大学共同利用機関法人 情報・システム研究機構
国立情報学研究所
National Institute of Informatics

INSTITUT
DE RECHERCHE
EN INFORMATIQUE
FONDAMENTALE

香 港 大 學
THE UNIVERSITY OF HONG KONG

Type-Safe Compilation of Dynamic Inheritance via Merging

Yaozhu Sun, Xuejing Huang, Bruno C. d. S. Oliveira

18 October 2025

Dynamic Inheritance

in JavaScript

Mixin Pattern

Class as a parameter

Dynamic inheritance

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

Dynamic Inheritance

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

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

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!

Overloading vs Merging

- **Implicit overriding is dangerous**,
  both for type safety and semantics (e.g. *fragile base class problem*).

- We advocate a **trait model with merging**, which
  - keeps both behaviors if there is no conflict, and
  - requires explicit resolution if there are conflicts.

Type-Safe Merging

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 }
```

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

Unambiguous Merging

in CP

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

We express absence by disjointness!

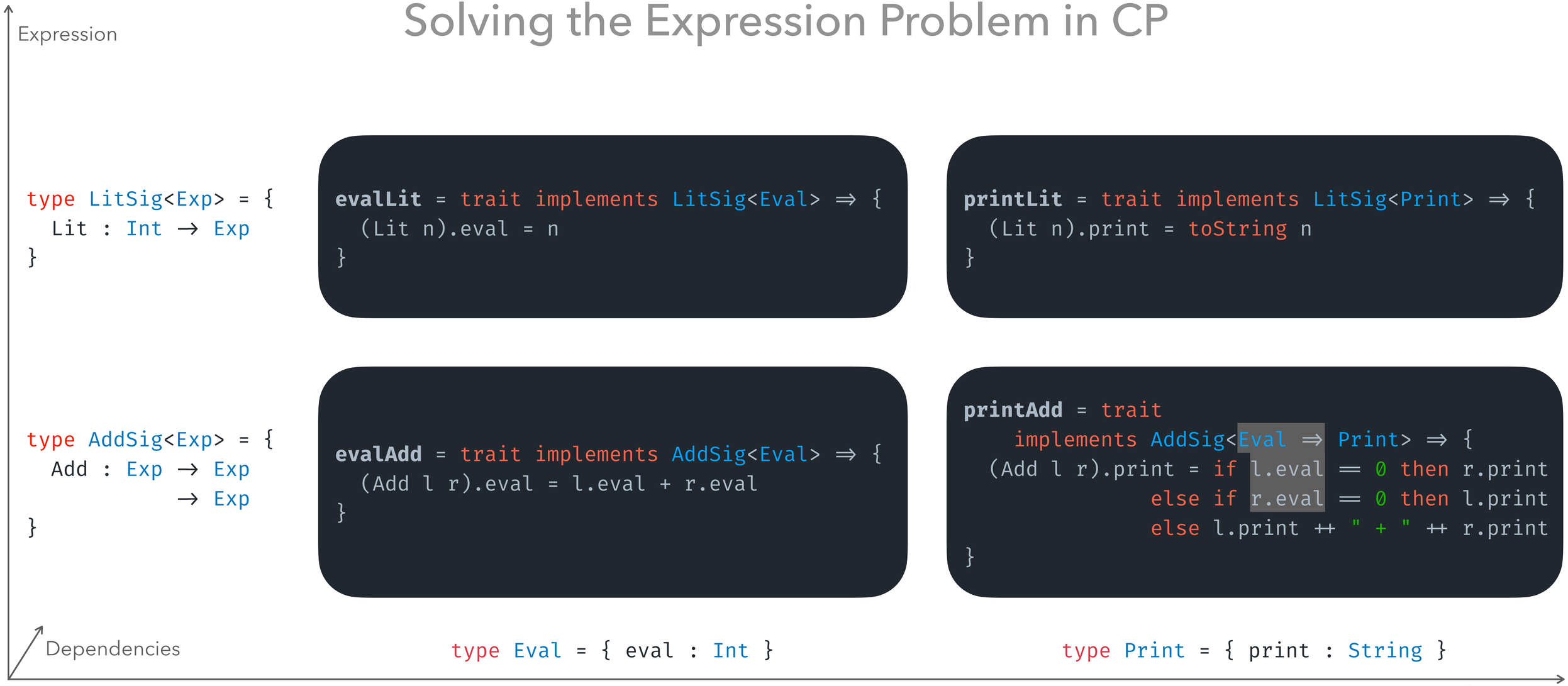
```
mkA = trait [this: { m: String; n: String }] ⇒ {
  m = "foobar";
  n = toUpperCase this.m;
};

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

The two “m” fields conflict because they have non-disjoint type (~~String * String~~).

