香 港 大 學
THE UNIVERSITY OF HONG KONG

Compositional Programming

= Modularity × Extensibility³

손 요 주
孫耀珠 | 2024-11-01

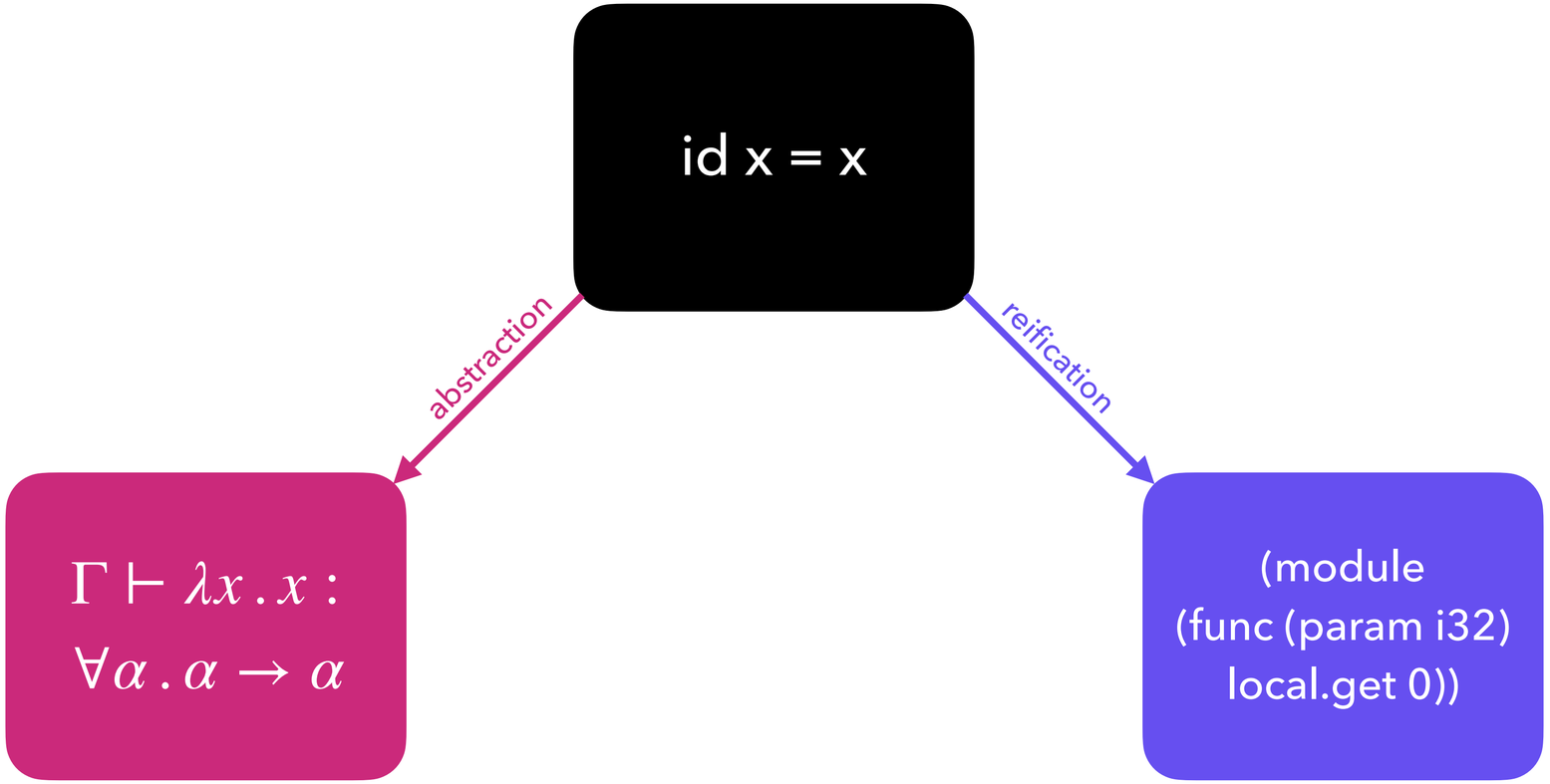
The Golden Age of PL Research*

- **Syntactic type soundness** made the proofs of type safety easy;
- **Java** popularized garbage collection and parametric polymorphism;
- **Dynamic languages** introduced first-class functions to the masses;
- **SAT and SMT solvers** became a building block for many PL tools;
- **Separation logic** enabled modular reasoning about imperative programs;
- **Mature proof assistants** haven been applied to broad areas, such as compilers, operating systems, and even the four color theorem;
- **……**

[*] Neel Krishnaswami. https://semantic-domain.blogspot.com/2022/09/the-golden-age-of-pl-research.html

What's PL Research About?

