QuickSub: Efficient Iso-Recursive Subtyping

January 24, 2025

Litao Zhou, Bruno C. d. S. Oliveira

University of Hong Kong

Recursive Types

Recursive types are useful for defining recursive data structures like lists, trees, and objects.

Types $A ::= \mathrm{Int} \mid \top \mid A_1 \to A_2 \mid \mu\alpha.A \mid \alpha \mid \ldots$

Recursive Types

Recursive types are useful for defining recursive data structures like lists, trees, and objects.

Types $A ::= \mathrm{Int} \mid \top \mid A_1 \to A_2 \mid \mu\alpha. A \mid \alpha \mid \ldots$

```
class A {
  foo(x: Int) : A
  bar(x: A) : Int
   ...
}
```

represented as $\mu\alpha. \left\{ \begin{array}{l} \mathsf{foo} : \mathrm{Int} \to \alpha \\ \mathsf{bar} : \alpha \to \mathrm{Int} \\ \ldots \end{array} \right\}$

Recursive Types

Recursive types are useful for defining recursive data structures like lists, trees, and objects.

Types $\quad A ::= \mathsf{Int} \mid \top \mid A_1 \rightarrow A_2 \mid \mu\alpha.A \mid \alpha \mid \ldots$

```
class A {
  foo(x: Int) : A
  bar(x: A) : Int
   ...
}
```

represented as $\quad \mu\alpha. \left\{ \begin{array}{l} \mathsf{foo} : \mathsf{Int} \rightarrow \alpha \\ \mathsf{bar} : \alpha \rightarrow \mathsf{Int} \\ \ldots \end{array} \right\}$

Two main approaches: iso-recursive and equi-recursive types.

Iso-Recursive Types

Unlike equi-recursive types, recursive types and their unfoldings are not equal, and require explicit fold and unfold operations.

✗ $\mu\alpha.\,\mathsf{Int} \to \alpha = \mathsf{Int} \to (\mu\alpha.\,\mathsf{Int} \to \alpha)$

Iso-Recursive Types

Unlike equi-recursive types, recursive types and their unfoldings are not equal, and require explicit fold and unfold operations.

✗ $\mu\alpha.\,\mathsf{Int} \to \alpha = \mathsf{Int} \to (\mu\alpha.\,\mathsf{Int} \to \alpha)$

$\mathsf{unfold}\,[\mu\alpha.\,\mathsf{Int} \to \alpha]\; e_1$

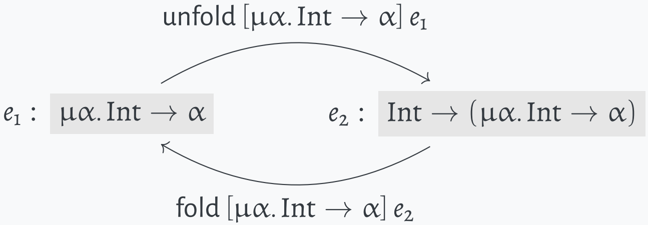
$e_1 : \mu\alpha.\,\mathsf{Int} \to \alpha$ $\qquad$ $e_2 : \mathsf{Int} \to (\mu\alpha.\,\mathsf{Int} \to \alpha)$

$\mathsf{fold}\,[\mu\alpha.\,\mathsf{Int} \to \alpha]\; e_2$

Iso-Recursive Types

Unlike equi-recursive types, recursive types and their unfoldings are not equal, and require explicit fold and unfold operations.

✗ $\mu\alpha.\,\mathrm{Int} \to \alpha = \mathrm{Int} \to (\mu\alpha.\,\mathrm{Int} \to \alpha)$



Iso-Recursive Subtyping

✓ $\mu\alpha.\,\top \to \alpha \leqslant \mu\alpha.\,\mathrm{Int} \to \alpha$

Recursive Subtyping

Iso-Recursive Subtyping (vs. Equi-Recursive Subtyping)

- ✓ Simpler metatheory
  - No coinductive reasoning is needed
- ✓ Easier to scale to more features[a,b,c]
- ✓ More efficient meta operations[d] (e.g. equivalence checking)

---

[a] Dreyer et al., *Toward a Practical Type Theory for Recursive Modules*.
[b] Chugh, "IsoLATE: A type system for self-recursion".
[c] L. Zhou et al., "Recursive Subtyping for All".
[d] Rossberg, "Mutually Iso-Recursive Subtyping".

