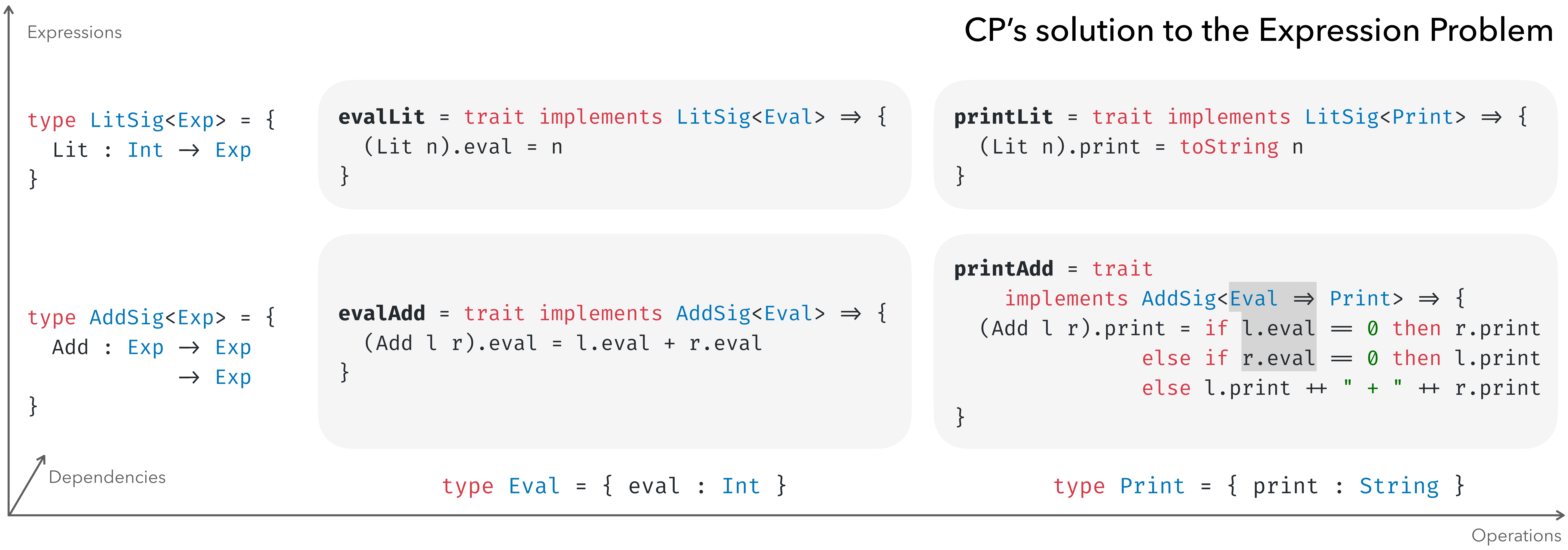
Separate Compilation for Compositional Programming via Extensible Records

Yaozhu Sun <yzsun@cs.hku.hk>
Supervised by Bruno C. d. S. Oliveira

The University of Hong Kong



CP's solution to the Expression Problem

```
type LitSig<Exp> = {
  Lit : Int → Exp
}
```

```
evalLit = trait implements LitSig<Eval> ⇒ {
  (Lit n).eval = n
}
```

```
printLit = trait implements LitSig<Print> ⇒ {
  (Lit n).print = toString n
}
```

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

```
evalAdd = trait implements AddSig<Eval> ⇒ {
  (Add l r).eval = l.eval + r.eval
}
```

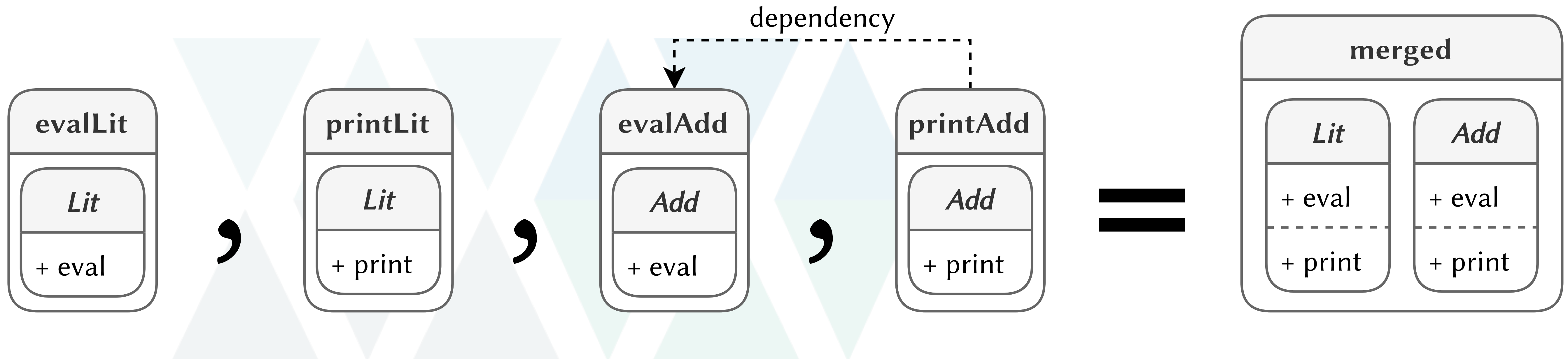
```
printAdd = trait
    implements AddSig<Eval ⇒ Print> ⇒ {
  (Add l r).print = if l.eval == 0 then r.print
                    else if r.eval == 0 then l.print
                    else l.print ++ " + " ++ r.print
}
```

```
type Eval = { eval : Int }
```

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

‣ Challenge of feature modularity: modular type checking & separate compilation

CP's solution:
nested trait composition
by the *merge operator*
(denoted by a comma)



‣ How to compile merged features?