Theory, Design, Implementation, and Application



What's My Research About?

Compositional Programming

- **Design of the CP language:** advocating a new statically-typed modular programming paradigm called *Compositional Programming*. [TOPLAS'21]
- **Metatheory of the core calculus:** type soundness and semantic determinism are proven using Coq [ECOOP'22], later extended with references [OOPSLA'24]
- **Application to embedded DSLs:** *compositional embeddings* naturally solves challenges like the Expression Problem. [OOPSLA'22]
- **Implementation of CP:** efficient compilation to JavaScript. [APLAS'23 SRC]

[TOPLAS'21] Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. Compositional Programming.
[ECOOP'22] Andong Fan, Xuejing Huang, Han Xu, Yaozhu Sun, and Bruno C. d. S. Oliveira. Direct Foundations for Compositional Programming.
[OOPSLA'22] Yaozhu Sun, Utkarsh Dhandhania, and Bruno C. d. S. Oliveira. Compositional Embeddings of Domain-Specific Languages.
[APLAS'23 SRC] Yaozhu Sun, Xuejing Huang, and Bruno C. d. S. Oliveira. Separate Compilation of Compositional Programming via Extensible Records.
[OOPSLA'24] Wenjia Ye, Yaozhu Sun, and Bruno C. d. S. Oliveira. Imperative Compositional Programming.

The Expression Problem

Two-Dimensional Extensibility

```haskell
data Exp = Lit Int


eval :: Exp → Int
eval (Lit   n) = n
```

Haskell

The Expression Problem

Two-Dimensional Extensibility

```haskell
data Exp = Lit Int


eval :: Exp → Int
eval (Lit   n) = n


print :: Exp → String
print (Lit   n) = show n
```

modular

Haskell

The Expression Problem

Two-Dimensional Extensibility

```haskell
data Exp = Lit Int
         | Add Exp Exp

eval :: Exp → Int
eval (Lit   n) = n
eval (Add l r) = eval l + eval r

print :: Exp → String
print (Lit   n) = show n
print (Add l r) = print l
        ++ "+" ++ print r
```

non-modular

non-modular

non-modular

Haskell

The Expression Problem

Two-Dimensional Extensibility

```haskell
data Exp = Lit Int
         | Add Exp Exp

eval :: Exp → Int
eval (Lit   n) = n
eval (Add l r) = eval l + eval r

print :: Exp → String
print (Lit   n) = show n
print (Add l r) = print l
        ++ "+" ++ print r
```

Haskell

```scala
trait Exp:
  def eval:  Int

class Lit(n: Int) extends Exp:
  def eval  = n
```

Scala

The Expression Problem

Two-Dimensional Extensibility

```haskell
data Exp = Lit Int
         | Add Exp Exp

eval :: Exp → Int
eval (Lit   n) = n
eval (Add l r) = eval l + eval r

print :: Exp → String
print (Lit   n) = show n
print (Add l r) = print l
        ++ "+" ++ print r
```

Haskell

modular

```scala
trait Exp:
  def eval:  Int

class Lit(n: Int) extends Exp:
  def eval  = n

class Add(l: Exp, r: Exp) extends Exp:
  def eval  = l.eval + r.eval
```

Scala

The Expression Problem

Two-Dimensional Extensibility

```haskell
data Exp = Lit Int
         | Add Exp Exp

eval :: Exp → Int
eval (Lit   n) = n
eval (Add l r) = eval l + eval r

print :: Exp → String
print (Lit   n) = show n
print (Add l r) = print l
        ++ "+" ++ print r
```

Haskell

non-modular

non-modular

non-modular

```scala
trait Exp:
  def eval:  Int
  def print: String

class Lit(n: Int) extends Exp:
  def eval  = n
  def print = n.toString

class Add(l: Exp, r: Exp) extends Exp:
  def eval  = l.eval + r.eval
  def print = l.print + "+" + r.print
```

