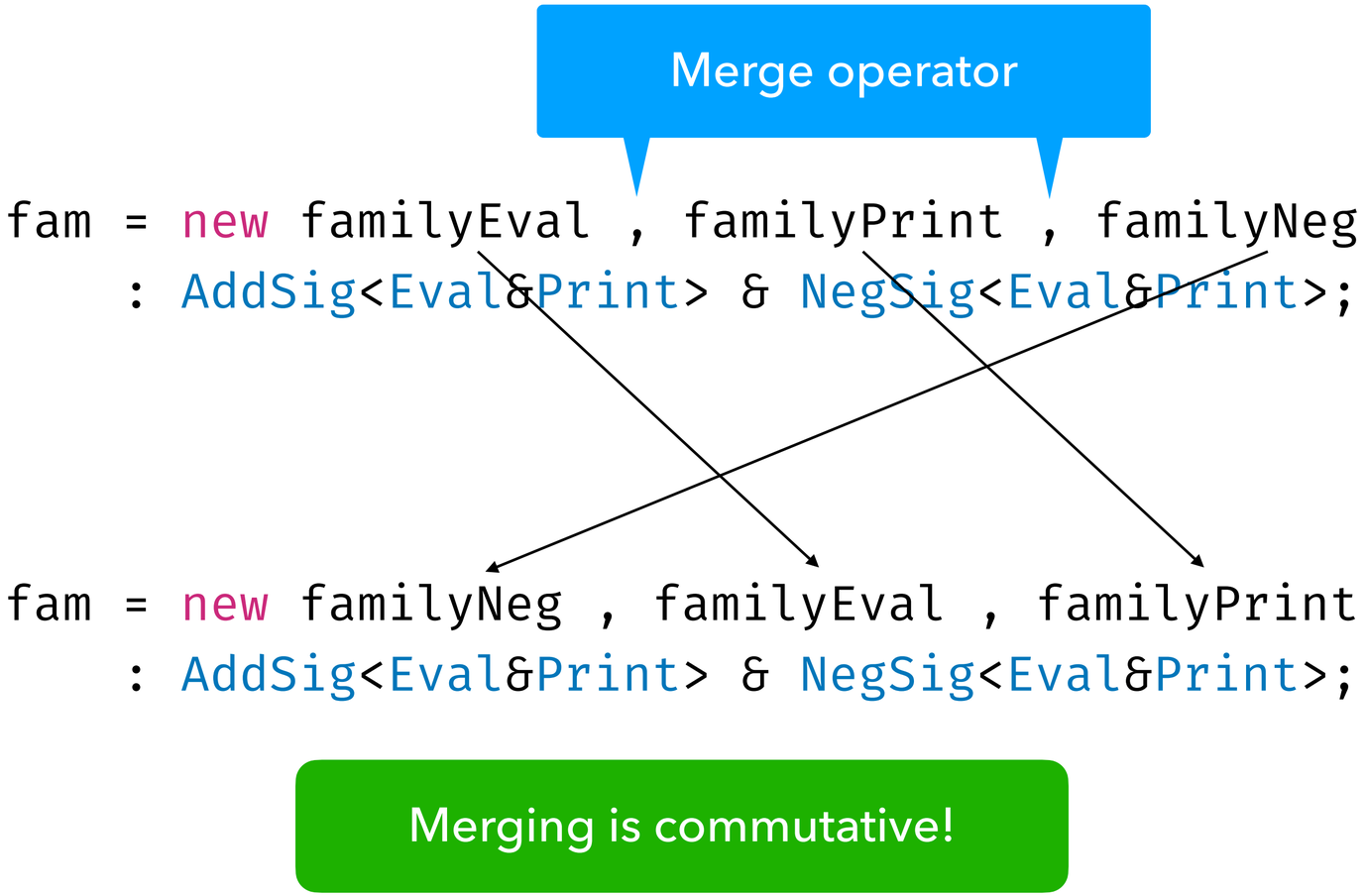
in CP

```
type AddSig<Exp> = {
  Lit: Int → Exp;
  Add: Exp → Exp → Exp;
};

type Eval = { eval: Int };

familyEval =
  trait implements AddSig<Eval> ⇒ {
    Initial family
  };
```

```
type Print = { print: String };

familyPrint =
  trait implements AddSig<Print> ⇒ {
    Adding a new operation
  };

type NegSig<Exp> = { Neg: Exp → Exp };

familyNeg =
  trait implements NegSig<Eval&Print> ⇒ {
    Adding a new expression
  };
```

Merging in the Expression Problem

in CP



Agenda

1. *Type-Safe Compilation of Dynamic Inheritance via Merging* ¿TOPLAS?
   - **Classes/Traits** are first-class
   - For type safety: merging rather than overriding

2. *Named Arguments as Intersections, Optional Arguments as Unions* [ESOP'25]
   - **Named arguments** are first-class
   - For type safety: rewriting call sites

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 and Optional Arguments

in Python 3

Default value for an optional argument

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

Type-Checking Named Arguments

in mypy for Python 3

```
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 with Named Arguments

in mypy for Python 3

