# Recursive Subtyping for All

*May 28, 2025*

*Cambium team, Inria Paris*

**Litao Zhou**

University of Hong Kong
Programming Language Group
Supervised by Bruno C. d. S. Oliveira

# Outline

# Recursive Types: What and Why?

- Useful for modeling structures that refer to themselves:
  - Algebraic Data Types (e.g., Lists, Trees)
  - Recursive Objects

# Recursive Types: What and Why?

- Useful for modeling structures that refer to themselves:
  - Algebraic Data Types (e.g., Lists, Trees)
  - Recursive Objects

### *Class Example*

```
class Point {
  x : Int, y : Int,
  move : Int → Int → Point
}
```

### *As a Recursive Type*

Can be represented as:

$$\mu\alpha.\,\{\mathsf{x} : \mathsf{Int}, \mathsf{y} : \mathsf{Int}, \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\}$$

# Recursive Types: What and Why?

- Useful for modeling structures that refer to themselves:
  - Algebraic Data Types (e.g., Lists, Trees)
  - Recursive Objects

### *Class Example*

```
class Point {
  x : Int, y : Int,
  move : Int → Int → Point
}
```

### *As a Recursive Type*

Can be represented as:

$$\mu\alpha.\,\{\mathsf{x} : \mathsf{Int}, \mathsf{y} : \mathsf{Int}, \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\}$$

### *Two Main Approaches*

- **Equi-recursive types**: $\mu\alpha.\,A$ and its unfolding are considered equal.
- **Iso-recursive types**: $\mu\alpha.\,A$ and its unfolding $A[\mu\alpha.\,A/\alpha]$ are distinct but isomorphic up to unfold and fold operations. (Focus of this talk)

# Iso-Recursive Types: Explicit Folding/Unfolding

- A recursive type $\mu\alpha.\, A$ is **not equal** to its one-step unfolding $A[\mu\alpha.\, A/\alpha]$.
- They are only isomorphic, requiring explicit conversion operators:
  - $\mathsf{unfold}[\mu\alpha.\, A] : \mu\alpha.\, A \to A[\mu\alpha.\, A/\alpha]$
  - $\mathsf{fold}[\mu\alpha.\, A] : A[\mu\alpha.\, A/\alpha] \to \mu\alpha.\, A$

# Iso-Recursive Types: Explicit Folding/Unfolding

- A recursive type $\mu\alpha.\,A$ is **not equal** to its one-step unfolding $A[\mu\alpha.\,A/\alpha]$.
- They are only isomorphic, requiring explicit conversion operators:
  - $\mathsf{unfold}[\mu\alpha.\,A] : \mu\alpha.\,A \to A[\mu\alpha.\,A/\alpha]$
  - $\mathsf{fold}[\mu\alpha.\,A] : A[\mu\alpha.\,A/\alpha] \to \mu\alpha.\,A$

$$\begin{tikzcd} e_1 : \mu\alpha.\,\mathsf{Int} \to \alpha \arrow[r, bend left, "\mathsf{unfold}[\mu\alpha.\,\mathsf{Int} \to \alpha]\ e_1"] & e_2 : \mathsf{Int} \to (\mu\alpha.\,\mathsf{Int} \to \alpha) \arrow[l, bend left, "\mathsf{fold}[\mu\alpha.\,\mathsf{Int} \to \alpha]\ e_2"] \end{tikzcd}$$

# Iso-Recursive Types: Explicit Folding/Unfolding

- A recursive type $\mu\alpha.\, A$ is **not equal** to its one-step unfolding $A[\mu\alpha.\, A/\alpha]$.
- They are only isomorphic, requiring explicit conversion operators:
  - $\mathsf{unfold}[\mu\alpha.\, A] : \mu\alpha.\, A \to A[\mu\alpha.\, A/\alpha]$
  - $\mathsf{fold}[\mu\alpha.\, A] : A[\mu\alpha.\, A/\alpha] \to \mu\alpha.\, A$

$$
\begin{tikzcd}
e_1 : \mu\alpha.\, \mathsf{Int} \to \alpha \arrow[r, bend left, "\mathsf{unfold}[\mu\alpha.\, \mathsf{Int} \to \alpha]\, e_1"] & e_2 : \mathsf{Int} \to (\mu\alpha.\, \mathsf{Int} \to \alpha) \arrow[l, bend left, "\mathsf{fold}[\mu\alpha.\, \mathsf{Int} \to \alpha]\, e_2"]
\end{tikzcd}
$$

- unfold and fold operators are eliminated in the operational semantics.

$$\mathsf{unfold}[A](\mathsf{fold}[A] v) \hookrightarrow v$$

# Iso-Recursive Subtyping

*It is useful to have subtyping for iso-recursive types*

## *Point Class*

```
class Point {
  x : Int, y : Int,
  move : Int → Int → Point
}
```

# Iso-Recursive Subtyping

*It is useful to have subtyping for iso-recursive types*

*Point Class*

```
class Point {
  x : Int, y : Int,
  move : Int → Int → Point
}
```

*Colored Point Class (extends Point)*

```
class ColorPoint {
  color : String,
  x : Int, y : Int,
  move : Int → Int →
       ColorPoint
}
```

# Iso-Recursive Subtyping

*It is useful to have subtyping for iso-recursive types*

### *Point Class*

```
class Point {
  x : Int, y : Int,
  move : Int → Int → Point
}
```

### *Colored Point Class (extends Point)*

```
class ColorPoint {
  color : String,
  x : Int, y : Int,
  move : Int → Int →
      ColorPoint
}
```

### *As Recursive Types*

`ColorPoint` can be used wherever `Point` is expected.

$$\mu\alpha.\left\{\begin{array}{l}\mathsf{color} : \mathsf{String},\\ \mathsf{x} : \mathsf{Int},\\ \mathsf{y} : \mathsf{Int},\\ \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\end{array}\right\}$$

$$\leqslant \mu\alpha.\left\{\begin{array}{l}\mathsf{x} : \mathsf{Int},\\ \mathsf{y} : \mathsf{Int},\\ \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\end{array}\right\}$$

# Iso-Recursive Subtyping

*It is useful to have subtyping for iso-recursive types*

## *Point Class*

```
class Point {
  x : Int, y : Int,
  move : Int → Int → Point
}
```

## *Colored Point Class (extends Point)*

```
class ColorPoint {
  color : String,
  x : Int, y : Int,
  move : Int → Int →
      ColorPoint
}
```

## *As Recursive Types*

`ColorPoint` can be used wherever `Point` is expected.

$$\mu\alpha.\left\{\begin{array}{l}\mathsf{color} : \mathsf{String},\\ \mathsf{x} : \mathsf{Int},\\ \mathsf{y} : \mathsf{Int},\\ \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\end{array}\right\}$$

$$\leqslant \mu\alpha.\left\{\begin{array}{l}\mathsf{x} : \mathsf{Int},\\ \mathsf{y} : \mathsf{Int},\\ \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\end{array}\right\}$$

## *Subtyping can be tricky*

We need rules to determine when $\mu\alpha.\,A \leqslant \mu\alpha.\,B$.

# Challenges in Iso-Recursive Subtyping: Examples

A desirable property: **Unfolding Lemma** for type soundness

$$\text{If } \mu\alpha.\, A \leqslant \mu\alpha.\, B, \text{ then } A[\mu\alpha.\, A/\alpha] \leqslant B[\mu\alpha.\, B/\alpha].$$

# Challenges in Iso-Recursive Subtyping: Examples

A desirable property: **Unfolding Lemma** for type soundness

$$\text{If } \mu\alpha.\,A \leqslant \mu\alpha.\,B, \text{ then } A[\mu\alpha.\,A/\alpha] \leqslant B[\mu\alpha.\,B/\alpha].$$

- **Positive occurrences** of $\alpha$: Usually straightforward.
  - $\mu\alpha.\,\top \to \alpha \leqslant \mu\alpha.\,\mathsf{Int} \to \alpha$ (✔, assumes $\mathsf{Int} \leqslant \top$ and recursive check)
  - Unfolds to: $\top \to (\mu\alpha.\,\top \to \alpha) \leqslant \mathsf{Int} \to (\mu\alpha.\,\mathsf{Int} \to \alpha)$

# Challenges in Iso-Recursive Subtyping: Examples

A desirable property: **Unfolding Lemma** for type soundness

$$\text{If } \mu\alpha.\, A \leqslant \mu\alpha.\, B, \text{ then } A[\mu\alpha.\, A/\alpha] \leqslant B[\mu\alpha.\, B/\alpha].$$

- **Positive occurrences** of $\alpha$: Usually straightforward.
  - $\mu\alpha.\, \top \to \alpha \leqslant \mu\alpha.\, \mathsf{Int} \to \alpha$ (✔, assumes $\mathsf{Int} \leqslant \top$ and recursive check)
  - Unfolds to: $\top \to (\mu\alpha.\, \top \to \alpha) \leqslant \mathsf{Int} \to (\mu\alpha.\, \mathsf{Int} \to \alpha)$
- **Negative occurrences** of $\alpha$: More complex.
  - $\mu\alpha.\, \alpha \to \mathsf{Int} \not\leqslant \mu\alpha.\, \alpha \to \top$ (✘)
  - Unfolds once to: $(\mu\alpha.\, \alpha \to \mathsf{Int}) \to \mathsf{Int} \leqslant (\mu\alpha.\, \alpha \to \top) \to \top$
  - This requires $(\mu\alpha.\, \alpha \to \top) \leqslant (\mu\alpha.\, \alpha \to \mathsf{Int})$ (problematic flip!)