Why Merging Matters?

Solving the Expression Problem in CP

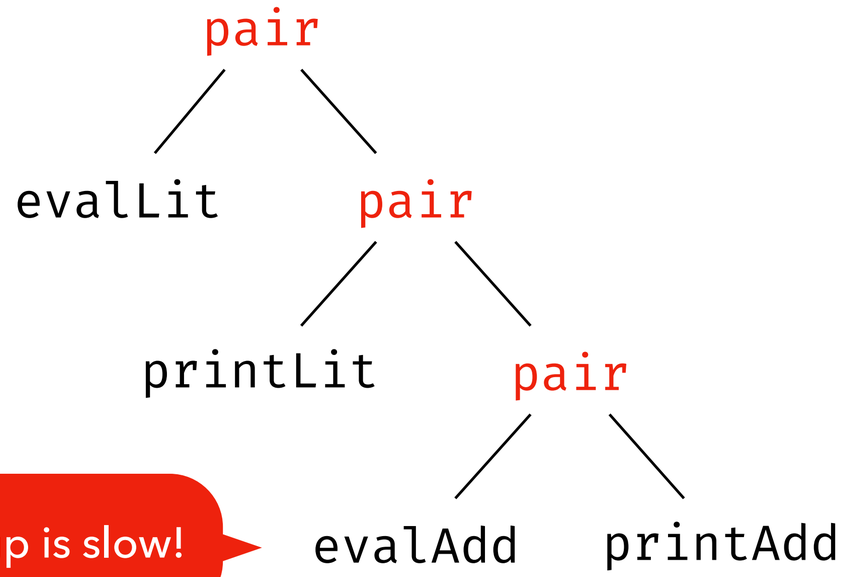


Compiling Merges

Previous Work vs Our Scheme

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

their types are disjoint



lookup is slow!

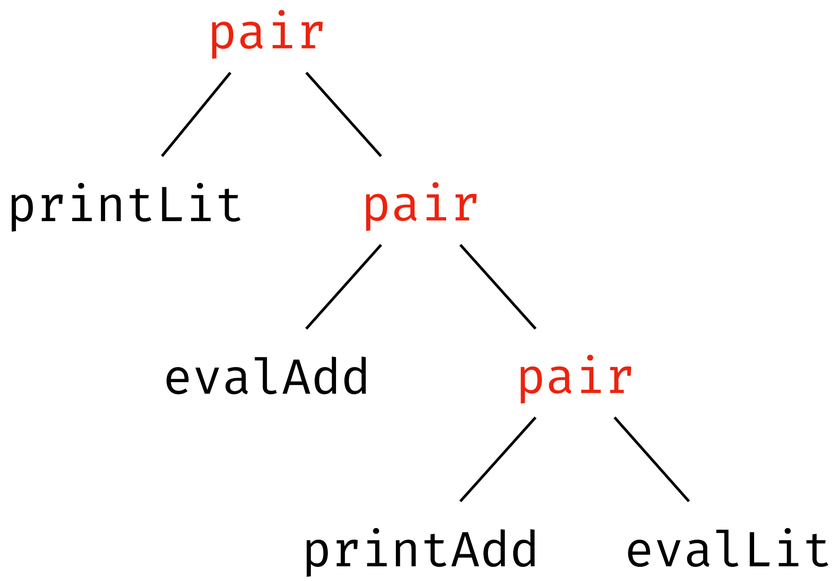
```
{ LitSig<Eval>   ⟾ evalLit
; LitSig<Print>  ⟾ printLit
; AddSig<Eval>   ⟾ evalAdd
; AddSig<Print>  ⟾ printAdd
}
```

their indices never conflict

Order-Sensitivity

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

different pairs!