Scala

Non-Solution: Transposing Dimensions

Visitor Pattern

```
trait Exp:
  def accept[T](v: Visitor[T]): T
```

```
class Lit(n: Int) extends Exp:
  def accept[T](v: Visitor[T]) =
    v.visitLit(n)
```

```
trait Visitor[T]:
  def visitLit(n: Int): T
```

```
object EvalVisitor extends Visitor[Int]:
  def visitLit(n: Int) = n
```

Non-Solution: Transposing Dimensions

Visitor Pattern

```
trait Exp:
  def accept[T](v: Visitor[T]): T
```

```
class Lit(n: Int) extends Exp:
  def accept[T](v: Visitor[T]) =
    v.visitLit(n)
```

```
trait Visitor[T]:
  def visitLit(n: Int): T
```

```
object EvalVisitor extends Visitor[Int]:
  def visitLit(n: Int) = n
```

modular

```
object PrintVisitor extends Visitor[String]:
  def visitLit(n: Int) = n.toString
```

Non-Solution: Transposing Dimensions

Visitor Pattern

```
trait Exp:
  def accept[T](v: Visitor[T]): T
```

non-modular

```
class Lit(n: Int) extends Exp:
  def accept[T](v: Visitor[T]) =
    v.visitLit(n)
```

non-modular

```
class Add(l: Exp, r: Exp) extends Exp:
  def accept[T](v: Visitor[T]) =
    v.visitAdd(l, r)
```

non-modular

modular

```
trait Visitor[T]:
  def visitLit(n: Int): T
  def visitAdd(l: Exp, r: Exp): T
```

```
object EvalVisitor extends Visitor[Int]:
  def visitLit(n: Int) = n
  def visitAdd(l: Exp, r: Exp) =
    l.accept(this) + r.accept(this)
```

```
object PrintVisitor extends Visitor[String]:
  def visitLit(n: Int) = n.toString
  def visitAdd(l: Exp, r: Exp) =
    l.accept(this) + "+" + r.accept(this)
```

A Well-Known Solution in Haskell

Tagless-Final Embedding

```haskell
class Exp repr where
  lit :: Int → repr

instance Exp Int where
  lit n = n

instance Exp String where
  lit = show
```

```haskell
class Exp' repr where
  add :: repr → repr → repr

instance Exp' Int where
  add l r = l + r

instance Exp' String where
  add l r = l ++ "+" ++ r
```

```haskell
e :: (Exp repr, Exp' repr) ⇒ repr
e = add (lit 0) (lit 1)
```

Dependencies between Operations

The Third Dimension of the Expression Problem

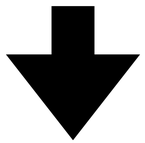
```haskell
data EvalPrint = EP { eval  :: Int
                    , print :: String
                    }


instance Exp' EvalPrint where
  add l r = EP { eval  = eval l + eval r
               , print = if eval l == 0 then print r
                         else if eval r == 0 then print l
                         else print l ++ "+" ++ print r
               }
```

Dependencies between Operations

The Third Dimension of the Expression Problem

```
instance Exp' EvalPrint where
  add l r = EP { eval  = eval l + eval r
               , print = if eval l == 0 then print r
                         else if eval r == 0 then print l
                         else print l ++ "+" ++ print r
               }
```