# Challenges in Iso-Recursive Subtyping: Examples

A desirable property: **Unfolding Lemma** for type soundness

$$\text{If } \mu\alpha.\,A \leqslant \mu\alpha.\,B\text{, then } A[\mu\alpha.\,A/\alpha] \leqslant B[\mu\alpha.\,B/\alpha].$$

- **Positive occurrences** of $\alpha$: Usually straightforward.
  - $\mu\alpha.\,\top \to \alpha \leqslant \mu\alpha.\,\mathsf{Int} \to \alpha$ (✔, assumes $\mathsf{Int} \leqslant \top$ and recursive check)
  - Unfolds to: $\top \to (\mu\alpha.\,\top \to \alpha) \leqslant \mathsf{Int} \to (\mu\alpha.\,\mathsf{Int} \to \alpha)$
- **Negative occurrences** of $\alpha$: More complex.
  - $\mu\alpha.\,\alpha \to \mathsf{Int} \not\leqslant \mu\alpha.\,\alpha \to \top$ (✘)
  - Unfolds once to: $(\mu\alpha.\,\alpha \to \mathsf{Int}) \to \mathsf{Int} \leqslant (\mu\alpha.\,\alpha \to \top) \to \top$
  - This requires $(\mu\alpha.\,\alpha \to \top) \leqslant (\mu\alpha.\,\alpha \to \mathsf{Int})$ (problematic flip!)
- **Negative occurrences with $\top$ / Reflexivity**:
  - $\mu\alpha.\,\alpha \to \alpha \leqslant \mu\alpha.\,\alpha \to \alpha$ (✔, by reflexivity)
  - $\mu\alpha.\,\top \to \alpha \leqslant \mu\alpha.\,\alpha \to \alpha$ (✔, if $\alpha \leqslant \top$ allowed in unfolding)
  - Unfolds to: $\top \to (\mu\alpha.\,\top \to \alpha) \leqslant (\mu\alpha.\,\alpha \to \alpha) \to (\mu\alpha.\,\alpha \to \alpha)$

# Challenges in Iso-Recursive Subtyping: Examples

A desirable property: **Unfolding Lemma** for type soundness

$$\text{If } \mu\alpha.\, A \leqslant \mu\alpha.\, B, \text{ then } A[\mu\alpha.\, A/\alpha] \leqslant B[\mu\alpha.\, B/\alpha].$$

- **Positive occurrences** of $\alpha$: Usually straightforward.
  - $\mu\alpha.\, \top \to \alpha \leqslant \mu\alpha.\, \mathsf{Int} \to \alpha$ (✓, assumes $\mathsf{Int} \leqslant \top$ and recursive check)
  - Unfolds to: $\top \to (\mu\alpha.\, \top \to \alpha) \leqslant \mathsf{Int} \to (\mu\alpha.\, \mathsf{Int} \to \alpha)$
- **Negative occurrences** of $\alpha$: More complex.
  - $\mu\alpha.\, \alpha \to \mathsf{Int} \not\leqslant \mu\alpha.\, \alpha \to \top$ (✗)
  - Unfolds once to: $(\mu\alpha.\, \alpha \to \mathsf{Int}) \to \mathsf{Int} \leqslant (\mu\alpha.\, \alpha \to \top) \to \top$
  - This requires $(\mu\alpha.\, \alpha \to \top) \leqslant (\mu\alpha.\, \alpha \to \mathsf{Int})$ (problematic flip!)
- **Negative occurrences with $\top$ / Reflexivity**:
  - $\mu\alpha.\, \alpha \to \alpha \leqslant \mu\alpha.\, \alpha \to \alpha$ (✓, by reflexivity)
  - $\mu\alpha.\, \top \to \alpha \leqslant \mu\alpha.\, \alpha \to \alpha$ (✓, if $\alpha \leqslant \top$ allowed in unfolding)
  - Unfolds to: $\top \to (\mu\alpha.\, \top \to \alpha) \leqslant (\mu\alpha.\, \alpha \to \alpha) \to (\mu\alpha.\, \alpha \to \alpha)$
- **Nested Recursive Types**: $\mu\beta.\, \top \to (\mu\alpha.\, \alpha \to \beta) \quad ? \quad \mu\beta.\, \mathsf{Int} \to (\mu\alpha.\, \alpha \to \beta)$ (✗)

# Traditional Approach: The Amber Rules

The classical formulation for iso-recursive subtyping[1][2].

**S-Amber**

$$\frac{\Gamma, \alpha \leqslant \beta \vdash A \leqslant B}{\Gamma \vdash \mu\alpha.\, A \leqslant \mu\beta.\, B}$$

**S-Assmp**

$$\frac{\alpha \leqslant \beta \in \Gamma}{\Gamma \vdash \alpha \leqslant \beta}$$

**S-Refl**

$$\frac{}{\Gamma \vdash \mu\alpha.\, A \leqslant \mu\alpha.\, A}$$

[1] Cardelli, 1985, “Amber, Combinators and Functional Programming Languages”.
[2] Amadio and Cardelli, 1993, “Subtyping Recursive Types”.

# Traditional Approach: The Amber Rules

The classical formulation for iso-recursive subtyping[1] [2].

*S-Amber*

$$\frac{\Gamma, \alpha \leqslant \beta \vdash A \leqslant B}{\Gamma \vdash \mu\alpha.\, A \leqslant \mu\beta.\, B}$$

*S-Assmp*

$$\frac{\alpha \leqslant \beta \in \Gamma}{\Gamma \vdash \alpha \leqslant \beta}$$

*S-Refl*

$$\frac{}{\Gamma \vdash \mu\alpha.\, A \leqslant \mu\alpha.\, A}$$

- **Pros:**
  - Simple to state.
  - Handles many cases by adding assumptions $\alpha \leqslant \beta$.

[1] Cardelli, 1985, "Amber, Combinators and Functional Programming Languages".
[2] Amadio and Cardelli, 1993, "Subtyping Recursive Types".

# Traditional Approach: The Amber Rules

The classical formulation for iso-recursive subtyping[1] [2].

*S-Amber*

$$\frac{\Gamma, \alpha \leqslant \beta \vdash A \leqslant B}{\Gamma \vdash \mu\alpha.\, A \leqslant \mu\beta.\, B}$$

*S-Assmp*

$$\frac{\alpha \leqslant \beta \in \Gamma}{\Gamma \vdash \alpha \leqslant \beta}$$

*S-Refl*

$$\frac{}{\Gamma \vdash \mu\alpha.\, A \leqslant \mu\alpha.\, A}$$

- **Pros:**
  - Simple to state.
  - Handles many cases by adding assumptions $\alpha \leqslant \beta$.
- **Cons:**
  - Metatheory (e.g., transitivity) is hard to prove.
  - The special context makes the rules non-modular to extend.

[1] Cardelli, 1985, "Amber, Combinators and Functional Programming Languages".
[2] Amadio and Cardelli, 1993, "Subtyping Recursive Types".

# Traditional Approach: The Amber Rules

The classical formulation for iso-recursive subtyping[1] [2].

*S-Amber*

$$\frac{\Gamma, \alpha \leqslant \beta \vdash A \leqslant B}{\Gamma \vdash \mu\alpha.\, A \leqslant \mu\beta.\, B}$$

*S-Assmp*

$$\frac{\alpha \leqslant \beta \in \Gamma}{\Gamma \vdash \alpha \leqslant \beta}$$

*S-Refl*

$$\frac{}{\Gamma \vdash \mu\alpha.\, A \leqslant \mu\alpha.\, A}$$

- **Pros:**
  - Simple to state.
  - Handles many cases by adding assumptions $\alpha \leqslant \beta$.
- **Cons:**
  - Metatheory (e.g., transitivity) is hard to prove.
  - The special context makes the rules non-modular to extend.

$\Rightarrow$ an alternative, more tractable iso-recursive subtyping formulation.

[1] Cardelli, 1985, “Amber, Combinators and Functional Programming Languages”.
[2] Amadio and Cardelli, 1993, “Subtyping Recursive Types”.

# Alternative: Nominal Unfolding Rules

An approach that addresses Amber rules' issues[3].

- "Unfolding twice" triggers enough comparisons in covariant and contravariant positions.
- Simulates a "double unfolding" using *labeled types* $A^{\gamma}$.
- A fresh label $\gamma$ is used for each recursive comparison.

**S-Nominal**

$$\frac{\Gamma, \alpha \vdash A[\alpha \mapsto A^{\gamma}] \leqslant B[\alpha \mapsto B^{\gamma}] \quad (\gamma \text{ fresh})}{\Gamma \vdash \mu\alpha.\, A \leqslant \mu\alpha.\, B}$$

**S-Label**

$$\frac{\Gamma \vdash A \leqslant B}{\Gamma \vdash A^{\gamma} \leqslant B^{\gamma}}$$

[3] Zhou, Zhao, and Oliveira, 2022, "Revisiting Iso-Recursive Subtyping".

# Alternative: Nominal Unfolding Rules

An approach that addresses Amber rules' issues[3].

- "Unfolding twice" triggers enough comparisons in covariant and contravariant positions.
- Simulates a "double unfolding" using *labeled types* $A^{\gamma}$.
- A fresh label $\gamma$ is used for each recursive comparison.

***S-Nominal***

$$\frac{\Gamma, \alpha \vdash A[\alpha \mapsto A^{\gamma}] \leqslant B[\alpha \mapsto B^{\gamma}] \quad (\gamma \text{ fresh})}{\Gamma \vdash \mu\alpha.\, A \leqslant \mu\alpha.\, B}$$

***S-Label***

$$\frac{\Gamma \vdash A \leqslant B}{\Gamma \vdash A^{\gamma} \leqslant B^{\gamma}}$$

***Benefits***

Type sound, same expressive power as Amber rules, transitivity, simpler metatheory.
($\Rightarrow$ *Foundation for this work* $F^{\mu}_{\leqslant}$).

[3] Zhou, Zhao, and Oliveira, 2022, "Revisiting Iso-Recursive Subtyping".

