Row and Bounded Polymorphism via Disjoint Polymorphism

Ningning Xie Bruno C. d. S. Oliveira
Xuan Bi Tom Schrijvers

香 港 大 學
THE UNIVERSITY OF HONG KONG

KU LEUVEN

In this paper

Object-Oriented Languages

Polymorphism

Subtyping

In this paper

Object-Oriented Languages

Bounded Polymorphism

Polymorphism

Subtyping

In this paper

Object-Oriented Languages

Polymorphism

Subtyping

Bounded Polymorphism

Row Polymorphism

In this paper

Object-Oriented Languages

Polymorphism

Subtyping

Bounded Polymorphism

Row Polymorphism

intersection types + Polymorphism

In this paper

Object-Oriented Languages

Polymorphism

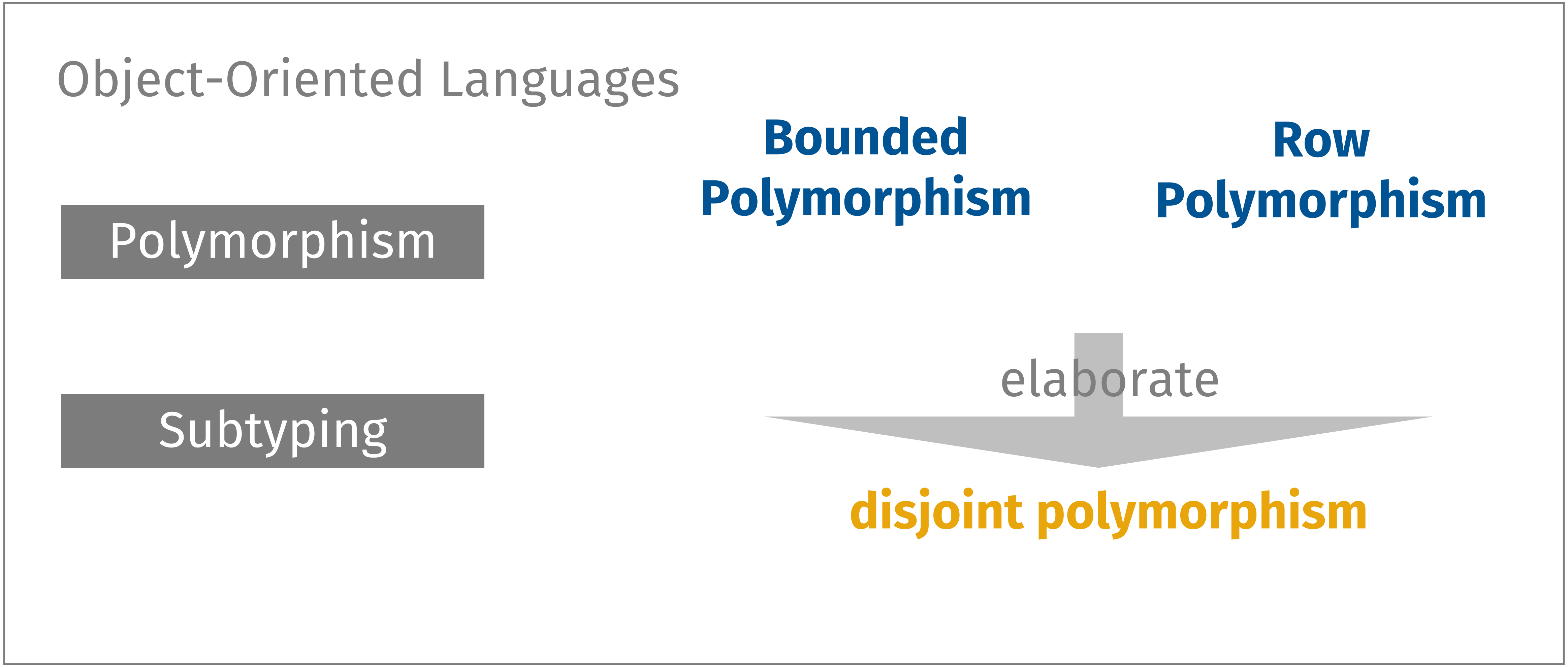
Subtyping

Bounded Polymorphism

Row Polymorphism

disjoint polymorphism

In this paper



In this paper

Object-Oriented Languages

Polymorphism

Subtyping

Bounded Polymorphism

kernel F<:
[Cardelli and Wegner 1985]

Row Polymorphism

λ||
[Harper, and Pierce 1991]

elaborate

disjoint polymorphism

Fi+
[Bi et al. 2019]

Intersection Types 101

Intersection types are useful to express *multiple interface inheritance*. They feature in *Scala*, *TypeScript*, *Ceylon*, *Flow*, ...