Recursive Subtyping

Iso-Recursive Subtyping (vs. Equi-Recursive Subtyping)

- ✓ Simpler metatheory
  - No coinductive reasoning is needed
- ✓ Easier to scale to more features[a,b,c]
- ✓ More efficient meta operations[d] (e.g. equivalence checking)
- ☹ Lack of an efficient iso-recursive subtyping algorithm

---

[a] Dreyer et al., *Toward a Practical Type Theory for Recursive Modules*.
[b] Chugh, "IsoLATE: A type system for self-recursion".
[c] L. Zhou et al., "Recursive Subtyping for All".
[d] Rossberg, "Mutually Iso-Recursive Subtyping".

Recursive Subtyping

Iso-Recursive Subtyping (vs. Equi-Recursive Subtyping)

✓ Simpler metatheory
- No coinductive reasoning is needed

✓ Easier to scale to more features[a,b,c]

✓ More efficient meta operations[d] (e.g. equivalence checking)

☹ Lack of an efficient iso-recursive subtyping algorithm ⇒ **QuickSub**

---

[a] Dreyer et al., *Toward a Practical Type Theory for Recursive Modules*.
[b] Chugh, "IsoLATE: A type system for self-recursion".
[c] L. Zhou et al., "Recursive Subtyping for All".
[d] Rossberg, "Mutually Iso-Recursive Subtyping".

Considerations for subtyping iso-recursive types

Unfolding Lemma (expected)

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

Considerations for subtyping iso-recursive types

Unfolding Lemma (expected)

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

- Positive subtyping is easy to check by comparing the type body
  - $\mu\alpha. \top \to \alpha \leqslant \mu\alpha. \mathrm{Int} \to \alpha$

Considerations for subtyping iso-recursive types

Unfolding Lemma (expected)

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

- Positive subtyping is easy to check by comparing the type body
  - $\mu\alpha. \top \to \alpha \leqslant \mu\alpha. \mathrm{Int} \to \alpha$
    - $\top \to (\mu\alpha. \top \to \alpha) \leqslant \mathrm{Int} \to (\mu\alpha. \mathrm{Int} \to \alpha)$ ✓
    - $\top \to (\top \to (\mu\alpha. \top \to \alpha)) \leqslant \mathrm{Int} \to (\mathrm{Int} \to (\mu\alpha. \mathrm{Int} \to \alpha))$ ✓
    - ...

Considerations for subtyping iso-recursive types

Unfolding Lemma (expected)

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

- Positive subtyping is easy to check by comparing the type body
  - $\mu\alpha. \top \to \alpha \leqslant \mu\alpha. \mathrm{Int} \to \alpha$
- But negative subtyping (in most cases) has to be rejected.
  - $\mu\alpha. \alpha \to \mathrm{Int} \nleqslant \mu\alpha. \alpha \to \top$

Considerations for subtyping iso-recursive types

Unfolding Lemma (expected)

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

- Positive subtyping is easy to check by comparing the type body
  - $\mu\alpha. \top \to \alpha \leqslant \mu\alpha. \mathsf{Int} \to \alpha$
- But negative subtyping (in most cases) has to be rejected.
  - $\mu\alpha. \alpha \to \mathsf{Int} \nleqslant \mu\alpha. \alpha \to \top$
    - $(\mu\alpha. \alpha \to \mathsf{Int}) \to \boxed{\mathsf{Int}} \leqslant (\mu\alpha. \alpha \to \top) \to \boxed{\top}$ ✓
    - $((\mu\alpha. \alpha \to \mathsf{Int}) \to \boxed{\mathsf{Int}}) \to \mathsf{Int} \leqslant ((\mu\alpha. \alpha \to \top) \to \boxed{\top}) \to \top$ ✗

Considerations for subtyping iso-recursive types

Unfolding Lemma (expected)

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

- Positive subtyping is easy to check by comparing the type body
  - $\mu\alpha. \top \to \alpha \leqslant \mu\alpha. \mathrm{Int} \to \alpha$
- But negative subtyping (in most cases) has to be rejected.
  - $\mu\alpha. \alpha \to \mathrm{Int} \nleqslant \mu\alpha. \alpha \to \top$
- However, negative recursive types can be subtypes of themselves.
  - $\mu\alpha. \top \to \alpha \leqslant \mu\alpha. \top \to \alpha$ (by reflexivity)

Considerations for subtyping iso-recursive types

Unfolding Lemma (expected)