# Nominal Unfolding Rules: Example

***S-Nominal (Reminder)***

$$\frac{\Gamma, \alpha \vdash A[\alpha \mapsto A^{\gamma}] \leqslant B[\alpha \mapsto B^{\gamma}] \quad (\gamma \text{ fresh})}{\Gamma \vdash \mu\alpha.\, A \leqslant \mu\alpha.\, B}$$

***S-Label (Reminder)***

$$\frac{\Gamma \vdash A \leqslant B}{\Gamma \vdash A^{\gamma} \leqslant B^{\gamma}}$$

***Example:*** $\mu\alpha.\, \alpha \to \mathsf{nat} \not\leqslant \mu\alpha.\, \alpha \to \top$

# Nominal Unfolding Rules: Example

***S-Nominal (Reminder)***

$$\frac{\Gamma, \alpha \vdash A[\alpha \mapsto A^{\gamma}] \leqslant B[\alpha \mapsto B^{\gamma}] \quad (\gamma \text{ fresh})}{\Gamma \vdash \mu\alpha.\, A \leqslant \mu\alpha.\, B}$$

***S-Label (Reminder)***

$$\frac{\Gamma \vdash A \leqslant B}{\Gamma \vdash A^{\gamma} \leqslant B^{\gamma}}$$

***Example:*** $\mu\alpha.\, \alpha \to \mathsf{nat} \not\leqslant \mu\alpha.\, \alpha \to \top$

Applying S-Nominal with fresh $\gamma$, we need to show:

$$\Gamma, \alpha \vdash (\alpha \to \mathsf{nat})^{\gamma} \to \mathsf{nat} \quad \leqslant \quad (\alpha \to \top)^{\gamma} \to \top$$

# Nominal Unfolding Rules: Example

***S-Nominal (Reminder)***

$$\frac{\Gamma, \alpha \vdash A[\alpha \mapsto A^{\gamma}] \leqslant B[\alpha \mapsto B^{\gamma}] \quad (\gamma \text{ fresh})}{\Gamma \vdash \mu\alpha.\, A \leqslant \mu\alpha.\, B}$$

***S-Label (Reminder)***

$$\frac{\Gamma \vdash A \leqslant B}{\Gamma \vdash A^{\gamma} \leqslant B^{\gamma}}$$

***Example:*** $\mu\alpha.\, \alpha \to \mathsf{nat} \not\leqslant \mu\alpha.\, \alpha \to \top$

Applying S-Nominal with fresh $\gamma$, we need to show:

$$\Gamma, \alpha \vdash (\alpha \to \mathsf{nat})^{\gamma} \to \mathsf{nat} \quad \leqslant \quad (\alpha \to \top)^{\gamma} \to \top$$

- Covariant: $\Gamma, \alpha \vdash \mathsf{nat} \leqslant \top$ (✔)

# Nominal Unfolding Rules: Example

***S-Nominal (Reminder)***

$$\frac{\Gamma, \alpha \vdash A[\alpha \mapsto A^{\gamma}] \leqslant B[\alpha \mapsto B^{\gamma}] \quad (\gamma \text{ fresh})}{\Gamma \vdash \mu\alpha.\, A \leqslant \mu\alpha.\, B}$$

***S-Label (Reminder)***

$$\frac{\Gamma \vdash A \leqslant B}{\Gamma \vdash A^{\gamma} \leqslant B^{\gamma}}$$

***Example:*** $\mu\alpha.\, \alpha \to \mathsf{nat} \not\leqslant \mu\alpha.\, \alpha \to \top$

Applying S-Nominal with fresh $\gamma$, we need to show:

$$\Gamma, \alpha \vdash (\alpha \to \mathsf{nat})^{\gamma} \to \mathsf{nat} \quad \leqslant \quad (\alpha \to \top)^{\gamma} \to \top$$

- Covariant: $\Gamma, \alpha \vdash \mathsf{nat} \leqslant \top$ (✔)
- Contravariant: $\Gamma, \alpha \vdash (\alpha \to \top)^{\gamma} \leqslant (\alpha \to \mathsf{nat})^{\gamma}$
  - By S-Label: $\Gamma, \alpha \vdash \alpha \to \top \leqslant \alpha \to \mathsf{nat}$
  - This requires $\Gamma, \alpha \vdash \top \leqslant \mathsf{nat}$ (✘) and $\Gamma, \alpha \vdash \alpha \leqslant \alpha$ (✔).

# Bounded Quantification: Quick Overview

Allows abstraction over types that satisfy a subtyping constraint (a bound).

- Example: $\forall(\alpha \leqslant \mathsf{Number}).\ \alpha \to \alpha$
- A function polymorphic in $\alpha$, where $\alpha$ must be a subtype of Number.

[4] Pierce, "Bounded Quantification Is Undecidable".

# Bounded Quantification: Quick Overview

Allows abstraction over types that satisfy a subtyping constraint (a bound).

- Example: $\forall(\alpha \leqslant \mathsf{Number}).\, \alpha \to \alpha$
- A function polymorphic in $\alpha$, where $\alpha$ must be a subtype of Number.

### *Kernel* $F_{\leqslant}$

- $\forall(\alpha \leqslant A).\, B_1 \leqslant \forall(\alpha \leqslant A).\, B_2$
- Bounds must be identical (or equivalent).
- Decidable subtyping.

### *Full* $F_{\leqslant}$

- $\forall(\alpha \leqslant A_1).\, B_1 \leqslant \forall(\alpha \leqslant A_2).\, B_2$
- Bounds can be contravariant ($A_2 \leqslant A_1$).
- More expressive, but undecidable[4].

[4] Pierce, "Bounded Quantification Is Undecidable".

# Bounded Quantification: Quick Overview

Allows abstraction over types that satisfy a subtyping constraint (a bound).

- Example: $\forall(\alpha \leqslant \mathsf{Number}).\, \alpha \to \alpha$
- A function polymorphic in $\alpha$, where $\alpha$ must be a subtype of Number.

***Kernel*** $F_{\leqslant}$

- $\forall(\alpha \leqslant A).\, B_1 \leqslant \forall(\alpha \leqslant A).\, B_2$
- Bounds must be identical (or equivalent).
- Decidable subtyping.

***Full*** $F_{\leqslant}$

- $\forall(\alpha \leqslant A_1).\, B_1 \leqslant \forall(\alpha \leqslant A_2).\, B_2$
- Bounds can be contravariant ($A_2 \leqslant A_1$).
- More expressive, but undecidable[4].

**$F^{\mu}_{\leqslant}$ supports both variants**: Kernel $F^{\mu}_{\leqslant}$ uses (equivalent bound) kernel $F_{\leqslant}$ subtyping rule; Full $F^{\mu}_{\leqslant}$ uses the full $F_{\leqslant}$ subtyping rule.

[4] Pierce, "Bounded Quantification Is Undecidable".

# Our Calculus: $F^{\mu}_{\leqslant}$

- $F^{\mu}_{\leqslant}$ integrates:
  - Iso-recursive subtyping (via Nominal Unfolding rules)
  - Bounded quantification (extending Kernel $F_{\leqslant}$ with equivalent bounds / Full $F_{\leqslant}$ with contravariant bounds)
- Types =
  - $F_{\leqslant}$ (bounded universal types $\forall(\alpha \leqslant A).\, B$) +
  - $\mu\alpha.\, A$ (recursive types) and $A^{\gamma}$ (labeled types).
- Terms = $F_{\leqslant}$ terms + iso-recursive fold/unfolds

| | | | |
|---|---|---|---|
| Types | $A, B, \ldots$ | $::=$ | $\mathsf{nat} \mid \top \mid A_1 \to A_2 \mid \alpha \mid \mu\alpha.\, A \mid A^{\gamma} \mid \forall(\alpha \leqslant A).\, B$ |
| Expressions | $e$ | $::=$ | $x \mid \mathsf{i} \mid e_1\, e_2 \mid \lambda x : A.\, e \mid e\, A \mid \Lambda(\alpha \leqslant A).\, e \mid \mathsf{unfold}\ [A]\ e \mid \mathsf{fold}\ [A]\ e$ |
| Values | $v$ | $::=$ | $\mathsf{i} \mid \lambda x : A.\, e \mid \Lambda(\alpha \leqslant A).\, e \mid \mathsf{fold}\ [A]\ v$ |

# Our Calculus: $F^{\mu}_{\leqslant}$

- Subtyping rules: Combine $\mathbf{F}_{\leqslant}$ **rules** with **nominal unfolding** for $\mu$-types.
  - The only change: recursive variable in $\mu\alpha.\, A$ is introduced bounded as $\alpha \leqslant \top$, so we have a unified type context.

$$\text{Contexts} \quad \Gamma \quad ::= \quad \cdot \mid \Gamma, \alpha \leqslant A \mid \Gamma, x : A$$

*S-Forall(full)*

$$\frac{\Gamma \vdash A_2 \leqslant A_1 \qquad \Gamma, a \leqslant A_2 \vdash B_1 \leqslant B_2}{\Gamma \vdash \forall(a \leqslant A_1).\, B_1 \leqslant \forall(a \leqslant A_2).\, B_2}$$

*S-Var-trans*

$$\frac{\alpha \leqslant A \in \Gamma \qquad \Gamma \vdash A \leqslant B}{\Gamma \vdash \alpha \leqslant B}$$

*S-Nominal*

$$\frac{\Gamma, \alpha \leqslant \top \vdash A[\alpha \mapsto A^{\gamma}] \leqslant B[\alpha \mapsto B^{\gamma}] \qquad (\gamma \text{ fresh})}{\Gamma \vdash \mu\alpha.\, A \leqslant \mu\alpha.\, B}$$

*S-Label*

$$\frac{\Gamma \vdash A \leqslant B}{\Gamma \vdash A^{\gamma} \leqslant B^{\gamma}}$$

# Metatheory: Main Properties of $F^{\mu}_{\leqslant}$

We prove key properties for both Kernel $F^{\mu}_{\leqslant}$ and Full $F^{\mu}_{\leqslant}$, formalized in Rocq:

- **Type soundness**
  - **Progress**: If $\Gamma \vdash e : A$, then $e$ is either a value or exists a step $e \hookrightarrow e'$.
  - **Preservation**: If $\Gamma \vdash e : A$ and $e \hookrightarrow e'$, then $\Gamma \vdash e' : A$.
  - Key challenge: finding the correct generalization of the unfolding lemma $\mu\alpha.\,A \leqslant \mu\alpha.\,B \Rightarrow A[\mu\alpha.\,A/\alpha] \leqslant B[\mu\alpha.\,B/\alpha]$.

[5] Ghelli, 1993, "Recursive Types Are Not Conservative over F$\leqslant$".

# Metatheory: Main Properties of $F^{\mu}_{\leqslant}$

We prove key properties for both Kernel $F^{\mu}_{\leqslant}$ and Full $F^{\mu}_{\leqslant}$, formalized in Rocq:

- **Type soundness**
  - **Progress**: If $\Gamma \vdash e : A$, then $e$ is either a value or exists a step $e \hookrightarrow e'$.
  - **Preservation**: If $\Gamma \vdash e : A$ and $e \hookrightarrow e'$, then $\Gamma \vdash e' : A$.
  - Key challenge: finding the correct generalization of the unfolding lemma $\mu\alpha.\, A \leqslant \mu\alpha.\, B \Rightarrow A[\mu\alpha.\, A/\alpha] \leqslant B[\mu\alpha.\, B/\alpha]$.
- **Conservativity** over $F_{\leqslant}$
  - If $\Gamma \vdash e : A$ holds in $F_{\leqslant}$, then $\Gamma \vdash e : A$ in $F^{\mu}_{\leqslant}$.
  - If $\Gamma \vdash e : A$ holds in $F^{\mu}_{\leqslant}$ and only contains $F_{\leqslant}$ constructs, then $\Gamma \vdash e : A$ also holds in $F_{\leqslant}$.
  - Not the case in equi-recursive types + $F_{\leqslant}$.[5]

[5] Ghelli, 1993, "Recursive Types Are Not Conservative over F≤".