```
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

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

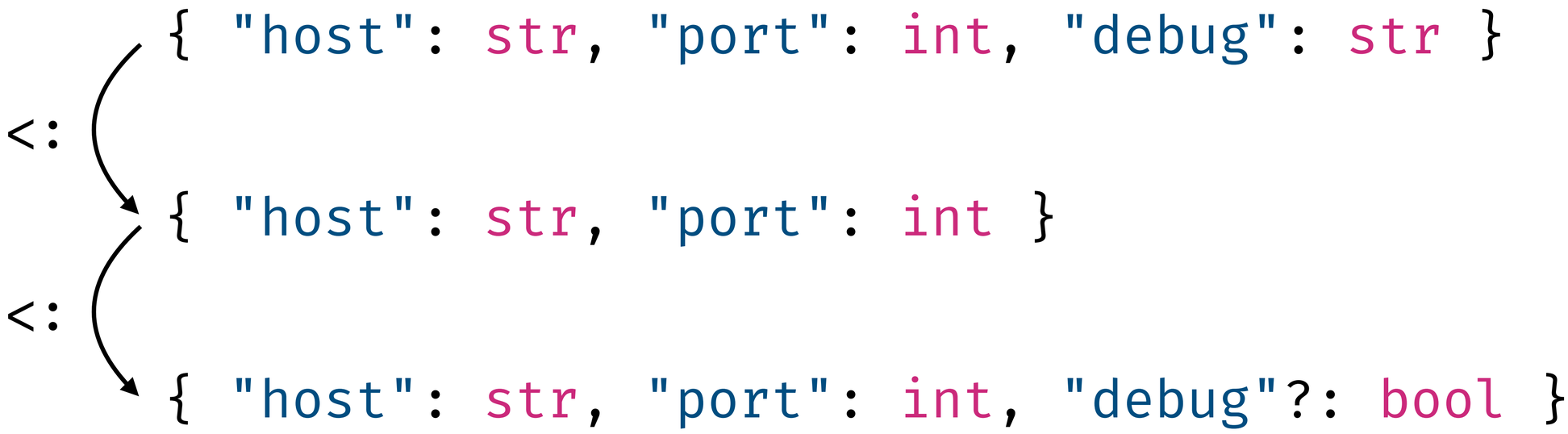
Runtime error because “debug” is not boolean!

The key “debug” is forgotten in the static type.

Type unsafe!

Questionable Subsumption Chain

in mypy for Python 3



Remodeling Named and Optional Arguments

from source to core in CP

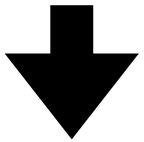
- **Named arguments as intersections**
  - *Type:* `{ x: Int; y: Int }` ⇝ `{ x: Int } & { y: Int }`
  - *Term:* `{ x = 1; y = 2 }` ⇝ `{ x = 1 } , { y = 2 }`
- **Optional arguments as unions**:
  - *Type:* `{ z?: Int }` ⇝ `{ z: Int|Null }`
  - *Term:*

```
switch z case Int  ⇒ e1
         case Null ⇒ e2
```

Translating the Function

in CP

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



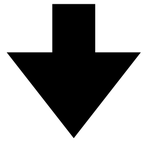
```
run (args: { host: String } & { port: Int } & { debug: Bool|Null } =
  let host = args.host in
  let port = args.port in
  let debug = switch args.debug as d case Bool ⇒ d
                                   case Null ⇒ false in
  -- function body
```

Rewriting the Call Site

in CP

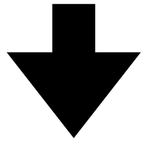
f removes “debug” because of coercive subtyping in CP.

```
args = f { host = "0.0.0.0"; port = 80; debug = "Oops!" };
run args
```



Rewrite the potentially forgotten “debug”.

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



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

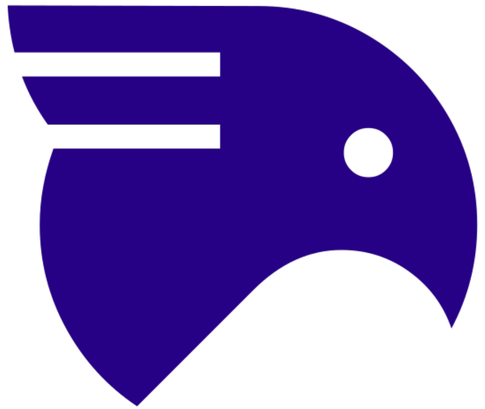
Actual (and more efficient) core code.

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, { optional = 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.

Both works are …



implemented in CP and formalized in Rocq.

Epilogue: Proving Type Safety

via Elaboration Semantics

[Core] typing $\Gamma \vdash e : A$; reduction $e \longrightarrow e'$.

[Source] elaboration $\Delta \vdash \epsilon : \mathscr{A} \rightsquigarrow e$; translation $|\Delta| = \Gamma$ and $|\mathscr{A}| = A$.

1. Core type soundness:
   - 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* $|\Delta| \vdash e : |\mathscr{A}|$.


Q&A