香 港 大 學
THE UNIVERSITY OF HONG KONG

APLAS'23 SRC

Separate Compilation of Compositional Programming

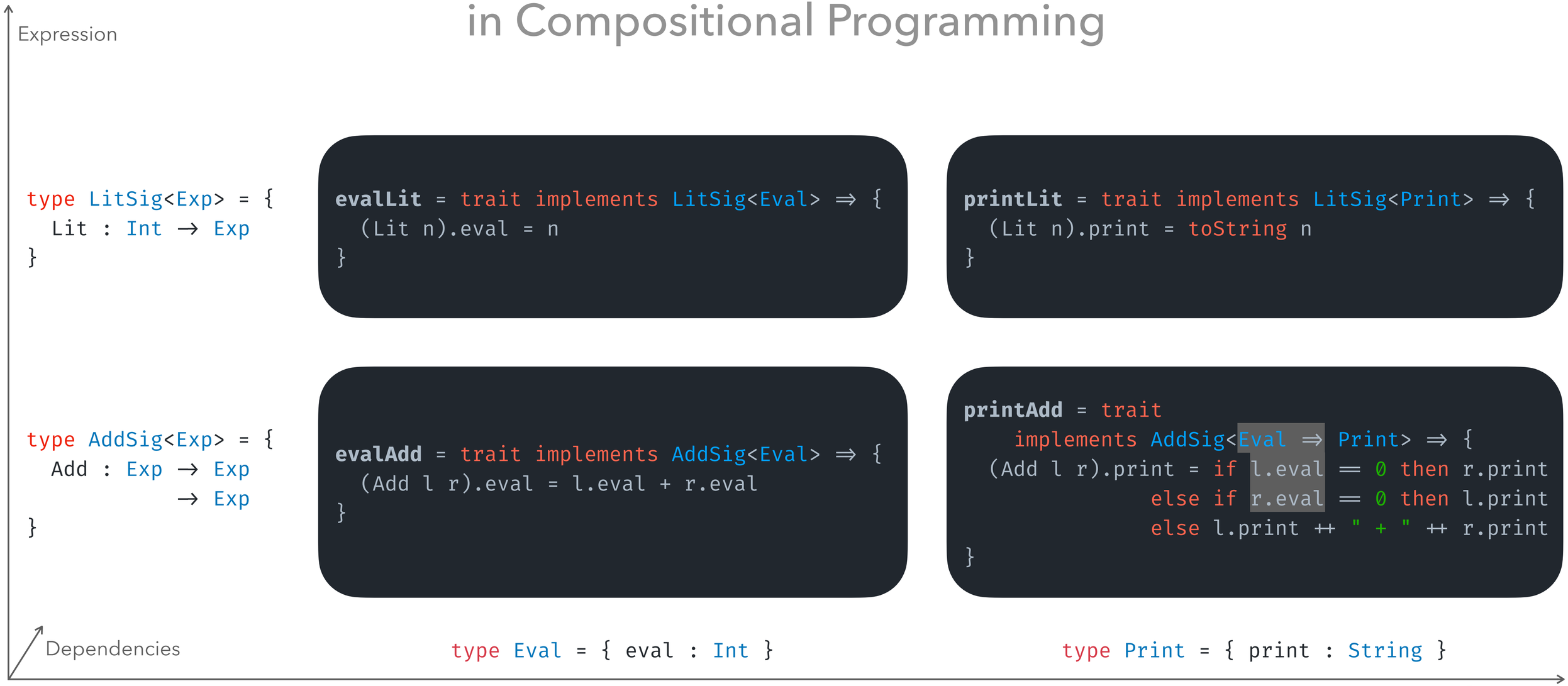
via Extensible Records

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

28 November 2023

Expression Problem

in Compositional Programming



“Most feature-oriented implementation mechanisms lack proper interfaces and support neither modular type checking nor separate compilation.”