# Metatheory: Main Properties of $F^{\mu}_{\leqslant}$

We prove key properties for both Kernel $F^{\mu}_{\leqslant}$ and Full $F^{\mu}_{\leqslant}$, formalized in Rocq:

- **Type soundness**
  - **Progress**: If $\Gamma \vdash e : A$, then $e$ is either a value or exists a step $e \hookrightarrow e'$.
  - **Preservation**: If $\Gamma \vdash e : A$ and $e \hookrightarrow e'$, then $\Gamma \vdash e' : A$.
  - Key challenge: finding the correct generalization of the unfolding lemma $\mu\alpha.\, A \leqslant \mu\alpha.\, B \Rightarrow A[\mu\alpha.\, A/\alpha] \leqslant B[\mu\alpha.\, B/\alpha]$.
- **Conservativity** over $F_{\leqslant}$
  - If $\Gamma \vdash e : A$ holds in $F_{\leqslant}$, then $\Gamma \vdash e : A$ in $F^{\mu}_{\leqslant}$.
  - If $\Gamma \vdash e : A$ holds in $F^{\mu}_{\leqslant}$ and only contains $F_{\leqslant}$ constructs, then $\Gamma \vdash e : A$ also holds in $F_{\leqslant}$.
  - Not the case in equi-recursive types + $F_{\leqslant}$.[5]
- **Decidability** of typing & subtyping (for Kernel $F^{\mu}_{\leqslant}$)

[5] Ghelli, 1993, "Recursive Types Are Not Conservative over F≤".

# Comparison with Other Systems

| Calculus | Recursive Types | Bounded Quant. | Decidability (Subtyping) | Transitivity (Subtyping) | Conservativity (over $F_{\leqslant}$) | Modular Extension | Mechanized Proofs |
|---|---|---|---|---|---|---|---|
| Amadio et al. (1993) | Iso | N/A | ✓ | Built-in | N/A | × | × |
| Zhou et al. (2022) | Iso | N/A | ✓ | ✓ | N/A | ✓ | ✓ |
| Ghelli (1993) | Equi | Full | | × | × | × | × |
| Colazzo et al. (2005) | Equi | Kernel | ✓ | ✓ | | × | × |
| Jeffrey (2001) | Equi | Full | ✓$^{\times}$ | ✓ | × | × | × |
| Abadi et al. (1996) | Iso | Full | | Built-in | | | × |
| **Kernel $F^{\mu}_{\leqslant}$ (This work)** | Iso | Kernel | ✓ | ✓ | ✓ | ✓ | ✓ |
| **Full $F^{\mu}_{\leqslant}$ (This work)** | Iso | Full | × | ✓ | ✓ | ✓ | ✓ |

Table: Comparison among related works.

# Application: Object Encodings

Objects can be encoded in typed lambda calculi in various ways[6].

- Recursive types: $\mu\alpha.\{\ldots \text{methods}(\alpha) \ldots\}$ **(OR)**

[6] Bruce, Cardelli, and Pierce, 1999, "Comparing Object Encodings".

# Application: Object Encodings

Objects can be encoded in typed lambda calculi in various ways[6].

- Recursive types: $\mu\alpha.\{\ldots \text{methods}(\alpha) \ldots\}$ **(OR)**
- Existential types: $\exists\beta.\beta \times (\beta \rightarrow \{\ldots \text{methods}(\beta) \ldots\})$ **(OE)**
  - $\beta$ is interpreted as the state of the object, while methods are functions that depend on the state

[6] Bruce, Cardelli, and Pierce, 1999, "Comparing Object Encodings".

# Application: Object Encodings

Objects can be encoded in typed lambda calculi in various ways[6].

- Recursive types: $\mu\alpha.\{\ldots \mathsf{methods}(\alpha) \ldots\}$ **(OR)**
- Existential types: $\exists\beta.\beta \times (\beta \to \{\ldots \mathsf{methods}(\beta) \ldots\})$ **(OE)**
  - $\beta$ is interpreted as the state of the object, while methods are functions that depend on the state
- Recursive + existential types: $\mu\alpha.\exists\beta.\beta \times (\beta \to \{\ldots \mathsf{methods}(\alpha) \ldots\})$ **(ORE)**
- Recursive + bounded existentials: $\mu\alpha.\exists(\beta \leqslant \alpha).\beta \times (\beta \to \{\ldots \mathsf{methods}(\beta) \ldots\})$ **(ORBE)**

[6] Bruce, Cardelli, and Pierce, 1999, “Comparing Object Encodings”.

# Application: Object Encodings

Objects can be encoded in typed lambda calculi in various ways[6].

- Recursive types: $\mu\alpha.\{\ldots \text{methods}(\alpha)\ldots\}$ **(OR)**
- Existential types: $\exists\beta.\beta \times (\beta \to \{\ldots \text{methods}(\beta)\ldots\})$ **(OE)**
  - $\beta$ is interpreted as the state of the object, while methods are functions that depend on the state
- Recursive + existential types: $\mu\alpha.\exists\beta.\beta \times (\beta \to \{\ldots \text{methods}(\alpha)\ldots\})$ **(ORE)**
- Recursive + bounded existentials: $\mu\alpha.\exists(\beta \leqslant \alpha).\beta \times (\beta \to \{\ldots \text{methods}(\beta)\ldots\})$ **(ORBE)**

$$\exists(\alpha \leqslant A).B \triangleq \forall(\beta \leqslant \top).\,(\forall(\alpha \leqslant A).\,B \to \beta) \to \beta$$

[6] Bruce, Cardelli, and Pierce, 1999, “Comparing Object Encodings”.

# Application: Object Encodings

Objects can be encoded in typed lambda calculi in various ways[6].

- Recursive types: $\mu\alpha.\{\ldots \text{methods}(\alpha)\ldots\}$ **(OR)**
- Existential types: $\exists\beta.\beta \times (\beta \to \{\ldots \text{methods}(\beta)\ldots\})$ **(OE)**
  - $\beta$ is interpreted as the state of the object, while methods are functions that depend on the state
- Recursive + existential types: $\mu\alpha.\exists\beta.\beta \times (\beta \to \{\ldots \text{methods}(\alpha)\ldots\})$ **(ORE)**
- Recursive + bounded existentials: $\mu\alpha.\exists(\beta \leqslant \alpha).\beta \times (\beta \to \{\ldots \text{methods}(\beta)\ldots\})$ **(ORBE)**

$$\exists(\alpha \leqslant A).B \triangleq \forall(\beta \leqslant \top).\,(\forall(\alpha \leqslant A).\,B \to \beta) \to \beta$$

Kernel $F^{\mu}_{\leqslant}$ encodes **OR**, **OE**, and **ORE**. Full $F^{\mu}_{\leqslant}$ encodes **ORBE**.

[6] Bruce, Cardelli, and Pierce, 1999, "Comparing Object Encodings".

# Application: Precise Object Typing with Bounded Quantification

## *Point Object Type in (**OR**)*

$$\mathsf{Point} \triangleq \mu\alpha.\,\{\mathsf{x} : \mathsf{Int}, \mathsf{y} : \mathsf{Int}, \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\}$$

# Application: Precise Object Typing with Bounded Quantification

*Point Object Type in (**OR**)*

$$\mathsf{Point} \triangleq \mu\alpha.\,\{\mathsf{x} : \mathsf{Int}, \mathsf{y} : \mathsf{Int}, \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\}$$

Consider a polymorphic 'move1' function for points and their subtypes:

```
function move1 [P <= Point] (p : P) = (unfold [Point] p).move 1 1
```

# Application: Precise Object Typing with Bounded Quantification

### *Point Object Type in (**OR**)*

$$\mathsf{Point} \triangleq \mu\alpha.\,\{\mathsf{x} : \mathsf{Int}, \mathsf{y} : \mathsf{Int}, \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\}$$

Consider a polymorphic 'move1' function for points and their subtypes:

```
function move1 [P <= Point] (p : P) = (unfold [Point] p).move 1 1
```

- Desired precise type: $\forall(\mathsf{P} \leqslant \mathsf{Point}).\,\mathsf{P} \to \mathsf{P}$

# Application: Precise Object Typing with Bounded Quantification

***Point Object Type in (OR)***

$$\mathsf{Point} \triangleq \mu\alpha.\, \{\mathsf{x} : \mathsf{Int}, \mathsf{y} : \mathsf{Int}, \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\}$$

Consider a polymorphic 'move1' function for points and their subtypes:

```
function move1 [P <= Point] (p : P) = (unfold [Point] p).move 1 1
```

- Desired precise type: $\forall(\mathsf{P} \leqslant \mathsf{Point}).\, \mathsf{P} \to \mathsf{P}$
- Simpler systems might only give: $\forall(\mathsf{P} \leqslant \mathsf{Point}).\, \mathsf{P} \to \mathsf{Point}$ (loses precision)

# Application: Precise Object Typing with Bounded Quantification

*Point Object Type in (**OR**)*

$$\mathsf{Point} \triangleq \mu\alpha.\,\{\mathsf{x} : \mathsf{Int}, \mathsf{y} : \mathsf{Int}, \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\}$$

Consider a polymorphic 'move1' function for points and their subtypes:

```
function move1 [P <= Point] (p : P) = (unfold [Point] p).move 1 1
```

- Desired precise type: $\forall(\mathsf{P} \leqslant \mathsf{Point}).\,\mathsf{P} \to \mathsf{P}$
- Simpler systems might only give: $\forall(\mathsf{P} \leqslant \mathsf{Point}).\,\mathsf{P} \to \mathsf{Point}$ (loses precision)

$$\dfrac{\dfrac{\mathsf{P} \leqslant \mathsf{Point}, p : \mathsf{P} \vdash p : \mathsf{Point}}{\mathsf{P} \leqslant \mathsf{Point}, p : \mathsf{P} \vdash \mathsf{unfold}[\mathsf{Point}]\; p : \{\ldots, \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\}[\mathsf{Point}/\alpha]}\;\text{T-Unfold}}{\mathsf{P} \leqslant \mathsf{Point}, p : \mathsf{P} \vdash \mathsf{unfold}[\mathsf{Point}]\; p : \{\ldots, \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \mathsf{Point}\}}$$

# F-bounded Polymorphism: The Challenge and Our Approach

- **F-bounded quantification** is a known technique to achieve self-type polymorphism[7], typically written as $\forall(\alpha \leqslant F[\alpha]).A$.
  - Example: $F_{\mathsf{Point}}(\alpha) \triangleq \{\mathsf{x} : \mathsf{Int}, \mathsf{y} : \mathsf{Int}, \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\}$
  - move1 can have the type $\forall(\mathsf{P} \leqslant F_{\mathsf{Point}}(\mathsf{P})).\mathsf{P} \to \mathsf{P}$

[7] Canning et al., 1989, "F-Bounded Polymorphism for Object-Oriented Programming".
[8] Baldan, Ghelli, and Raffaetà, 1999, "Basic Theory of F-Bounded Quantification".