Intersection Types 101

Intersection types are useful to express *multiple interface inheritance*. They feature in *Scala*, *TypeScript*, *Ceylon*, *Flow*, ...

```
function extend<A, B>(first: A, second: B): A & B
```

Intersection Types 101

Intersection types are useful to express *multiple interface inheritance*. They feature in *Scala*, *TypeScript*, *Ceylon*, *Flow*, ...

```
function extend<A, B>(first: A, second: B): A & B
```

intersection type

Intersection Types 101

Intersection types are useful to express *multiple interface inheritance*. They feature in *Scala*, *TypeScript*, *Ceylon*, *Flow*, ...

```
function extend<A, B>(first: A, second: B): A & B

var jim = extend(new Person(‘Jim’), new ConsoleLogger());
```

intersection type

Intersection Types 101

Intersection types are useful to express *multiple interface inheritance*. They feature in *Scala*, *TypeScript*, *Ceylon*, *Flow*, ...

```
function extend<A, B>(first: A, second: B): A & B
```

intersection type

```
var jim = extend(new Person('Jim'), new ConsoleLogger());
```

Ambiguity

```
var jim = extend(new Person('Jim'), new Person('Alice'));
```

Intersection Types 101

Intersection types are useful to express *multiple interface inheritance*. They feature in *Scala*, *TypeScript*, *Ceylon*, *Flow*, ...

```
function extend<A, B>(first: A, second: B): A & B
```

intersection type

```
var jim = extend(new Person('Jim'), new ConsoleLogger());
```

Ambiguity

```
var jim = extend(new Person('Jim'), new Person('Alice'));
```

Not type-safe

```
var jim = extend(new Person('Jim'), new ConsoleLogger());
```

name: 'Jim'

name: False

Disjoint Intersection Types

- Intersection Types

  `A & B`

```
(A * Int). \(x : A).  x ,, 1
  : ∀(A * Int). A → A & Int
```

Disjoint Intersection Types

- Intersection Types

```
A & B
```

- Merge Operator [Reynolds 1988]

```
e1 ,, e2 :: A & B
```

```
(A * Int). \(x : A).  x ,, 1
  : ∀(A * Int). A → A & Int
```

Disjoint Intersection Types

- Intersection Types

  `A & B`

- Merge Operator [Reynolds 1988]

  `e1 ,, e2 :: A & B`

```
1 ,, True : Int & Bool
```

```
(A * Int). \(x : A).  x ,, 1
  : ∀(A * Int). A → A & Int
```

Disjoint Intersection Types

- Intersection Types

  `A & B`

- Merge Operator [Reynolds 1988]

  `e1 ,, e2 :: A & B`

```
1 ,, True : Int & Bool

(1 ,, True) : Int    ==  1

(A * Int). \(x : A).  x ,, 1
  : ∀(A * Int). A → A & Int
```

Disjoint Intersection Types

- Intersection Types

  `A & B`

- Merge Operator [Reynolds 1988]

  `e1 ,, e2 :: A & B`

```
1 ,, True : Int & Bool

(1 ,, True) : Int    ==  1
(1 ,, True) : Bool   ==  True




(A * Int). \(x : A).  x ,, 1
  : ∀(A * Int). A → A & Int
```

Disjoint Intersection Types

Fi+ [Bi et al. 2019], disjoint polymorphic calculus

- Intersection Types

```
A & B
```

- Merge Operator [Reynolds 1988]

```
e1 ,, e2 :: A & B
```

```
1 ,, True : Int & Bool

(1 ,, True) : Int    ==  1
(1 ,, True) : Bool   ==  True

(A * Int). \(x : A).  x ,, 1
  : ∀(A * Int). A → A & Int
```

Disjoint Intersection Types

Fi+ [Bi et al. 2019], **disjoint polymorphic calculus**

- Intersection Types

  `A & B`

- Merge Operator [Reynolds 1988]

  `e1 ,, e2 :: A & B`

- Disjointness [Oliveira et al. 2016]

  `A * B`

```
1 ,, True : Int & Bool

(1 ,, True) : Int    ==  1
(1 ,, True) : Bool   ==  True

(A * Int). \(x : A).  x ,, 1
  : ∀(A * Int). A → A & Int
```