– Christian Kästner, Sven Apel, and Klaus Ostermann

Merging Features

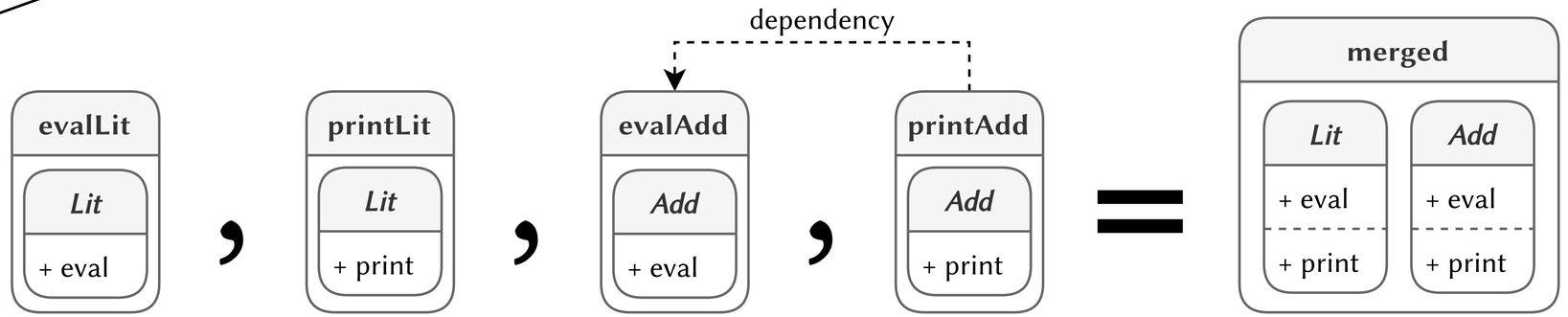
from Two Perspectives of Modularity

```
merged = trait implements LitSig<Eval&Print> & AddSig<Eval&Print> ⇒ {
  -- evalLit:
  (Lit   n).eval = n;
  -- evalAdd:
  (Add l r).eval = l.eval + r.eval;
  -- printLit:
  (Lit   n).print = toString n;
  -- printAdd:
  (Add l r).print = if l.eval == 0 then r.print
                    else if r.eval == 0 then l.print
                    else l.print ++ " + " ++ r.print
}
```

Information hiding

Cohesion

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

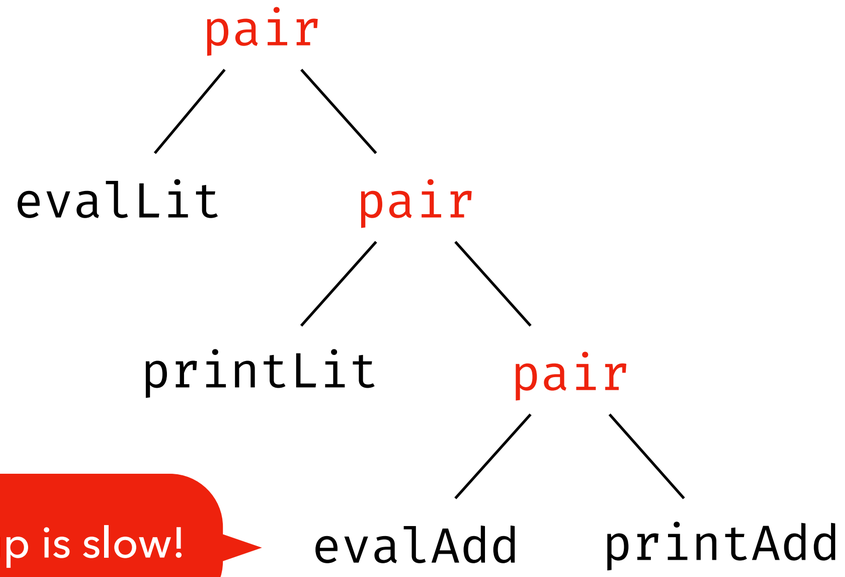


Compiling Merges

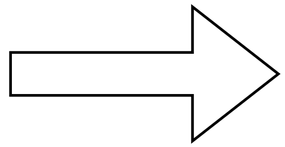
Previous Work vs Our Scheme

```
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

Challenges

- **Nested composition:**