# F-bounded Polymorphism: The Challenge and Our Approach

- **F-bounded quantification** is a known technique to achieve self-type polymorphism[7], typically written as $\forall(\alpha \leqslant F[\alpha]).A$.
  - Example: $F_{\mathsf{Point}}(\alpha) \triangleq \{\mathsf{x} : \mathsf{Int}, \mathsf{y} : \mathsf{Int}, \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\}$
  - move1 can have the type $\forall(\mathsf{P} \leqslant F_{\mathsf{Point}}(\mathsf{P})).\mathsf{P} \to \mathsf{P}$
- Directly adding F-bounds to $F_{\leqslant}$ significantly complicates the system and its metatheory[8].

[7] Canning et al., 1989, "F-Bounded Polymorphism for Object-Oriented Programming".
[8] Baldan, Ghelli, and Raffaetà, 1999, "Basic Theory of F-Bounded Quantification".

# F-bounded Polymorphism: The Challenge and Our Approach

- **F-bounded quantification** is a known technique to achieve self-type polymorphism[7], typically written as $\forall(\alpha \leqslant F[\alpha]).A$.
  - Example: $F_{\mathsf{Point}}(\alpha) \triangleq \{\mathsf{x} : \mathsf{Int}, \mathsf{y} : \mathsf{Int}, \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\}$
  - move1 can have the type $\forall(\mathsf{P} \leqslant F_{\mathsf{Point}}(\mathsf{P})).\mathsf{P} \to \mathsf{P}$
- Directly adding F-bounds to $F_{\leqslant}$ significantly complicates the system and its metatheory[8].
- **Our Approach in $F^{\mu}_{\leqslant}$**: We achieve a similar expressive power for "positive" F-bounds *without* adding explicit F-bounded universal types.

[7] Canning et al., 1989, "F-Bounded Polymorphism for Object-Oriented Programming".
[8] Baldan, Ghelli, and Raffaetà, 1999, "Basic Theory of F-Bounded Quantification".

# F-bounded Polymorphism: The Challenge and Our Approach

- **F-bounded quantification** is a known technique to achieve self-type polymorphism[7], typically written as $\forall(\alpha \leqslant F[\alpha]).A$.
  - Example: $F_{\mathsf{Point}}(\alpha) \triangleq \{\mathsf{x} : \mathsf{Int}, \mathsf{y} : \mathsf{Int}, \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\}$
  - move1 can have the type $\forall(\mathsf{P} \leqslant F_{\mathsf{Point}}(\mathsf{P})).\mathsf{P} \to \mathsf{P}$
- Directly adding F-bounds to $F_{\leqslant}$ significantly complicates the system and its metatheory[8].
- **Our Approach in $F^{\mu}_{\leqslant}$**: We achieve a similar expressive power for "positive" F-bounds *without* adding explicit F-bounded universal types.

[7] Canning et al., 1989, "F-Bounded Polymorphism for Object-Oriented Programming".
[8] Baldan, Ghelli, and Raffaetà, 1999, "Basic Theory of F-Bounded Quantification".

# Key Design 1: Structural Folding/Unfolding Rules

Inspired by Abadi et al.[9], we use structural rules for typing 'fold/unfold' terms:

***T-SUnfold (Structural Unfold)***

$$\frac{\Gamma \vdash e : A \quad \Gamma \vdash A \leqslant \mu\alpha.\, B}{\Gamma \vdash \mathsf{unfold}[A]e : B[\alpha \mapsto A]}$$

***T-SFold (Structural Fold)***

$$\frac{\Gamma \vdash e : A[\alpha \mapsto B] \quad \Gamma \vdash \mu\alpha.\, A \leqslant B}{\Gamma \vdash \mathsf{fold}[B]e : B}$$

[9] Abadi, Cardelli, and Viswanathan, 1996, "An Interpretation of Objects and Object Types".

# Key Design 1: Structural Folding/Unfolding Rules

Inspired by Abadi et al.[9], we use structural rules for typing 'fold/unfold' terms:

*T-SUnfold (Structural Unfold)*

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash A \leqslant \mu\alpha.\, B}{\Gamma \vdash \mathsf{unfold}[A]e : B[\alpha \mapsto A]}$$

*T-SFold (Structural Fold)*

$$\frac{\Gamma \vdash e : A[\alpha \mapsto B] \qquad \Gamma \vdash \mu\alpha.\, A \leqslant B}{\Gamma \vdash \mathsf{fold}[B]e : B}$$

$$\dfrac{\mathsf{P} \leqslant \mathsf{Point}, p : \mathsf{P} \vdash p : \mathsf{Point} \qquad \mathsf{P} \leqslant \mathsf{Point}}{\dfrac{\mathsf{P} \leqslant \mathsf{Point}, p : \mathsf{P} \vdash \mathsf{unfold}[{\color{red}\mathsf{P}}]\; p : \{\ldots, \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\}{\color{red}[\mathsf{P}/\alpha]}}{\mathsf{P} \leqslant \mathsf{Point}, p : \mathsf{P} \vdash \mathsf{unfold}[{\color{red}\mathsf{P}}]\; p : \{\ldots, \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \mathsf{P}\}}}\;\text{T-SUnfold}$$

[9] Abadi, Cardelli, and Viswanathan, 1996, "An Interpretation of Objects and Object Types".

# Key Design 1: Structural Folding/Unfolding Rules

Inspired by Abadi et al.[9], we use structural rules for typing 'fold/unfold' terms:

***T-SUnfold (Structural Unfold)***

$$\frac{\Gamma \vdash e : A \quad \Gamma \vdash A \leqslant \mu\alpha.\, B}{\Gamma \vdash \mathsf{unfold}[A]e : B[\alpha \mapsto A]}$$

***T-SFold (Structural Fold)***

$$\frac{\Gamma \vdash e : A[\alpha \mapsto B] \quad \Gamma \vdash \mu\alpha.\, A \leqslant B}{\Gamma \vdash \mathsf{fold}[B]e : B}$$

$$\dfrac{\dfrac{\mathsf{P} \leqslant \mathsf{Point}, p : \mathsf{P} \vdash p : \mathsf{Point} \quad \mathsf{P} \leqslant \mathsf{Point}}{\mathsf{P} \leqslant \mathsf{Point}, p : \mathsf{P} \vdash \mathsf{unfold}[\mathsf{P}]\ p : \{\ldots, \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\}[\mathsf{P}/\alpha]}\ \textsf{T-SUnfold}}{\mathsf{P} \leqslant \mathsf{Point}, p : \mathsf{P} \vdash \mathsf{unfold}[\mathsf{P}]\ p : \{\ldots, \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \mathsf{P}\}}$$

`(unfold [Point] p).move` has the type $\mathsf{Int} \to \mathsf{Int} \to \mathsf{P}$ instead of $\mathsf{Int} \to \mathsf{Int} \to \mathsf{Point}$ so `move1` has the type $\forall(\mathsf{P} \leqslant \mathsf{Point}).\, \mathsf{P} \to \mathsf{P}$.

[9] Abadi, Cardelli, and Viswanathan, 1996, "An Interpretation of Objects and Object Types".

# Application 2: Subtyping for Algebraic Data Types (ADTs)

The intentional behavior of ADTs can be encoded in $F^{\mu}_{\leqslant}$ using Scott encodings[10]:

```
data Exp1 = Num Int
          | Add Exp1 Exp1
          | Sub Exp1 Exp1
```

$$\mathsf{Exp}_1 \triangleq \mu\, \mathsf{E}.\, \forall(\mathsf{R} \leqslant \top).\ \left\{ \begin{array}{l} \mathsf{num} : \mathsf{Int} \to \mathsf{R}, \\ \mathsf{add} : \mathsf{E} \to \mathsf{E} \to \mathsf{R}, \\ \mathsf{sub} : \mathsf{E} \to \mathsf{E} \to \mathsf{R} \end{array} \right\} \to \mathsf{R}$$

[10] Scott, "A System of Functional Abstraction".

# Application 2: Subtyping for Algebraic Data Types (ADTs)

The intentional behavior of ADTs can be encoded in $F^{\mu}_{\leqslant}$ using Scott encodings[10]:

```
data Exp1 = Num Int
          | Add Exp1 Exp1
          | Sub Exp1 Exp1
```

$$\mathsf{Exp}_1 \triangleq \mu\, E.\, \forall (R \leqslant \top).\ \left\{ \begin{array}{l} \mathsf{num} : \mathsf{Int} \to R, \\ \mathsf{add} : E \to E \to R, \\ \mathsf{sub} : E \to E \to R \end{array} \right\} \to R$$

Constructors ($\mathsf{Num}_1$, $\mathsf{Add}_1$) and case analysis functions (eval) are definable.

```
function Num₁ (n: Int) = fold [Exp₁] (Λ A. λ e. (e.num n))
function Add₁ (e₁ : Exp₁, e₂ : Exp₁) = fold [Exp₁] (Λ A. λ e. (e.add e₁ e₂))

function eval (e : Exp₁) = (unfold [Exp₁] e) [Int] {
  num = λn. n,
  add = λe₁ e₂. (eval e₁ + eval e₂),
  sub = λe₁ e₂. (eval e₁ - eval e₂)
}
```

[10] Scott, "A System of Functional Abstraction".