Disjoint Intersection Types

Fi+ [Bi et al. 2019], **disjoint polymorphic calculus**

- Intersection Types

```
A & B
```

- Merge Operator [Reynolds 1988]

```
e1 ,, e2 :: A & B
```

- Disjointness [Oliveira et al. 2016]

```
A * B
```

```
1 ,, True : Int & Bool
Int * Bool
(1 ,, True) : Int    ==  1
(1 ,, True) : Bool   ==  True


(A * Int). \(x : A).  x ,, 1
  : ∀(A * Int). A → A & Int
```

Disjoint Intersection Types

Fi+ [Bi et al. 2019], **disjoint polymorphic calculus**

- Intersection Types

  `A & B`

- Merge Operator [Reynolds 1988]

  `e1 ,, e2 :: A & B`

- Disjointness [Oliveira et al. 2016]

  `A * B`

```
1 ,, True : Int & Bool
Int * Bool
(1 ,, True) : Int    ==  1
(1 ,, True) : Bool   ==  True

1 ,, 2 : Int & Int    ✗

(A * Int). \(x : A).  x ,, 1
  : ∀(A * Int). A → A & Int
```

Disjoint Intersection Types

Fi+ [Bi et al. 2019], **disjoint polymorphic calculus**

- Intersection Types

```
A & B
```

- Merge Operator [Reynolds 1988]

```
e1 ,, e2 :: A & B
```

- Disjointness [Oliveira et al. 2016]

```
A * B
```

```
1 ,, True : Int & Bool
Int * Bool
(1 ,, True) : Int    ==  1
(1 ,, True) : Bool   ==  True

1 ,, 2 : Int & Int   ✗
Int * Int ✗


(A * Int). \(x : A).  x ,, 1
  : ∀(A * Int). A → A & Int
```

Disjoint Intersection Types

Fi+ [Bi et al. 2019], **disjoint polymorphic calculus**

- Intersection Types

  `A & B`

- Merge Operator [Reynolds 1988]

  `e1 ,, e2 :: A & B`

- Disjointness [Oliveira et al. 2016]

  `A * B`

```
1 ,, True : Int & Bool
Int * Bool
(1 ,, True) : Int    ==  1
(1 ,, True) : Bool   ==  True

1 ,, 2 : Int & Int     ✗
Int * Int ✗
(1 ,, 2)    : Int    ==  1? 2?  ✗

(A * Int). \(x : A).  x ,, 1
  : ∀(A * Int). A → A & Int
```

Disjoint Intersection Types

Fi+ [Bi et al. 2019], **disjoint polymorphic calculus**

- Intersection Types

  `A & B`

- Merge Operator [Reynolds 1988]

  `e1 ,, e2 :: A & B`

- Disjointness [Oliveira et al. 2016]

  `A * B`

```
(A * Int). \(x : A).  x ,, 1
1 ,, True : Int & Bool
Int * Bool
(1 ,, True) : Int    ==  1
(1 ,, True) : Bool   ==  True

1 ,, 2 : Int & Int
Int * Int
(1 ,, 2)    : Int    ==  1? 2?

  : ∀(A * Int). A → A & Int   : ∀(A
* Int). A → A & Int
```

Disjoint Intersection Types

TypeScript

```
function extend<A, B>(first: A, second: B): A & B
```

Disjoint Intersection Types

Ambiguity
Not type-safe

TypeScript

```
function extend<A, B>(first: A, second: B): A & B
```

Disjoint Intersection Types

Ambiguity

Not type-safe

TypeScript

```
function extend<A, B>(first: A, second: B): A & B
```

Low-level and biased implementation

Disjoint Intersection Types

TypeScript

```
function extend<A, B>(first: A, second: B): A & B
```

Fi+

```
let extend A (B * A) (first: A, second: B): A & B
  = first ,, second
```

Disjoint Intersection Types

TypeScript

```
function extend<A, B>(first: A, second: B): A & B
```

Fi+

explicit disjointness

```
let extend A (B * A) (first: A, second: B): A & B
  = first ,, second
```

Disjoint Intersection Types

TypeScript

```
function extend<A, B>(first: A, second: B): A & B
```

Fi+

explicit disjointness

```
let extend A (B * A) (first: A, second: B): A & B
  = first ,, second
```

clear implementation

Row Types

Row types [Wand 1989] , provide an approach to typing extensible records.