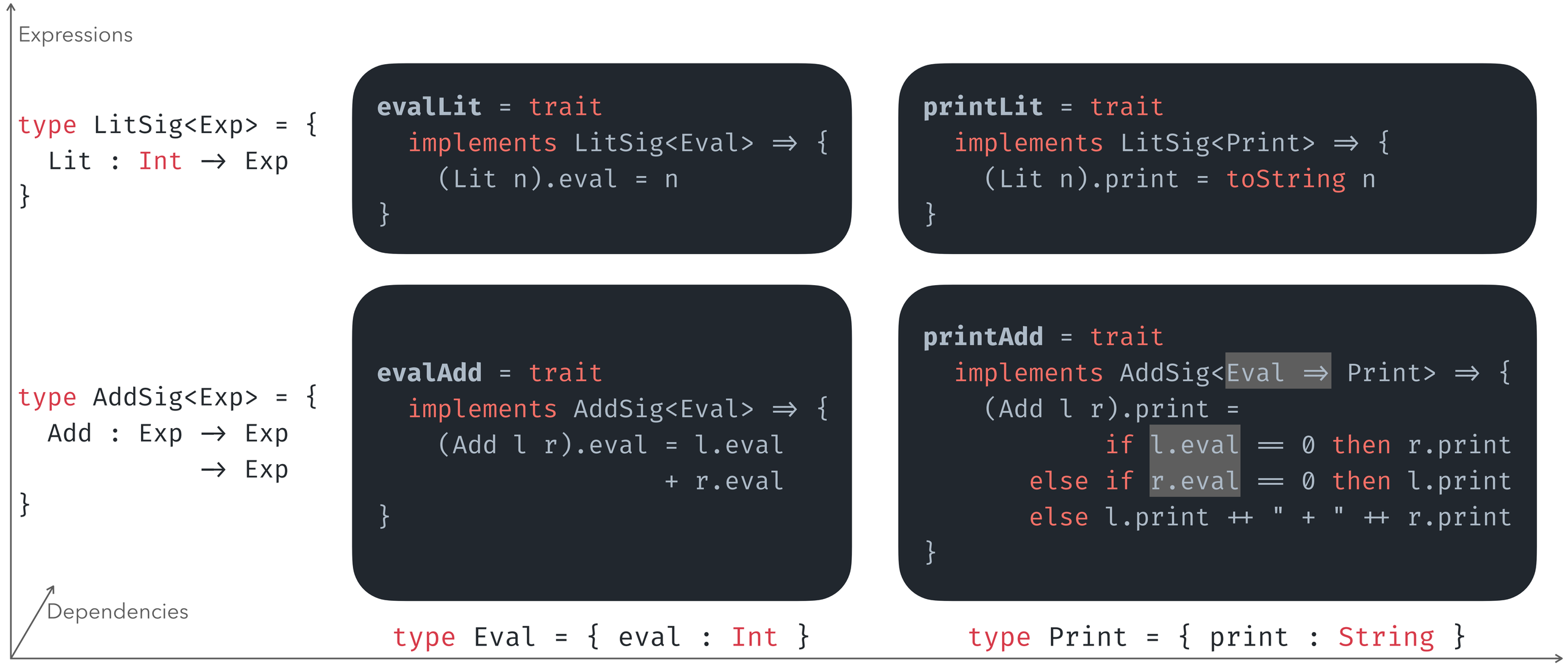
```
instance Exp' Int where
  add l r = l + r
```

What to write?

```
instance Exp' String where
  add l r = if ??? == 0 then r
            else if ??? == 0 then l
            else l ++ "+" ++ r
```

CP = Modularity × Extensibility^3

CP's Solution to the Expression Problem



Putting Everything Together

Merging Five Traits

intersection type

```
type ExpSig<Exp> = LitSig<Exp> & AddSig<Exp>

repo Exp = trait [self: ExpSig<Exp>] ⇒ { e = Add (Lit 0) (Lit 1) }

merged = new repo @(Eval&Print) , evalLit , evalAdd , printLit , printAdd

merged.e.eval   ⟶  1
merged.e.print  ⟶ "1"
```

merge operator

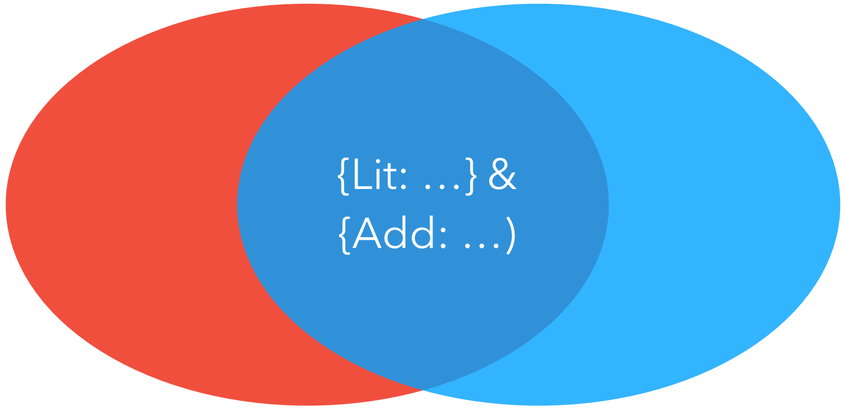
Theories behind CP (1)

F_i^+ [ECOOP'22]