# Application 2: Subtyping for Algebraic Data Types (ADTs)

Similarly, we can define $\mathsf{Exp}_2$ extending $\mathsf{Exp}_1$ with a $\mathsf{Neg}$ constructor:

$$\mathsf{Exp}_2 \triangleq \mu\, E.\, \forall (R \leqslant \top).\ \left\{ \begin{array}{l} \mathsf{num} : \mathsf{Int} \to R, \\ \mathsf{add} : E \to E \to R, \\ \mathsf{sub} : E \to E \to R, \\ \mathsf{neg} : E \to R \end{array} \right\} \to R$$

# Application 2: Subtyping for Algebraic Data Types (ADTs)

Similarly, we can define $\mathsf{Exp}_2$ extending $\mathsf{Exp}_1$ with a Neg constructor:

$$\mathsf{Exp}_2 \triangleq \mu\,\mathsf{E}.\,\forall(\mathsf{R} \leqslant \top).\ \left\{\begin{array}{l} \mathsf{num} : \mathsf{Int} \to \mathsf{R}, \\ \mathsf{add} : \mathsf{E} \to \mathsf{E} \to \mathsf{R}, \\ \mathsf{sub} : \mathsf{E} \to \mathsf{E} \to \mathsf{R}, \\ \mathsf{neg} : \mathsf{E} \to \mathsf{R} \end{array}\right\} \to \mathsf{R}$$

In $F^{\mu}_{\leqslant}$, we can show that $\mathsf{Exp}_1 \leqslant \mathsf{Exp}_2$ with structural subtyping.

# Application 2: Subtyping for Algebraic Data Types (ADTs)

Similarly, we can define $\mathsf{Exp}_2$ extending $\mathsf{Exp}_1$ with a Neg constructor:

$$\mathsf{Exp}_2 \triangleq \mu\, E.\, \forall (R \leqslant \top).\ \left\{ \begin{array}{l} \mathsf{num} : \mathsf{Int} \to R, \\ \mathsf{add} : E \to E \to R, \\ \mathsf{sub} : E \to E \to R, \\ \mathsf{neg} : E \to R \end{array} \right\} \to R$$

In $F^{\mu}_{\leqslant}$, we can show that $\mathsf{Exp}_1 \leqslant \mathsf{Exp}_2$ with structural subtyping.

*Benefits of ADT Subtyping* ($\mathsf{Exp}_1 \leqslant \mathsf{Exp}_2$)

- **Code Reuse**: A function processing $\mathsf{Exp}_2$ ($\mathsf{eval}_2 : \mathsf{Exp}_2 \to \mathsf{Int}$) can operate on $\mathsf{Exp}_1$ terms.
- **Polymorphic Constructors**: Avoid code duplication but preserve type precision.

# ADTs: Polymorphic Constructors & Lower Bounds

Instead of writing individual constructors for each ADT:

```
function Add₁ (e₁ : Exp₁, e₂ : Exp₁) = fold [Exp₁] (Λ A. λ e. (e.add e₁ e₂))
function Add₂ (e₁ : Exp₂, e₂ : Exp₂) = fold [Exp₂] (Λ A. λ e. (e.add e₁ e₂))
```

# ADTs: Polymorphic Constructors & Lower Bounds

Instead of writing individual constructors for each ADT:

```
function Add₁ (e₁ : Exp₁, e₂ : Exp₁) = fold [Exp₁] (Λ A. λ e. (e.add e₁ e₂))
function Add₂ (e₁ : Exp₂, e₂ : Exp₂) = fold [Exp₂] (Λ A. λ e. (e.add e₁ e₂))
```

We can define a single constructor for both $\text{Exp}_1$ and $\text{Exp}_2$:

```
function Add∀ [E ⩾ Exp₁] (e₁ : E, e₂ : E) = fold [E] (Λ A. λ e. (e.add e₁ e₂))
```

# ADTs: Polymorphic Constructors & Lower Bounds

Instead of writing individual constructors for each ADT:

```
function Add₁ (e₁ : Exp₁, e₂ : Exp₁) = fold [Exp₁] (Λ A. λ e. (e.add e₁ e₂))
function Add₂ (e₁ : Exp₂, e₂ : Exp₂) = fold [Exp₂] (Λ A. λ e. (e.add e₁ e₂))
```

We can define a single constructor for both $\mathsf{Exp}_1$ and $\mathsf{Exp}_2$:

```
function Add∀ [E ⩾ Exp₁] (e₁ : E, e₂ : E) = fold [E] (Λ A. λ e. (e.add e₁ e₂))
```

Using $\mathsf{Add}_\forall : \forall(\mathsf{E} \geqslant \mathsf{Exp}_1).\, \mathsf{E} \to \mathsf{E} \to \mathsf{E}$, we can construct terms of type $\mathsf{Exp}_1$, $\mathsf{Exp}_2$, etc., if they are all supertypes of some base expression type.

Required $F^{\mu}_{\leqslant}$ ingredients:

- **Structural folding rules** for fold expressions
- **Lower bounded quantification**, also studied in this work.

# Key Design: Lower Bounded Quantification for ADTs

- Extends Kernel $F^{\mu}_{\leqslant}$ with:
  - Lower bounded quantification: $\forall(\alpha \geqslant A).\, B$ (type $\alpha$ is a supertype of $A$).
  - Note, a variable can only be either upper or lower bounded, not both.
  - Bottom type ($\perp$), single field record types ($\{\mathsf{l} : A\}$), and intersection types for records (&).

$$\{x : \mathsf{Int}, y : \mathsf{Int}\} \triangleq \{x : \mathsf{Int}\} \,\&\, \{y : \mathsf{Int}\}$$

# Key Design: Lower Bounded Quantification for ADTs

- Extends Kernel $F^{\mu}_{\leqslant}$ with:
  - Lower bounded quantification: $\forall(\alpha \geqslant A).\, B$ (type $\alpha$ is a supertype of $A$).
  - Note, a variable can only be either upper or lower bounded, not both.
  - Bottom type ($\bot$), single field record types ($\{\mathsf{l} : A\}$), and intersection types for records (&).

$$\{x : \mathsf{Int}, y : \mathsf{Int}\} \triangleq \{x : \mathsf{Int}\} \,\&\, \{y : \mathsf{Int}\}$$

- $F^{\mu\wedge}_{\leqslant\geqslant}$ successfully types polymorphic ADT constructors like $\mathsf{Add}_{\forall}$.
- Same metatheory results (type soundness, conservativity, decidability) proved for $F^{\mu\wedge}_{\leqslant\geqslant}$.

# Conclusion

- Successfully integrated iso-recursive types (nominal unfolding) with kernel and full $F_{\leqslant}$.
  - **Kernel** $F^{\mu}_{\leqslant}$ / $F^{\mu\wedge}_{\leqslant\geqslant}$: Transitive, decidable subtyping, conservative over kernel $F_{\leqslant}$, type sound.
  - **Full** $F^{\mu}_{\leqslant}$: Transitive, undecidable (as expected) subtyping, conservative over full $F_{\leqslant}$, type sound.
- Applications:
  - Foundational calculus for well-known object encodings
  - Encoding positive forms of F-bounded quantification
  - Enable subtyping for algebraic data types (ADTs) and polymorphic ADT constructors

# Other work on iso-recursive types

**Revisiting Iso-Recursive Subtyping**
(TOPLAS, Y. Zhou et al.)
Nominal Unfolding for $\mu\alpha.\,A$

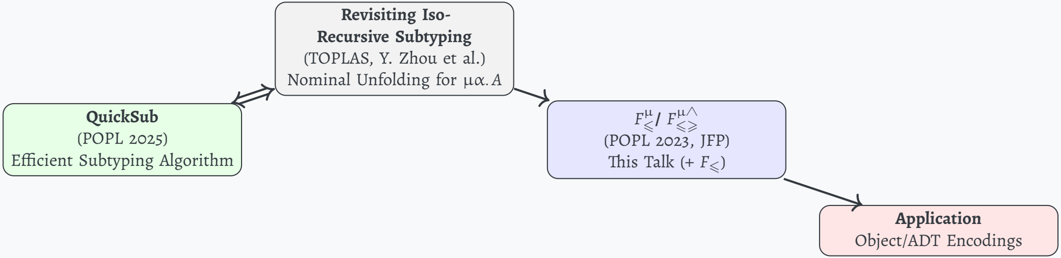
# Other work on iso-recursive types

**Revisiting Iso-Recursive Subtyping**
(TOPLAS, Y. Zhou et al.)
Nominal Unfolding for $\mu\alpha. A$

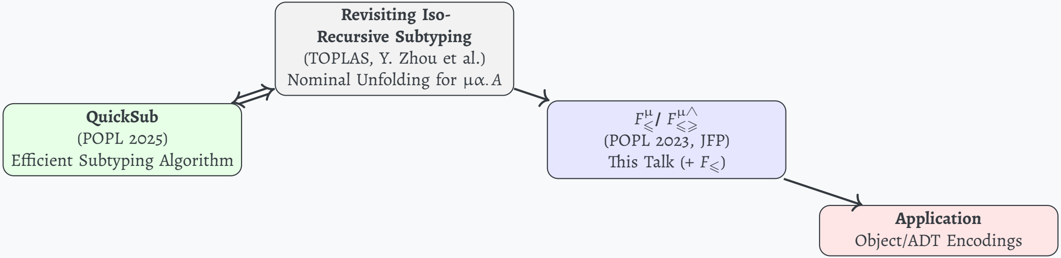
$F^{\mu}_{\leqslant}$ / $F^{\mu\wedge}_{\leqslant\geqslant}$
(POPL 2023, JFP)
This Talk (+ $F_{\leqslant}$)