Row Types

Row types [Wand 1989] , provide an approach to typing extensible records.

```
{ name = 'jim' } : { name : String }
```

Row Types

Row types [Wand 1989] , provide an approach to typing extensible records.

```
{ name = ’jim’ } : { name : String }

{ age = 8 } : { age : Int }
```

Row Types

Row types [Wand 1989] , provide an approach to typing extensible records.

```
{ name = 'jim' } : { name : String }

{ age = 8 } : { age : Int }

{ name = 'jim', age = 8 } : { name : String, age : Int }
```

Row Types

Row types [Wand 1989] , provide an approach to typing extensible records.

```
{ name = 'jim' } : { name : String }
```

+

```
{ age = 8 } : { age : Int }
```

=

```
{ name = 'jim', age = 8 } : { name : String, age : Int }
```

Row Types

Row types [Wand 1989] , provide an approach to typing extensible records.

```
{ name = 'jim' } : { name : String }

{ name = 'Alice' } : { name : String }
```

Row Types

Row types [Wand 1989] , provide an approach to typing extensible records.

```
{ name = 'jim' } : { name : String }
```

+

```
{ name = 'Alice' } : { name : String }
```

=

```
{ name = 'jim', name = 'Alice' } : { name : String
                                   , name : String }
```

Row Polymorphism

λ|| [Harper and Pierce 1991]

- Record Concatenate

```
{name = 'jim' } || {age = 8}  ==  {name = 'jim', age = 8}

{name = 'jim' } || {name = 'Alice'} ✗
```

Row Polymorphism

λ|| [Harper and Pierce 1991]

- Record Concatenate

```
{name = 'jim' } || {age = 8}  ==  {name = 'jim', age = 8}

{name = 'jim' } || {name = 'Alice'} ❌
```

- Compatibility constraint

```
A # B
```

Row Polymorphism

λ|| [Harper and Pierce 1991]

- Record Concatenate

```
{name = 'jim' } || {age = 8}  ==  {name = 'jim', age = 8}
{name = 'jim' } || {name = 'Alice'} ✗
```

- Compatibility constraint

```
A # B  : A lacks every field contained in B
```

Row Polymorphism

Fi+

```
let extend A (B * A) (first: A, second: B): A & B
  = first ,, second
```

Row Polymorphism

Fi+

```
let extend A (B * A) (first: A, second: B): A & B
  = first ,, second
```

λ||

```
let extend A (B # A) (first: A, second: B): A || B
  = first || second
```

Row Polymorphism

disjointness

Fi+

```
let extend A (B * A) (first: A, second: B): A & B
  = first ,, second
```

compatibility

λ||

```
let extend A (B # A) (first: A, second: B): A || B
  = first || second
```

Row Polymorphism

Fi+

```
let extend A (B * A) (first: A, second: B): A & B
  = first ,, second
```

disjointness

merge operator

λ||

```
let extend A (B # A) (first: A, second: B): A || B
  = first || second
```

compatibility

record concatenation

Row Polymorphism

Fi+

type variables · disjointness

```
let extend A (B * A) (first: A, second: B): A & B
  = first ,, second
```

merge operator

λ||

record variables · compatibility

```
let extend A (B # A) (first: A, second: B): A || B
  = first || second
```

record concatenation

Row Polymorphism through Disjoint Polymorphism

- ✓ Our encoding of λ|| into Fi+ is based on the similarities between the two calculi

Row Polymorphism through Disjoint Polymorphism

- ✓ Our encoding of λ|| into Fi+ is based on the similarities between the two calculi
- ✗ Straightforward elaboration does not work for all programs

Row Polymorphism through Disjoint Polymorphism

- ✓ Our encoding of λ|| into Fi+ is based on the similarities between the two calculi
- ✗ Straightforward elaboration does not work for all programs

```
λ||  Λ(A # { l : Bool }). \(x : A). \(y : { l : Int }). x || y
```

Row Polymorphism through Disjoint Polymorphism

- ✓ Our encoding of λ|| into Fi+ is based on the similarities between the two calculi
- ✗ Straightforward elaboration does not work for all programs

λ|| `Λ(A # { l : Bool }). \(x : A). \(y : { l : Int }). x || y`

Fi+ `Λ(A * { l : Bool }). \(x : A). \(y : { l : Int }). x ,, y`

Row Polymorphism through Disjoint Polymorphism

✓ Our encoding of λ|| into Fi+ is based on the similarities between the two calculi

✗ Straightforward elaboration does not work for all programs

λ||
```
Λ(A # { l : Bool }). \(x : A). \(y : { l : Int }). x || y
// A lacks field l, i.e., { l : A’ } for any A’
```