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

- Positive subtyping is easy to check by comparing the type body
  - $\mu\alpha. \top \to \alpha \leqslant \mu\alpha. \mathrm{Int} \to \alpha$
- But negative subtyping (in most cases) has to be rejected.
  - $\mu\alpha. \alpha \to \mathrm{Int} \not\leqslant \mu\alpha. \alpha \to \top$
- However, negative recursive types can be subtypes of themselves.
  - $\mu\alpha. \top \to \alpha \leqslant \mu\alpha. \top \to \alpha$ (by reflexivity)
- Moreover, negative variables can be subtypes of $\top$.
  - $\mu\alpha. \top \to \alpha \leqslant \mu\alpha. \alpha \to \alpha$

Considerations for subtyping iso-recursive types

Unfolding Lemma (expected)

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

- Positive subtyping is easy to check by comparing the type body
  - $\mu\alpha. \top \to \alpha \leqslant \mu\alpha. \mathsf{Int} \to \alpha$
- But negative subtyping (in most cases) has to be rejected.
  - $\mu\alpha. \alpha \to \mathsf{Int} \not\leqslant \mu\alpha. \alpha \to \top$
- However, negative recursive types can be subtypes of themselves.
  - $\mu\alpha. \top \to \alpha \leqslant \mu\alpha. \top \to \alpha$ (by reflexivity)
- Moreover, negative variables can be subtypes of $\top$.
  - $\mu\alpha. \top \to \alpha \leqslant \mu\alpha. \alpha \to \alpha$
    - $\top \to (\mu\alpha. \top \to \alpha) \leqslant (\mu\alpha. \alpha \to \alpha) \to (\mu\alpha. \alpha \to \alpha)$ ✓
    - $\ldots$

Considerations for subtyping iso-recursive types

Unfolding Lemma (expected)

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

- Positive subtyping is easy to check by comparing the type body
  - $\mu\alpha. \top \to \alpha \leqslant \mu\alpha. \mathrm{Int} \to \alpha$
- But negative subtyping (in most cases) has to be rejected.
  - $\mu\alpha. \alpha \to \mathrm{Int} \nleqslant \mu\alpha. \alpha \to \top$
- However, negative recursive types can be subtypes of themselves.
  - $\mu\alpha. \top \to \alpha \leqslant \mu\alpha. \top \to \alpha$ (by reflexivity)
- Moreover, negative variables can be subtypes of $\top$.
  - $\mu\alpha. \top \to \alpha \leqslant \mu\alpha. \alpha \to \alpha$
- Nested recursive types make the problem even trickier.
  - What is the subtyping relation between $\mu\beta. \top \to (\mu\alpha. \alpha \to \beta)$ and $\mu\beta. \mathrm{Int} \to (\mu\alpha. \alpha \to \beta)$?

Considerations for subtyping iso-recursive types

Unfolding Lemma (expected)

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

- Positive subtyping is easy to check by comparing the type body
  - $\mu\alpha. \top \to \alpha \leqslant \mu\alpha. \mathrm{Int} \to \alpha$
- But negative subtyping (in most cases) has to be rejected.
  - $\mu\alpha. \alpha \to \mathrm{Int} \nleqslant \mu\alpha. \alpha \to \top$
- However, negative recursive types can be subtypes of themselves.
  - $\mu\alpha. \top \to \alpha \leqslant \mu\alpha. \top \to \alpha$ (by reflexivity)
- Moreover, negative variables can be subtypes of $\top$.
  - $\mu\alpha. \top \to \alpha \leqslant \mu\alpha. \alpha \to \alpha$
- Nested recursive types make the problem even trickier.
  - What is the subtyping relation between $\mu\beta. \top \to (\mu\alpha. \alpha \to \beta)$ and $\mu\beta. \mathrm{Int} \to (\mu\alpha. \alpha \to \beta)$? ✗

Why iso-recurisve subtyping have been inefficient?

Amber Rules[1,2]

(Amber-rec)
$$\frac{\Gamma, \alpha \leqslant \beta \vdash A \leqslant B}{\Gamma \vdash \mu\alpha. A \leqslant \mu\beta. B}$$

(Amber-assump)
$$\frac{\alpha \leqslant \beta \in \Gamma}{\Gamma \vdash \alpha \leqslant \beta}$$

(Amber-self)
$$\frac{}{\Gamma \vdash \mu\alpha. A \leqslant \mu\alpha. A}$$

°Cardelli, "Amber".

°Amadio et al., "Subtyping recursive types".

Why iso-recurisve subtyping have been inefficient?