**Application**
Object/ADT Encodings

# Other work on iso-recursive types



# Other work on iso-recursive types

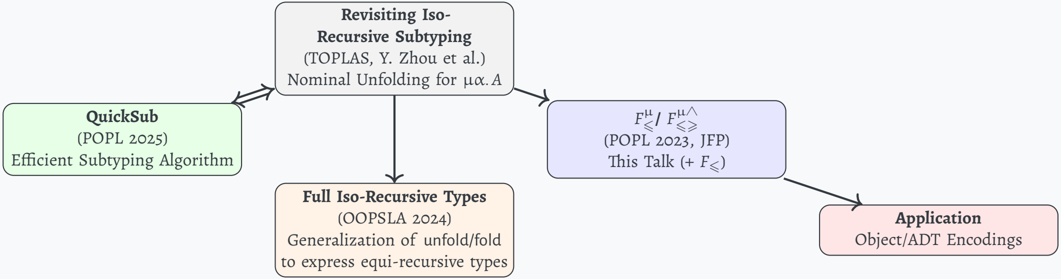



*QuickSub: Efficient Iso-Recursive Subtyping*

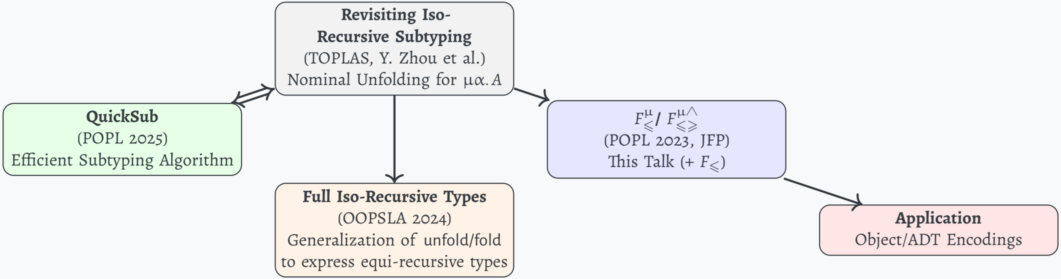
$$\Psi \vdash_{\oplus} A \lessapprox B$$

- Ideas: Tracking polarity $\oplus$, distinguish strict subtypes with equivalence on the fly ($\lessapprox ::= < \mid \approx_S$).
- Equivalent to Amber rules, linear complexity for practical subtypes

# Other work on iso-recursive types



# Other work on iso-recursive types

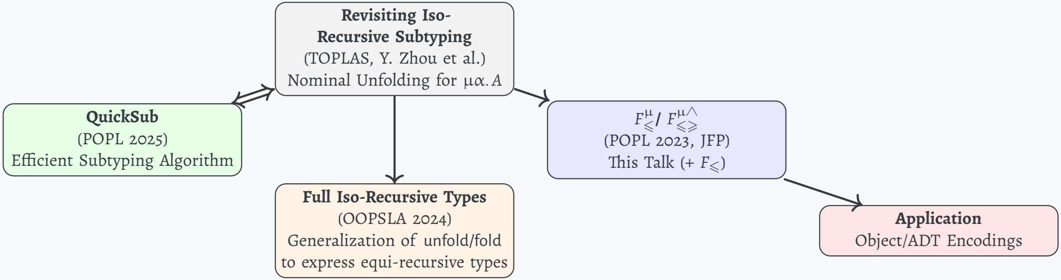



## *Full Iso-Recursive Types*

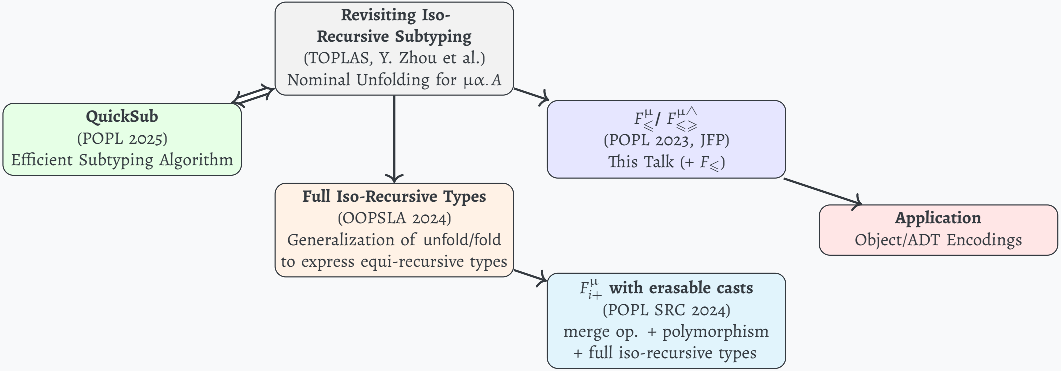
$$e ::= \ldots \mid \mathsf{cast}[c]e \qquad c ::= \mathsf{unfold}_{\mu\alpha.A} \mid \mathsf{fold}_{\mu\alpha.A} \mid c_1 \to c_2 \mid \mathsf{fix}\ \iota.c \mid \ldots$$

- Idea: Replace the standard unfold/fold operator with cast's, to enable deep isomorphic transformations on recursive types.
- Same expressiveness of equi-recursive types (easy proof!), casts are runtime irrelevant

# Other work on iso-recursive types



# Other work on iso-recursive types



# Other work on iso-recursive types

**Revisiting Iso-Recursive Subtyping**
(TOPLAS, Y. Zhou et al.)
Nominal Unfolding for $\mu\alpha.A$

**QuickSub**
(POPL 2025)
Efficient Subtyping Algorithm

$F^{\mu}_{\leqslant}$ / $F^{\mu\wedge}_{\leqslant\geqslant}$
(POPL 2023, JFP)
This Talk (+ $F_{\leqslant}$)

**Full Iso-Recursive Types**
(OOPSLA 2024)
Generalization of unfold/fold
to express equi-recursive types

**Application**
Object/ADT Encodings

$F^{\mu}_{i+}$ **with erasable casts**
(POPL SRC 2024)
merge op. + polymorphism
+ full iso-recursive types

Merge operators $e_1 ,, e_2$ create inhabitants of intersection types:

$$\frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : B}{\Gamma \vdash e_1 ,, e_2 : A \& B}$$

With cast operators: compositional terms like
$\mathsf{cast}[\ldots \to \mathsf{fold}_{\mu a.A_1 \& A_2}](e_1 ,, e_2)$ can be statically typed.

# Other work on iso-recursive types

**Revisiting Iso-Recursive Subtyping**
(TOPLAS, Y. Zhou et al.)
Nominal Unfolding for $\mu\alpha. A$

**QuickSub**
(POPL 2025)
Efficient Subtyping Algorithm

$F^{\mu}_{\leqslant}$ / $F^{\mu\wedge}_{\leqslant\geqslant}$
(POPL 2023, JFP)
This Talk (+ $F_{\leqslant}$)

**Full Iso-Recursive Types**
(OOPSLA 2024)
Generalization of unfold/fold to express equi-recursive types

**Application**
Object/ADT Encodings

$F^{\mu}_{i+}$ **with erasable casts**
(POPL SRC 2024)
merge op. + polymorphism + full iso-recursive types

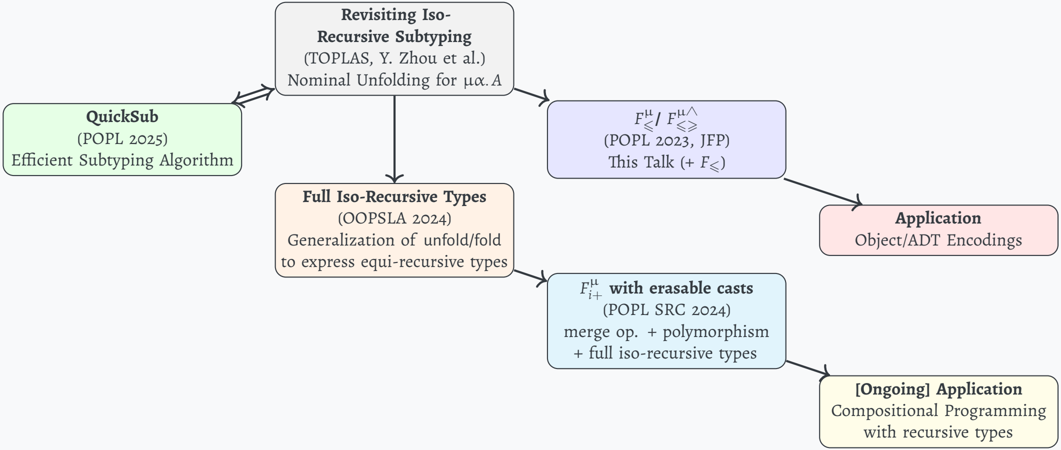
**[Ongoing] Application**
Compositional Programming with recursive types

Merge operators $e_1 ,, e_2$ create inhabitants of intersection types:

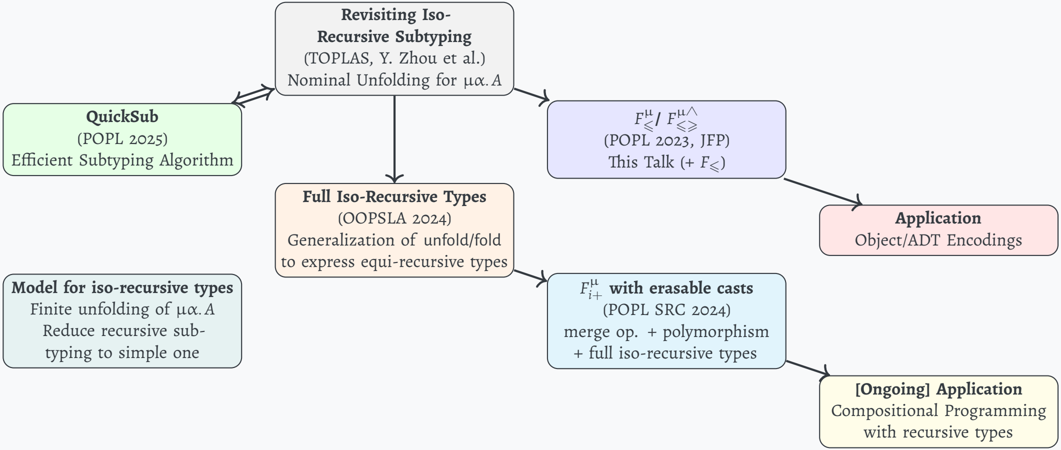
$$\frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : B}{\Gamma \vdash e_1 ,, e_2 : A \& B}$$

With cast operators: compositional terms like $\mathsf{cast}[\ldots \to \mathsf{fold}_{\mu a. A_1 \& A_2}](e_1 ,, e_2)$ can be statically typed.