Fi+
```
Λ(A * { l : Bool }). \(x : A). \(y : { l : Int }). x ,, y
```

Row Polymorphism through Disjoint Polymorphism

✓ Our encoding of λ|| into Fi+ is based on the similarities between the two calculi

✗ Straightforward elaboration does not work for all programs

λ||
```
Λ(A # { l : Bool }). \(x : A). \(y : { l : Int }). x || y
// A lacks field l, i.e., { l : A' } for any A'
// accepted
```

Fi+
```
Λ(A * { l : Bool }). \(x : A). \(y : { l : Int }). x ,, y
```

Row Polymorphism through Disjoint Polymorphism

✓ Our encoding of λ|| into Fi+ is based on the similarities between the two calculi

✗ Straightforward elaboration does not work for all programs

λ||
```
Λ(A # { l : Bool }). \(x : A). \(y : { l : Int }). x || y
// A lacks field l, i.e., { l : A' } for any A'
// accepted
```

Fi+
```
Λ(A * { l : Bool }). \(x : A). \(y : { l : Int }). x ,, y
// A * { l : Bool }
```

Row Polymorphism through Disjoint Polymorphism

✓ Our encoding of λ|| into Fi+ is based on the similarities between the two calculi

✗ Straightforward elaboration does not work for all programs

λ||
```
Λ(A # { l : Bool }). \(x : A). \(y : { l : Int }). x || y
// A lacks field l, i.e., { l : A’ } for any A’
// accepted
```

Fi+
```
Λ(A * { l : Bool }). \(x : A). \(y : { l : Int }). x ,, y
// A * { l : Bool }
// A can be { l : Int } as { l : Int } * { l : Bool } as Int * Bool
```

Row Polymorphism through Disjoint Polymorphism

✓ Our encoding of λ|| into Fi+ is based on the similarities between the two calculi

✗ Straightforward elaboration does not work for all programs

λ||
```
Λ(A # { l : Bool }). \(x : A). \(y : { l : Int }). x || y
// A lacks field l, i.e., { l : A′ } for any A′
// accepted
```

Fi+
```
Λ(A * { l : Bool }). \(x : A). \(y : { l : Int }). x ,, y
// A * { l : Bool }
// A can be { l : Int } as { l : Int } * { l : Bool } as Int * Bool
// rejected
```

Row Polymorphism through Disjoint Polymorphism

1. A simple yet incomplete encoding from *restricted* λ|| into Fi+

2. A complete encoding

Row Polymorphism through Disjoint Polymorphism

1. A simple yet incomplete encoding from *restricted* λ|| into Fi+

❌
```
Λ (A # { l : Bool }). \(x : A). \(y : { l : Int }). x || y
```

2. A complete encoding

Row Polymorphism through Disjoint Polymorphism

1. A simple yet incomplete encoding from *restricted* λ|| into Fi+

```
❌ Λ(A # { l : Bool }). \(x : A). \(y : { l : Int }). x || y
```

2. A complete encoding

```
Λ(A1 * ({ l : Bool } & { l : ⊥ }))
  (A2 * ({ l : Bool } & { l : ⊥ })).
   \(x : A1). \(y : { l : Int }). x ,, y
```

Row Polymorphism through Disjoint Polymorphism

1. A simple yet incomplete encoding from *restricted* λ|| into Fi+

```
❌ Λ(A # { l : Bool }). \(x : A). \(y : { l : Int }). x || y
```

2. A complete encoding

bottom-elaboration

```
Λ(A1 * ({ l : Bool } & { l : ⊥ }))
  (A2 * ({ l : Bool } & { l : ⊥ })).
   \(x : A1). \(y : { l : Int }). x ,, y
```

Row Polymorphism through Disjoint Polymorphism

1. A simple yet incomplete encoding from *restricted* λ|| into Fi+

❌
```
Λ(A # { l : Bool }). \(x : A). \(y : { l : Int }). x || y
```

2. A complete encoding

bottom-elaboration

A
```
Λ(A1 * ({ l : Bool } & { l : ⊥ }))
  (A2 * ({ l : Bool } & { l : ⊥ })).
   \(x : A1). \(y : { l : Int }). x ,, y
```

Row Polymorphism through Disjoint Polymorphism

1. A simple yet incomplete encoding from *restricted* λ|| into Fi+

❌
```
Λ (A # { l : Bool }). \(x : A). \(y : { l : Int }). x || y
```

2. A complete encoding

bottom-elaboration

```
Λ (A1 * ({ l : Bool } & { l : ⊥ }))
  (A2 * ({ l : Bool } & { l : ⊥ })).
   \(x : A1). \(y : { l : Int }). x ,, y
```