```
let e = { Lit = \n → { eval = n } } , { Lit = \n → { print = toString n } }
in e.Lit 48
-- { Lit: Int → Eval } & { Lit: Int → Print } <: { Lit: Int → Eval&Print }
```

- **Dynamic inheritance:**

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

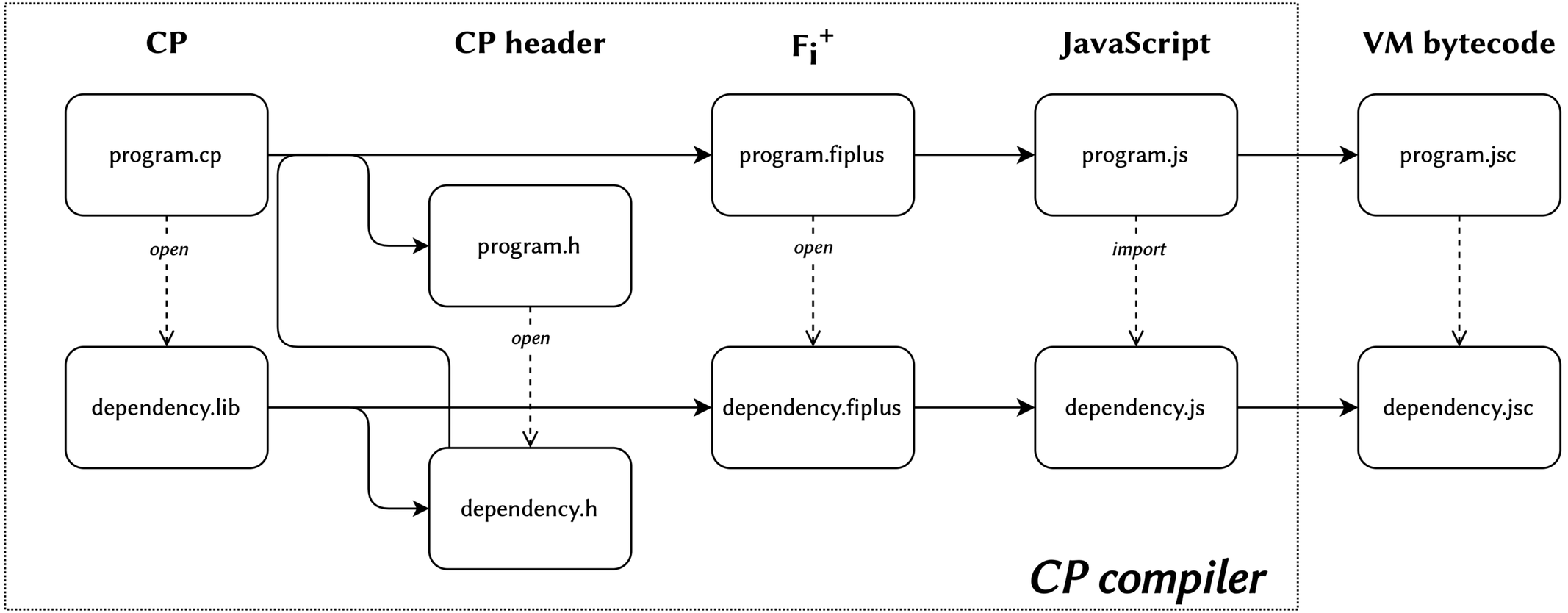
- **Parametric polymorphism:**