Amber Rules[1,2]

(Amber-rec)
$$\frac{\Gamma, \alpha \leqslant \beta \vdash A \leqslant B}{\Gamma \vdash \mu\alpha. A \leqslant \mu\beta. B}$$

(Amber-assump)
$$\frac{\alpha \leqslant \beta \in \Gamma}{\Gamma \vdash \alpha \leqslant \beta}$$

(Amber-self)
$$\frac{}{\Gamma \vdash \mu\alpha. A \leqslant \mu\alpha. A}$$

- Amber-rec rule deals with recursive subtyping, and rules out the problematic negative subtyping cases:
  - $\mu\alpha. \top \to \alpha \leqslant \mu\beta. \mathrm{Int} \to \beta$
  - $\mu\alpha. \alpha \to \mathrm{Int} \not\leqslant \mu\beta. \beta \to \top$

[1] Cardelli, "Amber".

[2] Amadio et al., "Subtyping recursive types".

Why iso-recurisve subtyping have been inefficient?

Amber Rules[1,2]

(Amber-rec)
$$\frac{\Gamma, \alpha \leqslant \beta \vdash A \leqslant B}{\Gamma \vdash \mu\alpha. A \leqslant \mu\beta. B}$$

(Amber-assump)
$$\frac{\alpha \leqslant \beta \in \Gamma}{\Gamma \vdash \alpha \leqslant \beta}$$

(Amber-self)
$$\frac{}{\Gamma \vdash \mu\alpha. A \leqslant \mu\alpha. A}$$

- Amber-rec rule deals with recursive subtyping, and rules out the problematic negative subtyping cases:
  - $\mu\alpha. \top \to \alpha \leqslant \mu\beta. \mathsf{Int} \to \beta$
  - $\mu\alpha. \alpha \to \mathsf{Int} \nleqslant \mu\beta. \beta \to \top$
- However, to ensure reflexivity for negative subtyping, Amber-self rule is needed. (⇒ backtracking, costly for nested types)
  - $\mu\alpha. \alpha \to \mathsf{Int} \leqslant \mu\alpha. \alpha \to \mathsf{Int}$

[1] Cardelli, "Amber".
[2] Amadio et al., "Subtyping recursive types".

Why iso-recurisve subtyping have been inefficient?

Amber Rules[1,2]

(Amber-rec)
$$\frac{\Gamma, \alpha \leqslant \beta \vdash A \leqslant B}{\Gamma \vdash \mu\alpha. A \leqslant \mu\beta. B}$$

(Amber-assump)
$$\frac{\alpha \leqslant \beta \in \Gamma}{\Gamma \vdash \alpha \leqslant \beta}$$

(Amber-self)
$$\frac{}{\Gamma \vdash \mu\alpha. A \leqslant \mu\alpha. A}$$

- Amber-rec rule deals with recursive subtyping, and rules out the problematic negative subtyping cases:
  - $\mu\alpha. \top \to \alpha \leqslant \mu\beta. \mathrm{Int} \to \beta$
  - $\mu\alpha. \alpha \to \mathrm{Int} \nleqslant \mu\beta. \beta \to \top$
- However, to ensure reflexivity for negative subtyping, Amber-self rule is needed. (⇒ backtracking, costly for nested types)
  - $\mu\alpha. \alpha \to \mathrm{Int} \leqslant \mu\alpha. \alpha \to \mathrm{Int}$
- Amber-rec requires variable names to be distinct, which is non-trivial and requires extra runtime overhead for renaming.

[1] Cardelli, "Amber".

[2] Amadio et al., "Subtyping recursive types".

QuickSub: Efficient Iso-Recursive Subtyping

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

Though written as a judgment, QuickSub can be easily interpreted as an algorithm.

- Input: subtyping context Ψ, polarity mode $\oplus$, types A and B.
- Output: subtyping result ($\lesssim ::= < \mid \approx_S$), or failure where no rules apply.
- Equivalent to Amber rules.

Key idea (1) - tracking polarity

Subtyping Context	Ψ	$::= \Psi, \alpha^{\oplus} \mid \cdot$
Polarity Mode	$\oplus$	$::= + \mid -$
Subtyping Results	$\lessapprox$	$::= < \mid \approx$

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

(QSub-RecLt)

$$\frac{\Psi, \alpha^{\oplus} \vdash_{\oplus} A < B}{\Psi \vdash_{\oplus} \mu\alpha. A < \mu\alpha. B}$$

(QSub-Fun)

$$\frac{\Psi \vdash_{\overline{\oplus}} A_2 \lessapprox_1 B_2 \quad \Psi \vdash_{\oplus} A_1 \lessapprox_2 B_1}{\Psi \vdash_{\oplus} A_1 \to A_2 (\lessapprox_1 \bullet \lessapprox_2) B_1 \to B_2}$$

When $\alpha^{\oplus}$ is the same as $\vdash_{\oplus}$, α is positive (to the right of $\to$'s)
When $\alpha^{\overline{\oplus}}$ is the flip of $\vdash_{\oplus}$, α is negative (to the left of $\to$'s)

Key idea (2) - distinguish strict subtypes from equivalence

Subtyping results are precisely tracked from the base cases:

(QSub-Int)
$$\frac{}{\Psi \vdash_{\oplus} \mathsf{Int} \approx \mathsf{Int}}$$

(QSub-Top)
$$\frac{}{\Psi \vdash_{\oplus} \top \approx \top}$$

(QSub-NTop)
$$\frac{A \neq \top}{\Psi \vdash_{\oplus} A < \top}$$

...