- **Intersection type**, e.g. `LitSig<Exp> & AddSig<Exp>`
  - Multi-field record type = Intersection of single-field record types

    `{Lit: Int → Exp; Add: Exp → Exp → Exp} = {Lit: ……} & {Add: ……}`
  - Function overloading = Intersection of function types

    `(Int → String) & (Double → String) <: Int|Double → String`



Theories behind CP (2)

F_i^+ [ECOOP'22]

- **Merge operator**, e.g. `evalLit , evalAdd , printLit , printAdd`
  (A syntactically explicit introduction rule for intersection types)
  - Multi-field record type
    `{Lit = ……} , {Add = ……} : {Lit: ……} & {Add: ……}`
  - Function overloading
    `Int.toString , Double.toString : Int|Double → String`

$$\frac{e : A \qquad e : B}{e : A \,\&\, B} \quad \Rightarrow \quad \frac{e_1 : A \qquad e_2 : B}{e_1, e_2 : A \,\&\, B}$$

Theories behind CP (3)

F_i^+ [ECOOP'22]

- **Disjointness**, e.g. ⭕ `{Lit: ……} * {Add: ……}`

  ❌ ~~`{Lit: Int} * {Lit: Int}`~~

  - to avoid semantic ambiguity, e.g. `({Lit = 1},{Lit = 2}).Lit`
  - to be commutative and associative, i.e. $x, y = y, x$ and $(x, y), z = x, (y, z)$

- **Disjoint polymorphism**, w/ disjointness constraints ($\forall \alpha \star \tau \,.\, \ldots\ldots$)
  reminiscent of Harper and Pierce's row polymorphism ($\forall \alpha \# R \,.\, \ldots\ldots$)
  - powerful enough to encode kernel $F_{<:}$

    $\forall \alpha <: \sigma \,.\, \alpha \quad \equiv \quad \forall \alpha \,.\, \alpha \,\&\, \sigma$

Theories behind CP (4)

F_i^+ [ECOOP'22]

- **Distributive subtyping**, e.g. $\{\ell : A\} \,\&\, \{\ell : B\} <: \{\ell : A \,\&\, B\}$

$$(A \to B) \,\&\, (A \to C) <: A \to B \,\&\, C$$

  - key to nested composition, e.g.

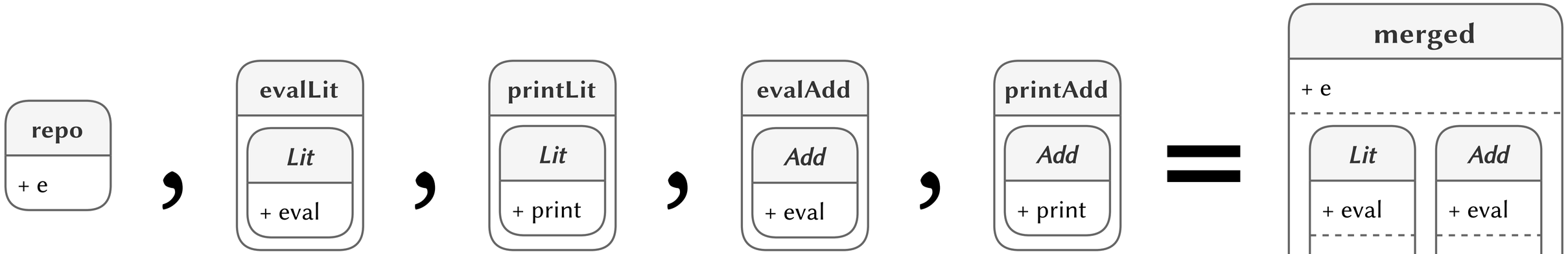
```
    { Lit n = {eval = n} } , { Lit n = {print = toString n} }
  : { Lit: Int → Eval } & { Lit: Int → Print }
<: { Lit: Int → Eval&Print}
```

  - coercion is implicitly inserted, e.g.

```
    { Lit n = {eval = n} } , { Lit n = {print = toString n} }
→ { Lit n = {eval = n; print = toString n} }
```

Visualization of Nested Composition

a.k.a. Family Polymorphism



Coercive Subtyping:

```
{Lit: I→E} & {Lit: I→P} & {Add: E→E→E} & {Add: P→P→P}
```

<:

```
{ Lit:          I → E&P
; Add: E&P → E&P → E&P }
```

Elaboration Semantics?