A

bottom-elaboration type variable

Bounded Polymorphism

Bounded quantification [Cardelli and Wegner 1985] is a language feature that integrates *parametric polymorphism* with *subtyping*

```
incr = \(x : { age : Int}).{ orig = x, age = x.age + 1 }
```

Bounded Polymorphism

Bounded quantification [Cardelli and Wegner 1985] is a language feature that integrates *parametric polymorphism* with *subtyping*

```
                          single-field record
incr = \(x : { age : Int}).{ orig = x, age = x.age + 1 }
```

Bounded Polymorphism

Bounded quantification [Cardelli and Wegner 1985] **is a language feature that integrates** *parametric polymorphism* **with** *subtyping*

single-field record

```
incr = \(x : { age : Int}).{ orig = x, age = x.age + 1 }


incr_poly = Λ(A <: { age : Int})
                \(x : A). { orig = x, age = x.age + 1 }
```

Bounded Polymorphism

Bounded quantification [Cardelli and Wegner 1985] **is a language feature that integrates** *parametric polymorphism* **with** *subtyping*

```
                                     single-field record
incr = \(x : { age : Int}).{ orig = x, age = x.age + 1 }

                                     subtype of { age : Int }
incr_poly = Λ(A <: { age : Int})
                   \(x : A). { orig = x, age = x.age + 1 }
```

Bounded Polymorphism through Intersection Types

subtype of { age : Int }

```
incr_poly = Λ(A <: { age : Int}).
              \(x : A). { orig = x, age = x.age + 1 }
```

Bounded Polymorphism through Intersection Types

subtype of { age : Int }

```
incr_poly = Λ(A <: { age : Int}).
              \(x : A). { orig = x, age = x.age + 1 }


incr_intersection = Λ A.
              \(x : A & { age : Int}).
                  { orig = x, age = x.age + 1 }
```

Bounded Polymorphism through Intersection Types

subtype of { age : Int }

```
incr_poly = Λ(A <: { age : Int}).
                \(x : A). { orig = x, age = x.age + 1 }
```

subtype of { age : Int }

```
incr_intersection = Λ A.
                \(x : A & { age : Int}).
                    { orig = x, age = x.age + 1 }
```

Bounded Polymorphism through Intersection Types

Pierce [1991] *informally* discussed an encoding of bounded quantification in terms of intersection types

```
∀(a <: A). B        ≜        ∀a.  B [a ~> a & A]
```

Bounded Polymorphism through Intersection Types

Pierce [1991] *informally* discussed an encoding of bounded quantification in terms of intersection types

```
∀(a <: A). B      ≜      ∀a.  B [a ~> a & A]
```

"This is not, however, an encoding of bounded quantification in a full sense ..."

Bounded Polymorphism through Intersection Types

Pierce [1991] *informally* discussed an encoding of bounded quantification in terms of intersection types

$$\texttt{∀(a <: A). B} \quad \triangleq \quad \texttt{∀a. B [a ~> a & A]}$$

"This is not, however, an encoding of bounded quantification in a full sense ..."

"... the encoding trick works better than might be expected."

Bounded Polymorphism via Disjoint Polymorphism

Clarify precisely the expressiveness of this encoding with a type-theoretic formalization

```
∀(a <: A). B    ≜    ∀a. B [a ~> a & A]
```

Bounded Polymorphism via Disjoint Polymorphism

Clarify precisely the expressiveness of this encoding with a type-theoretic formalization

`∀(a <: A). B` ≜ `∀a. B [a ~> a & A]`

Kernel F<:

[Cardelli and Wegner 1985]

Fi+

[Bi et al. 2019]

Bounded Polymorphism via Disjoint Polymorphism

Clarify precisely the expressiveness of this encoding with a type-theoretic formalization

`∀(a <: A). B` ≜ `∀a. B [a ~> a & A]`

Kernel F<:	Fi+
[Cardelli and Wegner 1985]	[Bi et al. 2019]
undirected	bidirectional
implicit subsumption	explicit subsumption
...	...

More in the paper

- Detailed Elaboration
- Extra expressive power of disjoint polymorphism
- More discussion
  - Variants of row polymorphism
  - Variants of bounded quantification
  - Variants of intersection types

https://github.com/xnning/Row-and-Bounded-via-Disjoint