Key idea (2) - distinguish strict subtypes from equivalence

Subtyping results are precisely tracked from the base cases:

(QSub-Int) $$\frac{}{\Psi \vdash_{\oplus} \text{Int} \approx \text{Int}}$$

(QSub-Top) $$\frac{}{\Psi \vdash_{\oplus} \top \approx \top}$$

(QSub-NTop) $$\frac{A \neq \top}{\Psi \vdash_{\oplus} A < \top}$$

$\ldots$

The results are composed accordingly in the function case:

$$\approx \bullet \approx \; = \; \approx \qquad \approx \bullet < \; = \; <$$
$$< \bullet < \; = \; < \qquad < \bullet \approx \; = \; <$$

Key idea (2) - distinguish strict subtypes from equivalence

Subtyping results are precisely tracked from the base cases:

(QSub-Int)
$$\frac{}{\Psi \vdash_{\oplus} \mathrm{Int} \approx \mathrm{Int}}$$

(QSub-Top)
$$\frac{}{\Psi \vdash_{\oplus} \top \approx \top}$$

(QSub-NTop)
$$\frac{A \neq \top}{\Psi \vdash_{\oplus} A < \top}$$

$\dots$

The results are composed accordingly in the function case:

$$\begin{aligned} \approx \bullet \approx &= \approx & \qquad \approx \bullet < &= < \\ < \bullet < &= < & \qquad < \bullet \approx &= < \end{aligned}$$

Positive variables can be simply considered equal.

(QSub-VarPos)
$$\frac{\alpha^{\oplus} \in \Psi}{\Psi \vdash_{\oplus} \alpha \approx \alpha}$$

Key idea (2) - distinguish strict subtypes from equivalence

Subtyping results are precisely tracked from the base cases:

(QSub-Int)
$$\frac{}{\Psi \vdash_{\oplus} \mathrm{Int} \approx \mathrm{Int}}$$

(QSub-Top)
$$\frac{}{\Psi \vdash_{\oplus} \top \approx \top}$$

(QSub-NTop)
$$\frac{A \neq \top}{\Psi \vdash_{\oplus} A < \top}$$

$\ldots$

The results are composed accordingly in the function case:

$$\approx \bullet \approx \;=\; \approx \qquad \approx \bullet < \;=\; <$$
$$< \bullet < \;=\; < \qquad < \bullet \approx \;=\; <$$

Positive variables can be simply considered equal.

(QSub-VarPos)
$$\frac{\alpha^{\oplus} \in \Psi}{\Psi \vdash_{\oplus} \alpha \approx \alpha}$$

✓ Positive subtyping: $\mu\alpha.\, \top \to \alpha < \mu\alpha.\, \mathrm{Int} \to \alpha$

Negative recursive subtyping

Goal: For efficiency, we want to check subtyping by directly comparing the type body, without backtracking or extra unfolding.

Negative recursive subtyping

Goal: For efficiency, we want to check subtyping by directly comparing the type body, without backtracking or extra unfolding.

Observation: When negative recursive variables are compared with themselves, their recursive types have to be equivalent.

causes failure

$$\dfrac{\dfrac{\alpha^{+} \vdash_{-} \alpha \approx \alpha \qquad \alpha^{+} \vdash_{+} \mathsf{Int} < \top}{\alpha^{+} \vdash_{+} \alpha \to \mathsf{Int} < \alpha \to \top}}{\cdot \vdash_{+} \mu\alpha.\alpha \to \mathsf{Int} \not< \mu\alpha.\alpha \to \top} \; ✗$$

Negative recursive subtyping

Goal: For efficiency, we want to check subtyping by directly comparing the type body, without backtracking or extra unfolding.