Alternatively

Type-Directed Operational Semantics

Upcasting + merge elimination (distributive applications / projections)

(First-Class) Traits

CP [TOPLAS'21]

- **Traits** are object generators, e.g. `trait ⇒ { x = 1 }`
  is encoded as `\(self: Top) → { x = 1 }`
- The function parameter is a **self-reference**, e.g.
  `trait [self: {x: Int}] ⇒ { x = 1; y = self.x }`
  is encoded as `\(self: {x: Int}) → { x = 1; y = self.x }`
- **Instantiation** of an object from a trait is a fixpoint, e.g. `new t`
  is encoded as `fix self. t self`
- **Inheritance** is modeled as merges, e.g. `trait inherits t ⇒ { …… }`
  is encoded as `\(self: Top) → let super = t self in super , { …… }`

Dependency Injection

CP [TOPLAS'21]

- **Self-type annotations**

```
trait [self : ExpSig<Exp>] ⇒ { e = …… self.Add …… self.Lit …… }
```

- **Compositional interface type refinement**

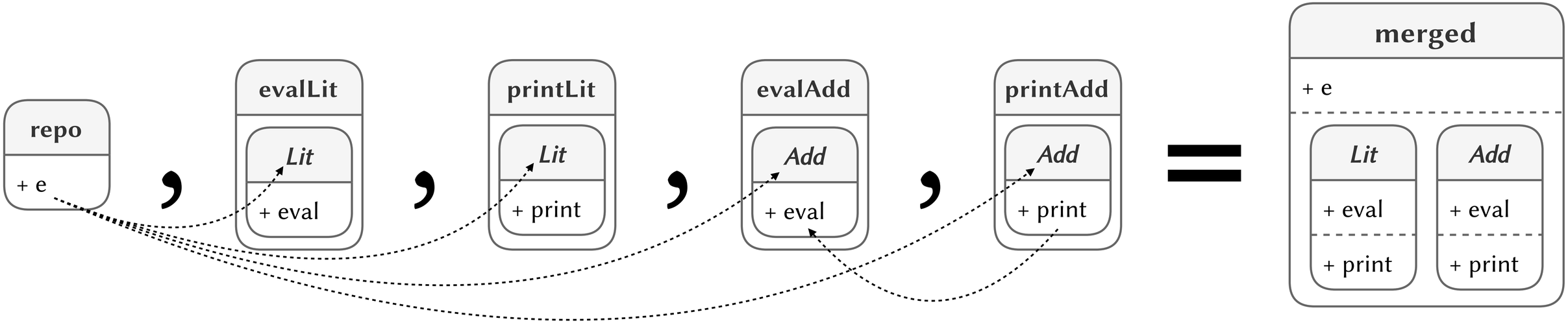
```
trait implements AddSig<Eval ⇒ Print> ⇒ {
  (Add l r).print = …… l.eval …… r.eval ……
}
```

- **Combining both**

```
trait implements AddSig<Eval ⇒ Print> ⇒ {
  [self]@(Add l r).print = …… self.eval ……
}
```

Dependency Injection

CP [TOPLAS'21]



Demo

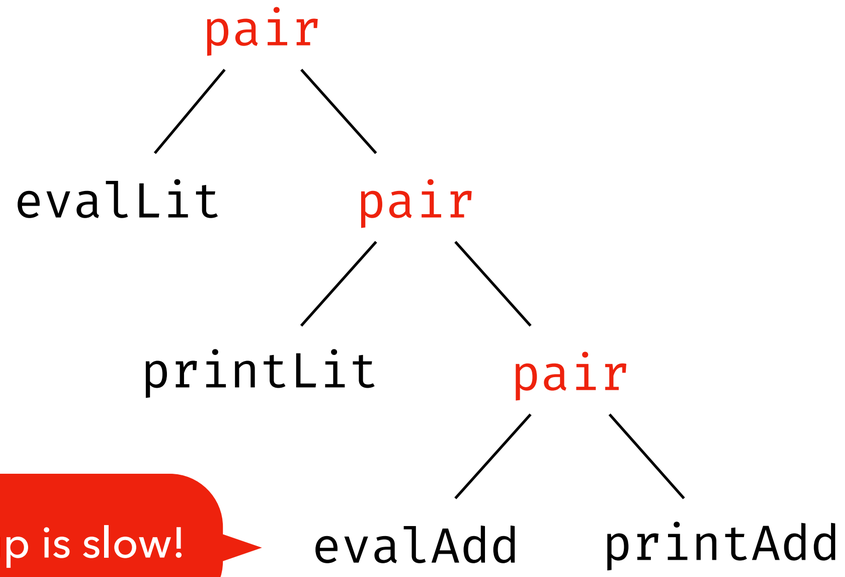
https://plground.org/

Recent Developments (1)