Previous work on elaboration of intersection types:
compiling merges to nested pairs, e.g.

```
(evalLit, (printLit, (evalAdd, printAdd)))
```

‣ Q1: Why can we compile this way?

Our type system guarantees that the merged terms have *disjoint* types, so there must be no conflict between type indices.

```
LitSig<Eval> * LitSig<Print> * AddSig<Eval> * AddSig<Print>
```

‣ Challenges in compiling CP

Challenge 1: nested composition

Like family polymorphism, nested traits are recursively composed in CP. To achieve this, subtyping is enhanced with distributivity rules of records, functions or traits over intersection types.

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

Challenge 2: dynamic inheritance

Unlike traditional OOP, inheritance hierarchies are not statically known in CP, so feature composition is delayed until runtime.

```
t2 (t1 : Trait<Feature>) = trait inherits t1 ⇒ { … }
```

Challenge 3: parametric polymorphism

Record labels cannot be statically computed for polymorphic types. First-class labels are needed to handle type instantiation.

Our compilation scheme:
compiling merges to *type-indexed* records, e.g.

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

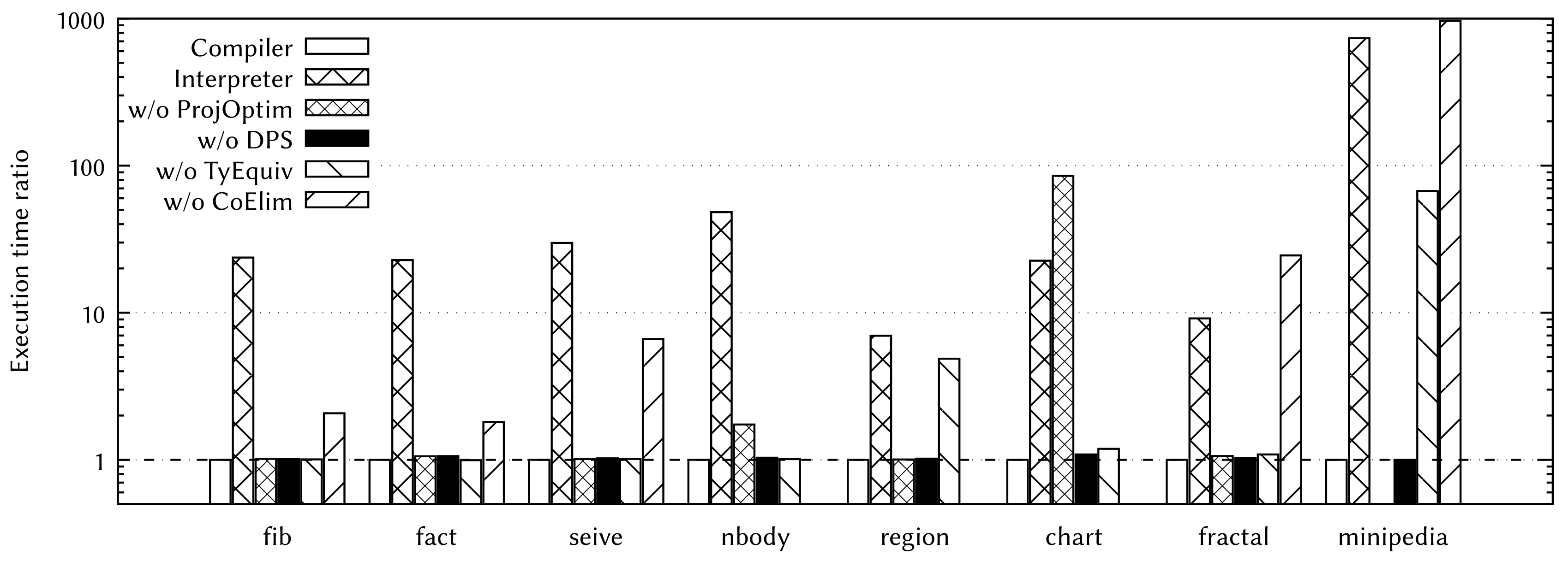
‣ Q2: Why do we choose to compile this way?

i) Looking up a component by type indices is much faster than doing that in nested pairs (linear time in the worst case).
ii) Type-indexed records require fewer coercions because some source terms compile to equivalent records.

‣ Identifying equivalent types

i) top-like types are all equivalent (empty records)
ii) intersection types are equivalent up to permutation, deduplication, and top-like type removal (records are unordered and labels are unique)

‣ Evaluating CP-specific compiler optimizations



The benchmarks show that the most important optimization is to eliminate redundant coercions for subtyping between equivalent types.

Artifact CP compiler implementation targeting JavaScript