Observation: When negative recursive variables are compared with themselves, their recursive types have to be equivalent.

causes failure

$$\dfrac{\dfrac{\alpha^{+} \vdash_{-} \alpha \approx \alpha \qquad \alpha^{+} \vdash_{+} \mathrm{Int} < \top}{\alpha^{+} \vdash_{+} \alpha \to \mathrm{Int} < \alpha \to \top}}{\cdot \vdash_{+} \mu\alpha.\alpha \to \mathrm{Int} \not< \mu\alpha.\alpha \to \top} \ ✗$$

$$\dfrac{\dfrac{\alpha^{+} \vdash_{-} \alpha < \top \qquad \alpha^{+} \vdash_{+} \alpha \approx \alpha}{\alpha^{+} \vdash_{+} \top \to \alpha < \alpha \to \alpha}}{\cdot \vdash_{+} \mu\alpha.\top \to \alpha < \mu\alpha.\alpha \to \alpha} \ ✓$$

Key idea: equality variable set

Subtyping Context	Ψ	$::= \Psi, \alpha^{\oplus} \mid \cdot$
Polarity Mode	$\oplus$	$::= + \mid -$
Subtyping Results	$\lessapprox$	$::= < \mid \approx_S$
Equality Var. Set	S	$::= \emptyset \mid \{\alpha_1, \ldots, \alpha_n\}$

Key idea: equality variable set

Subtyping Context	Ψ	$::= \Psi, \alpha^{\oplus} \mid \cdot$
Polarity Mode	$\oplus$	$::= + \mid -$
Subtyping Results	$\lessapprox$	$::= < \mid \approx_S$
Equality Var. Set	S	$::= \emptyset \mid \{\alpha_1, \ldots, \alpha_n\}$

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

(QSub-VarNeg)

$$\frac{\alpha^{\overline{\oplus}} \in \Psi}{\Psi \vdash_{\oplus} \alpha \approx_{\{\alpha\}} \alpha}$$

(QSub-VarPos)

$$\frac{\alpha^{\oplus} \in \Psi}{\Psi \vdash_{\oplus} \alpha \approx_{\emptyset} \alpha}$$

Key idea: equality variable set

Subtyping Context	Ψ	$::= \Psi, \alpha^{\oplus} \mid \cdot$
Polarity Mode	$\oplus$	$::= + \mid -$
Subtyping Results	$\lessapprox$	$::= < \mid \approx_S$
Equality Var. Set	S	$::= \emptyset \mid \{\alpha_1, \ldots, \alpha_n\}$

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

(QSub-VarNeg)

$$\frac{\alpha^{\overline{\oplus}} \in \Psi}{\Psi \vdash_{\oplus} \alpha \approx_{\{\alpha\}} \alpha}$$

(QSub-VarPos)

$$\frac{\alpha^{\oplus} \in \Psi}{\Psi \vdash_{\oplus} \alpha \approx_{\emptyset} \alpha}$$

(QSub-Fun)

$$\frac{\Psi \vdash_{\overline{\oplus}} A_2 \lessapprox_1 B_2 \quad \Psi \vdash_{\oplus} A_1 \lessapprox_2 B_1}{\Psi \vdash_{\oplus} A_1 \to A_2 (\lessapprox_1 \bullet \lessapprox_2) B_1 \to B_2}$$

$$\begin{aligned} \approx_{S_1} \bullet \approx_{S_2} &= \approx_{S_1 \cup S_2} \\ \approx_{\emptyset} \bullet < &= < \\ < \bullet < &= < \\ < \bullet \approx_{\emptyset} &= < \end{aligned}$$

Otherwise, $\lessapprox_1 \bullet \lessapprox_2$ fails

✓ $\mu\alpha.\,\alpha \to \mathrm{Int} \not\leq \mu\alpha.\,\alpha \to \top$

Key idea: equality variable set

$$\boxed{\Psi \vdash_{\oplus} A \lesssim B}$$

Subtyping Context	Ψ	$::= \Psi, \alpha^{\oplus} \mid \cdot$
Polarity Mode	$\oplus$	$::= + \mid -$
Subtyping Results	$\lesssim$	$::= < \mid \approx_S$
Equality Var. Set	S	$::= \emptyset \mid \{\alpha_1, \ldots, \alpha_n\}$

(QSub-VarNeg)

$$\frac{\alpha^{\overline{\oplus}} \in \Psi}{\Psi \vdash_{\oplus} \alpha \approx_{\{\alpha\}} \alpha}$$

(QSub-RecEq)

$$\frac{\Psi, \alpha^{\oplus} \vdash_{\oplus} A \approx_S B \quad \ldots}{\Psi \vdash_{\oplus} \mu\alpha. A \approx_{S'} \mu\alpha. B}$$

(QSub-VarPos)

$$\frac{\alpha^{\oplus} \in \Psi}{\Psi \vdash_{\oplus} \alpha \approx_{\emptyset} \alpha}$$