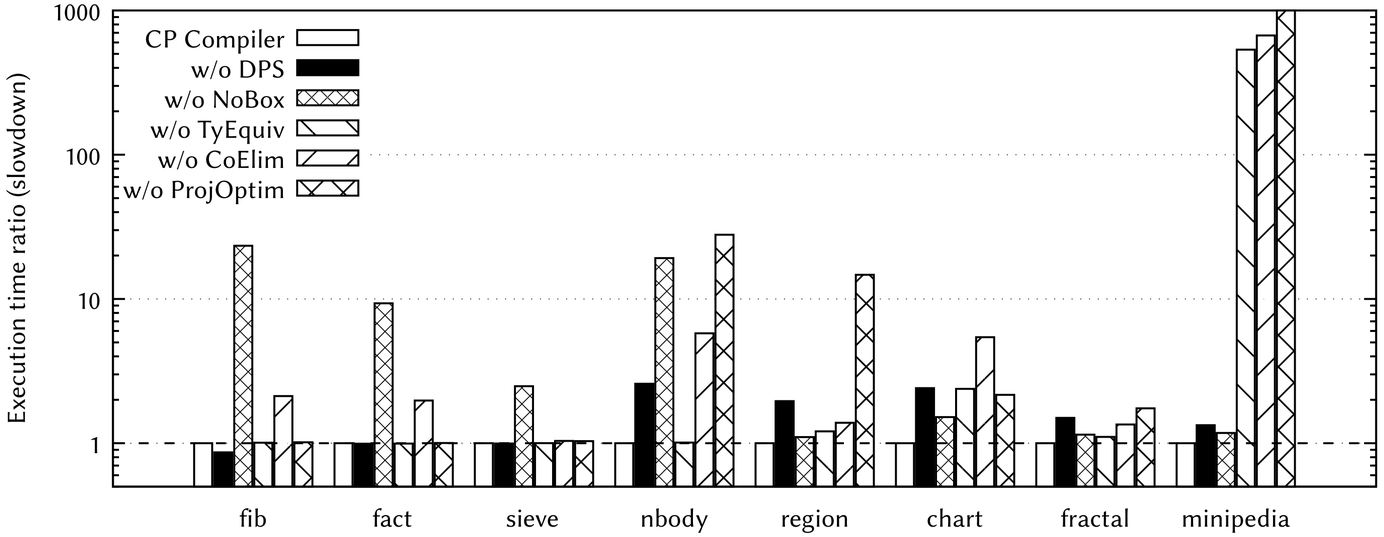

```
{ LitSig<Print>  ⟾ printLit
; AddSig<Eval>   ⟾ evalAdd
; AddSig<Print>  ⟾ printAdd
; LitSig<Eval>   ⟾ evalLit
}
```

equivalent records

Key Ideas

- Compiling to **extensible records** (with **type indices** as labels):
  - dynamic merge operator ⇝ runtime record concatenation
  - coercion to supertype ⇝ record filtering or reconstruction
- Coercions between *equivalent types* can be avoided:
  - 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)

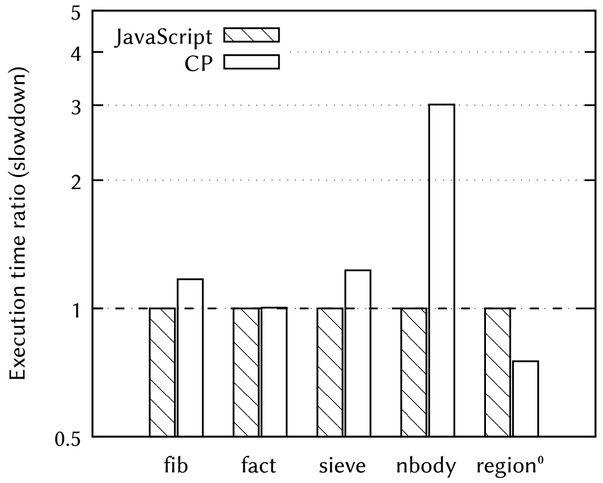
Benchmarks

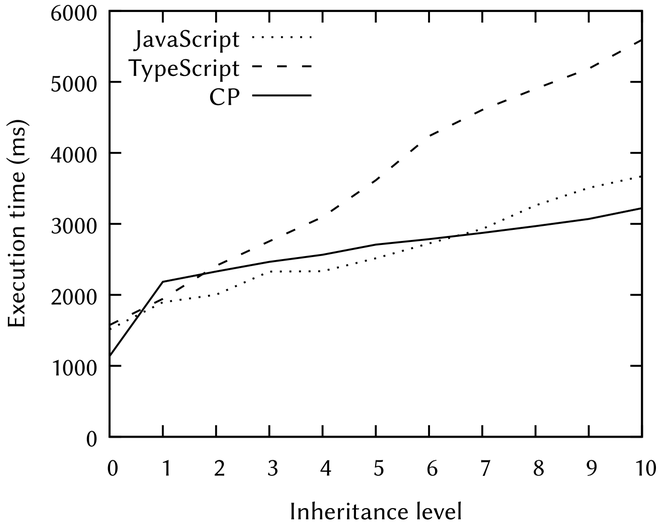


CP is slightly slower than handwritten JavaScript for general computation.

CP has the *most* smooth curve w.r.t. inheritance level.

All optimizations work in practice, while the *most* important one is the coercion elimination for equivalent types.





More in the Paper…

1. Complete examples for the type-safety issue in TypeScript
2. Compilation scheme for parametric polymorphism
3. Formalization of the compilation scheme with the Rocq prover
4. A prototype CP compiler targeting JavaScript
5. Detailed discussions on the implementation and more optimizations

[Artifact URL] https://github.com/yzyzsun/CP-next/tree/toplas

Thank you!