```
let poly = Λ A. { f = \(x:A) → x } in poly @Int
-- poly : forall A. { f : A → A }
```

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 (because their terms compile to equivalent records):
  - 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)

Flowchart of Separate Compilation



Implementation

SSA vs DPS

```
48 , true , chr 32
```

```
const $1 = { int: 48 };
const $2 = { bool: true };
const $3 = { ...$1, ...$2 };
const $4 = { int: 32 };
const $5 = $chr.fun_char($4);
const $6 = { ...$3, ...$5 };
```

Static Single Assignment

(6 objects, 2 merges)

```
const $1 = {};
$1.int = 48;
$1.bool = true;
const $2 = {};
$2.int = 32;
$chr.fun_char($2, $1);
```

Destination-Passing Style

(2 objects, 0 merges)

Implementation

CBV vs CBN

```
chr 32
```

```
const $2 = {};
$2.int = 32;
$chr.fun_char($2, $1);
```

Call by Value

```
const $2 = $chr.get;
const $3 = {
  get get() {
    const $4 = {};
    $4.int = 32;
    return $4;
  }
};
const $5 = {};
$2.fun_char($3, $1);
```

Call by Name

```
const $2 = $chr.get;
const $3 = {
  get get() {
    const $4 = {};
    $4.int = 32;
    delete this.get;
    return this.get = $4;
  }
};
const $5 = {};
$2.fun_char($3, $1);
```

Call by Need

Implementation

Parametric Polymorphism

```
poly = Λ (A * Int). \ (x : A) → x , 48
```

```
export const $poly = { get get() {
    const $1 = {};
    $1['forall_fun_(1&int)'] = function ($A, $2) {
      $2['fun_' + toIndex([ ...$A, 'int' ])] = function ($x, $3) {
        Object.assign($3, $x.get);
        $3.int = 48;
      };
    };  delete(this.get);  return this.get = $1;
} };
```

```
poly @(String & Bool) ("foo" , true)
```

```
const $4 = {};  const $5 = {};  const $6 = $poly.get;
$6['forall_fun_(1&int)']([ 'string', 'bool' ], $5);
const $7 = { get get() { /* code for ("foo" , true) */ } };
$5['fun_(bool&int&string)']($7, $4);
```

Implementation

CP-Specific Optimizations

- Eliminating redundant coercions for subtyping between equivalent types:

$$\frac{A \doteqdot B}{\epsilon : A <: B \twoheadrightarrow \epsilon}\,(?)$$

- Avoiding the insertion of coercions for record projections:

```
{x = 1; y = 2}.x  ⟿  ( {x = 1} , {y = 2} ÷ {x : Int} ).x
```

Benchmarks

The most important optimization is to eliminate redundant coercions for subtyping between equivalent types.

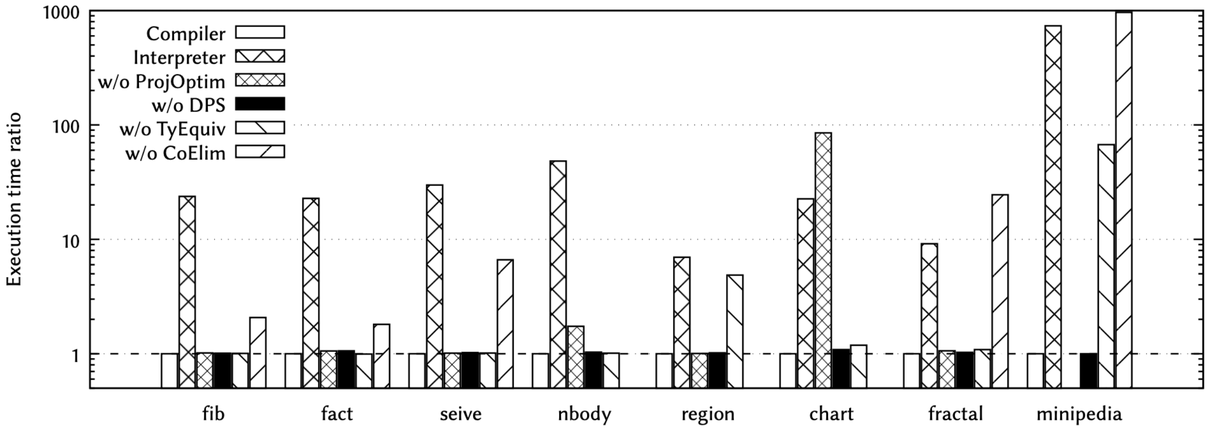
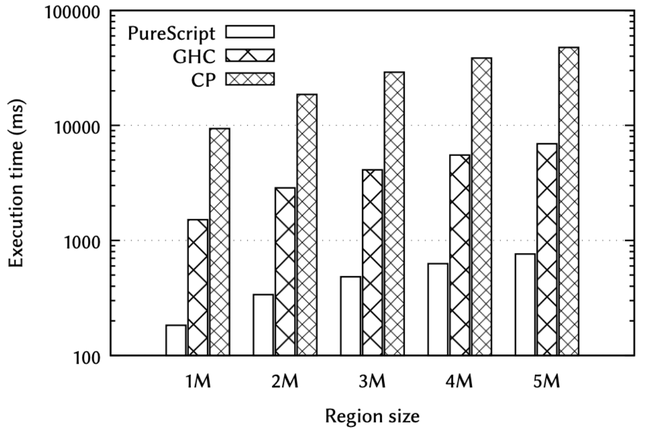


Fig. 1. Execution time ratios to the JavaScript code generated by the CP compiler for the benchmarks.‡

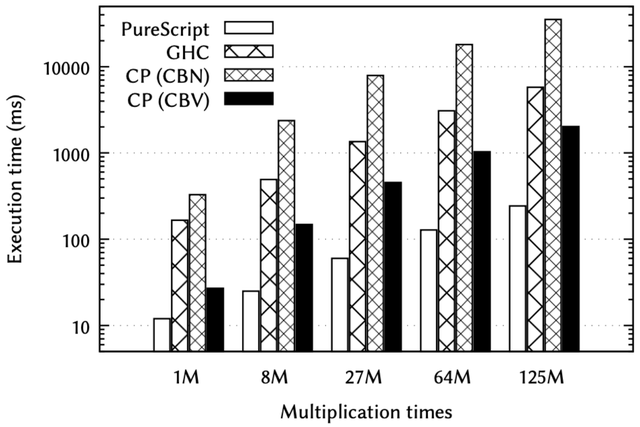
‡ No data for minipedia without the projection optimization (w/o ProjOptim) due to heap out of memory. No data for nbody, region, and chart without the coercion elimination (w/o CoElim) due to stack overflows.

CP compiler is 6–7× slower than GHC (JavaScript backend).



(a) Benchmark for the region DSL.

Call-by-value variant of CP compiler is ~3× faster than GHC (JS).



(b) Benchmark for the factorial calculations.

Fig. 2. Execution time of the compiled JavaScript code generated by different compilers.

Related Work

- Previous works on elaboration of intersection types [Dunfield 2014; Oliveira et al. 2016; Alpuim et al. 2017; Bi et al. 2018, 2019] translate merges to nested pairs, leading to inefficient merge lookups.
- *Type-indexed rows* [Shields and Meijer 2001] do not consider subtyping.
- The majority of *feature-oriented programming* (e.g. AHEAD, FeatureC++, FeatureHouse) basically does source-to-source transformations.
- Most languages/calculi with *virtual classes* and *family polymorphism* (e.g. BETA, Concord, CaesarJ, .FJ, vc, Tribe, Familia) do not support dynamic inheritance; gbeta [Ernst 2000] and Newspeak [Bracha et al. 2010] support it at the cost of separate compilation and static type checking respectively.

Q&A