(QSub-Fun)

$$\frac{\Psi \vdash_{\overline{\oplus}} A_2 \lesssim_1 B_2 \quad \Psi \vdash_{\oplus} A_1 \lesssim_2 B_1}{\Psi \vdash_{\oplus} A_1 \to A_2 (\lesssim_1 \bullet \lesssim_2) B_1 \to B_2}$$

$$\begin{aligned} \approx_{S_1} \bullet \approx_{S_2} &= \approx_{S_1 \cup S_2} \\ \approx_{\emptyset} \bullet < &= < \\ < \bullet < &= < \\ < \bullet \approx_{\emptyset} &= < \end{aligned}$$

Otherwise, $\lesssim_1 \bullet \lesssim_2$ fails

✓ $\mu\alpha. \alpha \to \text{Int} \nleq \mu\alpha. \alpha \to \top$

✓ Reflexive subtyping.

QuickSub in Functional Style

$$
\begin{array}{lcll}
\mathsf{Sub}_{\Psi}(\mathsf{Int}, \mathsf{Int}, \oplus) & = & \approx_{\emptyset} & \\
\mathsf{Sub}_{\Psi}(\top, \top, \oplus) & = & \approx_{\emptyset} & \\
\mathsf{Sub}_{\Psi}(A, \top, \oplus) & = & < & (\text{if } A \neq \top) \\
\mathsf{Sub}_{\Psi}(\alpha, \alpha, \oplus) & = & \approx_{\emptyset} & (\text{if } \alpha^{\oplus} \in \Psi) \\
\mathsf{Sub}_{\Psi}(\alpha, \alpha, \oplus) & = & \approx_{\{\alpha\}} & (\text{if } \alpha^{\overline{\oplus}} \in \Psi) \\
\mathsf{Sub}_{\Psi}(A_1 \to A_2, B_1 \to B_2, \oplus) & = & \mathsf{Sub}_{\Psi}(A_2, A_1, \overline{\oplus}) \bullet \mathsf{Sub}_{\Psi}(B_1, B_2, \oplus) & \\
\mathsf{Sub}_{\Psi}(\mu\alpha.\, A_1, \mu\alpha.\, A_2, \oplus) & = & < & (\text{if } \mathsf{Sub}_{\Psi, \alpha^{\oplus}}(A_1, A_2, \oplus) = <) \\
\mathsf{Sub}_{\Psi}(\mu\alpha.\, A_1, \mu\alpha.\, A_2, \oplus) & = & \approx_{S} & (\text{if } \mathsf{Sub}_{\Psi, \alpha^{\oplus}}(A_1, A_2, \oplus) = \approx_{S} \text{ and } \alpha \notin S) \\
\mathsf{Sub}_{\Psi}(\mu\alpha.\, A_1, \mu\alpha.\, A_2, \oplus) & = & \approx_{(S \cup \mathsf{FV}(A_1)) \setminus \{\alpha\}} & (\text{otherwise})
\end{array}
$$

otherwise, $\mathsf{Sub}_{\Psi}(A, B, \oplus)$ fails

- No backtracking $\Rightarrow O(n)$ traversal ($n =$ size of types)
- Set operations can be optimized with imperative data structures
  - $\Rightarrow O(m)$ cost in the worst case ($m = \#$ of recursive variables)
  - $\Rightarrow O(1)$ for practical cases
  - $\Rightarrow$ Overall: $O(mn)$ cost in the worst case, linear for practical cases

Evaluation

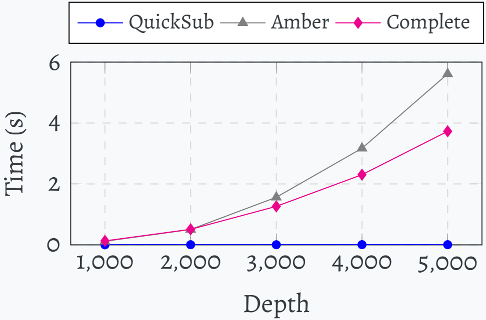
- Implement QuickSub in OCaml.
- Compare performance with existing algorithms:
  - Amber rules
  - Nominal unfolding rules[1]
    - Equivalent to Amber rules, addressing metatheory challenges.
    - Though being algorithmic, not designed with efficiency in mind.
  - Complete iso-recursive subtyping[2].
    - More expressive than Amber rules.
    - Ship with an $O(mn)$ algorithm.
  - Equi-recursive subtyping[3] (see paper).
- Benchmarks for different recursive type patterns and depths.