# Other work on iso-recursive types



# Other work on iso-recursive types




Idea: First translate recursive types into their (nominal) unfolding and then use the translation as a model for subtyping. $A \leqslant B \triangleq \mathsf{Trans}(A) <: \mathsf{Trans}(B)$.

# Other work on iso-recursive types

**Revisiting Iso-Recursive Subtyping**
(TOPLAS, Y. Zhou et al.)
Nominal Unfolding for $\mu\alpha.A$

**QuickSub**
(POPL 2025)
Efficient Subtyping Algorithm

**Model for iso-recursive types**
Finite unfolding of $\mu\alpha.A$
Reduce recursive subtyping to simple one

**Full Iso-Recursive Types**
(OOPSLA 2024)
Generalization of unfold/fold to express equi-recursive types

$F_{\leqslant}^{\mu}$ / $F_{\leqslant>}^{\mu\wedge}$
(POPL 2023, JFP)
This Talk (+ $F_{\leqslant}$)

**Application**
Object/ADT Encodings

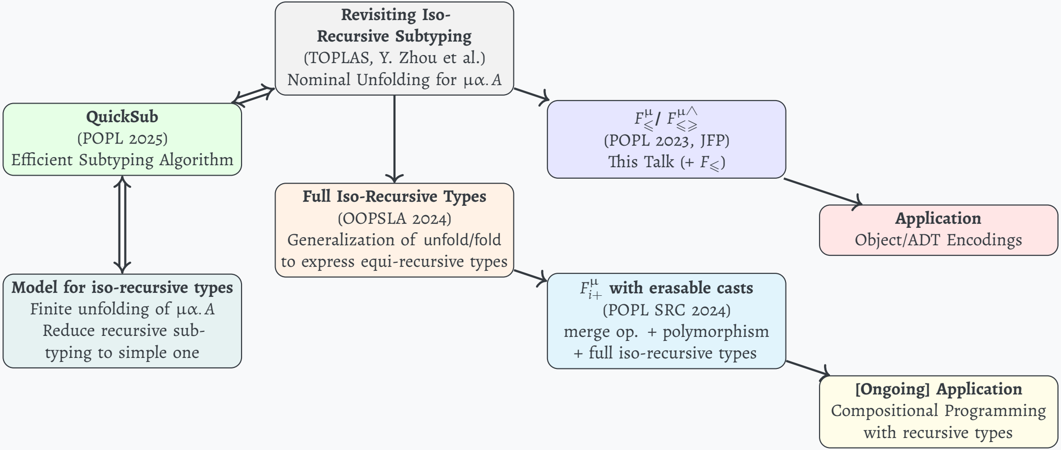
$F_{i+}^{\mu}$ **with erasable casts**
(POPL SRC 2024)
merge op. + polymorphism + full iso-recursive types

**[Ongoing] Application**
Compositional Programming with recursive types

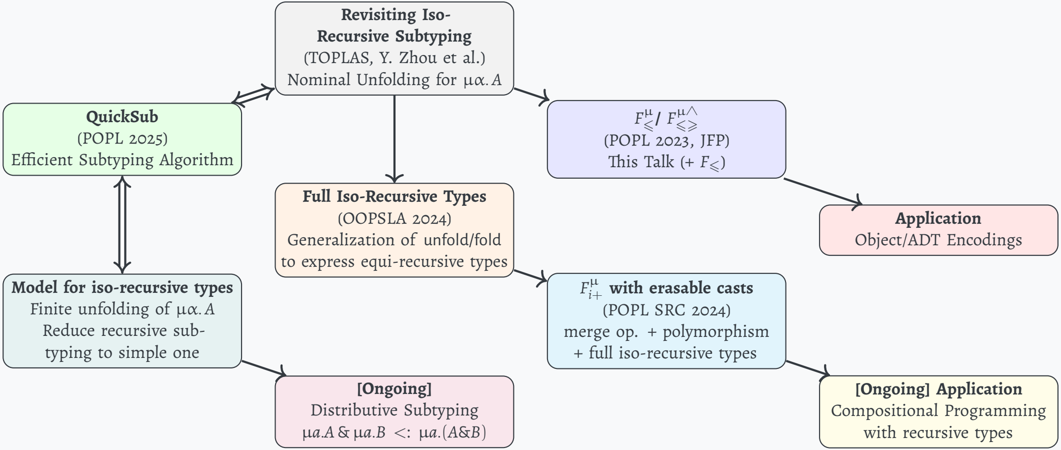
e.g.: $\mathsf{Trans}(\mu\alpha.\,\alpha \to \mathsf{Int}) = [\alpha.\{\alpha \to \mathsf{Int}\}^{\alpha} \to \mathsf{Int}]$

Idea: First translate recursive types into their (nominal) unfolding and then use the translation as a model for subtyping. $A \leqslant B \triangleq \mathsf{Trans}(A) <: \mathsf{Trans}(B)$.

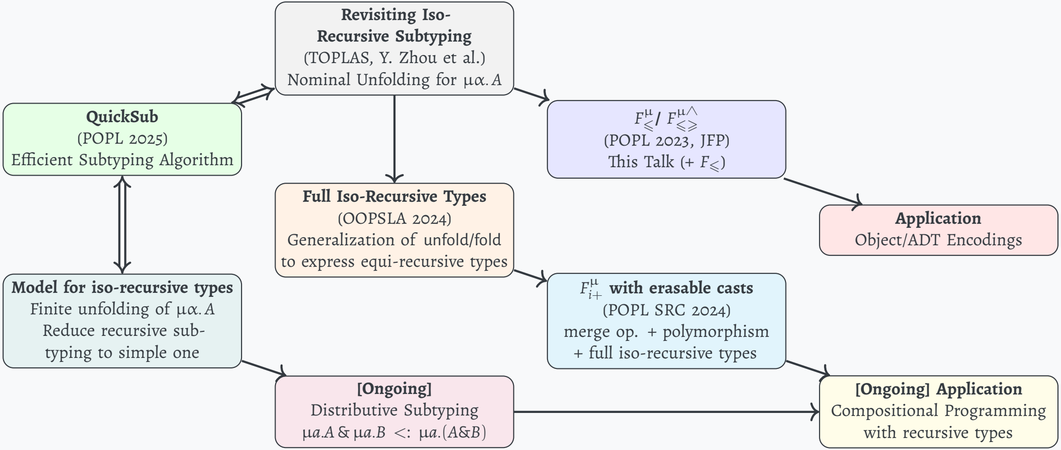
# Other work on iso-recursive types



# Other work on iso-recursive types



# Thank you!


**Revisiting Iso-Recursive Subtyping**
(TOPLAS, Y. Zhou et al.)
Nominal Unfolding for $\mu\alpha. A$
**QuickSub**
(POPL 2025)
Efficient Subtyping Algorithm
$F^{\mu}_{\leqslant}$ / $F^{\mu\wedge}_{\leqslant >}$
(POPL 2023, JFP)
This Talk (+ $F_{\leqslant}$)
**Full Iso-Recursive Types**
(OOPSLA 2024)
Generalization of unfold/fold to express equi-recursive types
**Application**
Object/ADT Encodings
**Model for iso-recursive types**
Finite unfolding of $\mu\alpha. A$
Reduce recursive sub-typing to simple one
$F^{\mu}_{i+}$ **with erasable casts**
(POPL SRC 2024)
merge op. + polymorphism + full iso-recursive types
**[Ongoing]**
Distributive Subtyping
$\mu a.A \,\&\, \mu a.B <: \mu a.(A\&B)$
**[Ongoing] Application**
Compositional Programming with recursive types

# References I


- Abadi, Martín, Luca Cardelli, and Ramesh Viswanathan. "An Interpretation of Objects and Object Types". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. New York, NY, USA: Association for Computing Machinery, Jan. 1996, pp. 396–409. ISBN: 978-0-89791-769-8. DOI: 10.1145/237721.237809. (Visited on 02/08/2024).
- Amadio, Roberto M and Luca Cardelli. "Subtyping Recursive Types". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15.4 (1993), pp. 575–631. DOI: 10.1145/155183.155231.
- Baldan, Paolo, Giorgio Ghelli, and Alessandra Raffaetà. "Basic Theory of F-Bounded Quantification". In: *Information and Computation* 153.2 (Sept. 1999), pp. 173–237. ISSN: 08905401. DOI: 10.1006/inco.1999.2802. (Visited on 07/27/2024).
- Bruce, Kim B., Luca Cardelli, and Benjamin C. Pierce. "Comparing Object Encodings". In: *Information and Computation* 155.1 (Nov. 1999), pp. 108–133. ISSN: 0890-5401. DOI: 10.1006/inco.1999.2829. (Visited on 02/09/2024).

# References II


- Canning, Peter et al. "F-Bounded Polymorphism for Object-Oriented Programming". In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. Fpca 1989. Imperial College, London, United Kingdom, 1989. DOI: 10.1145/99370.99392.
- Cardelli, Luca. "Amber, Combinators and Functional Programming Languages". In: *Proc. of the 13th Summer School of the LITP, Le Val D'Ajol, Vosges (France)*. 1985.
- Colazzo, Dario and Giorgio Ghelli. "Subtyping Recursion and Parametric Polymorphism in Kernel Fun". In: *Information and Computation* 198.2 (2005), pp. 71–147. DOI: 10.1016/j.ic.2004.11.003.
- Ghelli, Giorgio. "Recursive Types Are Not Conservative over $F_{\leqslant}$". In: *International Conference on Typed Lambda Calculi and Applications*. Springer, 1993, pp. 146–162. DOI: 10.1007/BFb0037104.
- Jeffrey, Alan. "A Symbolic Labelled Transition System for Coinductive Subtyping of $F_{\mu\leqslant}$ Types". In: *2001 IEEE Conference on Logic and Computer Science (LICS 2001)*. Vol. 323. 2001.

# References III


- Pierce, Benjamin C. "Bounded Quantification Is Undecidable". In: *Information and Computation* 112.1 (1994), pp. 131–165. DOI: 10.1006/inco.1994.1055.
- Scott, Dana. "A System of Functional Abstraction". Lectures delivered at University of California, Berkeley, California, USA, 1962/63. 1962.
- Zhou, Yaoda, Jinxu Zhao, and Bruno C. d. S. Oliveira. "Revisiting Iso-Recursive Subtyping". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 44.4 (2022), pp. 1–54. DOI: 10.1145/3549537.