Compiling Merges

```
merged = evalLit , printLit , evalAdd , printAdd
```

their types are disjoint



lookup is slow!

```
{ LitSig<Eval>   ⟾ evalLit
; LitSig<Print>  ⟾ printLit
; AddSig<Eval>   ⟾ evalAdd
; AddSig<Print>  ⟾ printAdd
}
```

their indices never conflict

Recent Developments (1)

Compiling CP to JavaScript [APLAS'23 SRC]

- *[Key Idea]* Compiling to **type-indexed records** (or, objects in JavaScript)

  `48 , true  ⇝  { Int ⟹ 48; Bool ⟹ true }`

  - dynamic merge operator ⇝ runtime record concatenation
  - coercion to supertype ⇝ record filtering or reconstruction
- *[Optimization]* Identifying **equivalent types** to avoid redundant coercions
  - top-like types are all equivalent (empty records)
  - intersection types are equivalent up to permutation, deduplication, and top-like type removal (records are unordered and labels are unique)

Recent Developments (2)

Intersection Subtyping with References

$$\frac{e : A}{\text{ref } e : \text{Ref } A}\ \text{T-Ref}$$

$$\dfrac{\dfrac{}{1 : \text{Nat}}\ \text{T-Ref} \qquad \dfrac{}{1 : \text{Pos}}\ \text{T-Ref}}{\text{ref } 1 : \text{Ref Nat} \,\&\, \text{Ref Pos}}\ \text{T-Intro}$$

[RUNTIME ERROR]
0 does not have type Pos!

```
let x = ref 1 : Ref Nat & Ref Pos;
x := 0;
!x : Pos
```

Type unsound!

Recent Developments (2)

Bidirectional Typing to the Rescue [OOPSLA'24]

$$\frac{e \Rightarrow A}{\text{ref } e \Rightarrow \text{Ref } A}\ \text{T-Ref}$$

$$\text{T-Intro}\ \frac{\text{T-Ref}\ \dfrac{1 \Leftarrow \text{Nat}}{\text{ref } 1 \Rightarrow \text{Ref Nat}} \qquad \text{T-Ref}\ \dfrac{1 \Rightarrow \text{Pos}}{\text{ref } 1 \Rightarrow \text{Ref Pos}}}{\text{ref } 1 \Rightarrow \text{Ref Nat \& Ref Pos}}$$

[TYPE ERROR]
"ref 1" does not have type "Ref Nat"

```
let x = ref 1 : Ref Nat & Ref Pos;
x := 0;
!x : Pos
```

Type sound!

Summary

- **Design of the CP language:** advocating a new statically-typed modular programming paradigm called *Compositional Programming*. [TOPLAS'21]
- **Metatheory of the core calculus:** type soundness and semantic determinism are proven using Coq [ECOOP'22], later extended with references [OOPSLA'24]
- **Application to embedded DSLs:** *compositional embeddings* naturally solves challenges like the Expression Problem. [OOPSLA'22]
- **Implementation of CP:** efficient compilation to JavaScript. [APLAS'23 SRC]

[TOPLAS'21] Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. Compositional Programming.
[ECOOP'22] Andong Fan, Xuejing Huang, Han Xu, Yaozhu Sun, and Bruno C. d. S. Oliveira. Direct Foundations for Compositional Programming.
[OOPSLA'22] Yaozhu Sun, Utkarsh Dhandhania, and Bruno C. d. S. Oliveira. Compositional Embeddings of Domain-Specific Languages.
[APLAS'23 SRC] Yaozhu Sun, Xuejing Huang, and Bruno C. d. S. Oliveira. Separate Compilation of Compositional Programming via Extensible Records.
[OOPSLA'24] Wenjia Ye, Yaozhu Sun, and Bruno C. d. S. Oliveira. Imperative Compositional Programming.

Q&A