[1] Y. Zhou et al., "Revisiting Iso-recursive subtyping".
[2] Ligatti et al., "On subtyping-relation completeness, with an application to iso-recursive types".
[3] Gapeyev et al., "Recursive subtyping revealed".

Nested positive subtyping, growing depths

```
class A {
  foo(x: Int) : A
   ...
   class B {
     bar(y: Real) : A
   }
}
```

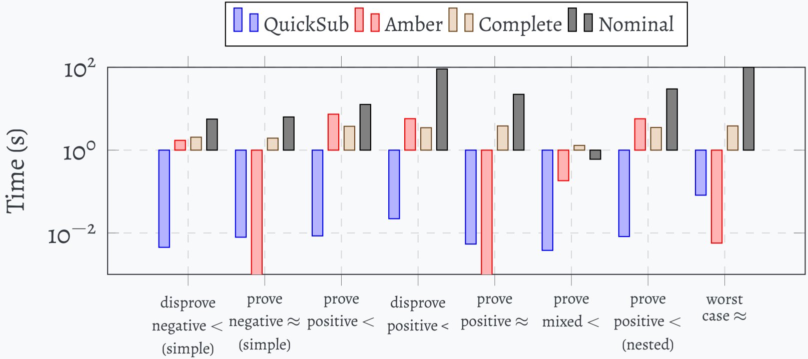
Complex Nested Objects



$$\mu\alpha_1.\,\mathrm{Int} \to (\mu\alpha_2.\,\mathrm{Int} \to \ldots(\mu\alpha_n.\,(\alpha_1, \ldots, \alpha_n)))$$

For positive nested recursive subtyping, Amber and Complete are quadratic in complexity, while QuickSub is linear.

Benchmark results, algorithm runtime at a large depth

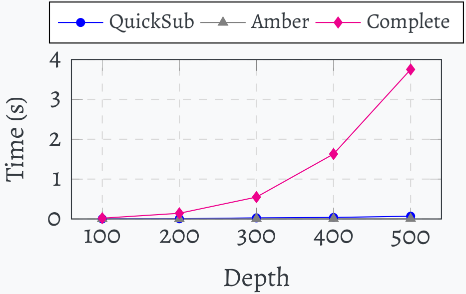


- QuickSub outperforms other algorithms in most cases
  - except in reflexive cases, where Amber performs faster (expected)
- Handles both simple and nested recursive types efficiently.
- Linear performance in practical scenarios.

Negative recursive subtyping, worst case

$$\mu\alpha_1.\ \alpha_1 \to (\mu\alpha_2.\ \alpha_1 \to \alpha_2 \to (\mu\alpha_3.\ \alpha_1 \to \alpha_2 \to \alpha_3 \to (\mu\alpha_4.\ \ldots \ldots)))$$

Worst Case Pattern



The worst case scenario only occurs when all variables are negative and the subtyping result is $\approx$, so that all variables are added to the equality variable set S. ($|S|_{\max} = m$)

The complexity is $O(mn)$. ($m = \#$ of variables, $n =$ size of types).

With the imperative optimization, QuickSub still demonstrates an efficient performance.

Conclusion

QuickSub, an efficient algorithm for iso-recursive subtyping, with linear complexity in practice.

More in the paper

- Equivalence proof to other iso-recursive subtyping formulations.
- Type soundness proof for a calculus using QuickSub.
- Extension to record types.

Future Work

- Extending QuickSub to handle more type system features.
- Applying QuickSub to deal with equi-recursive subtyping.

- Amadio, Roberto M et al. “Subtyping recursive types”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15.4 (1993), pp. 575–631. DOI: 10.1145/155183.155231.
- Cardelli, Luca. “Amber”. In: *Combinators and Functional Programming Languages: Thirteenth Spring School of the LITP Val d'Ajol, France, May 6–10, 1985 Proceedings* (1985). DOI: 10.1007/3-540-17184-3.
- Chugh, Ravi. “IsoLATE: A type system for self-recursion”. In: *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 24*. Springer. 2015, pp. 257–282. DOI: 10.1007/978-3-662-46669-8_11.
- Dreyer, Derek R. et al. *Toward a Practical Type Theory for Recursive Modules*. Tech. rep. CMU-CS-01-112. School of Computer Science, Carnegie Mellon University, 2001. DOI: 10.21236/ada460172.
- Gapeyev, Vladimir et al. “Recursive subtyping revealed”. In: *Journal of Functional Programming* 12.6 (2002), pp. 511–548. DOI: 10.1017/S0956796802004318.
- Ligatti, Jay et al. “On subtyping-relation completeness, with an