QuickSub: Efficient Iso-Recursive Subtyping

Bruno C. d. S. Oliveira (joint work with Litao Zhou)

Canberra WG2.1 Meeting

Motivation for QuickSub

- Recursive types are essential in many programming languages.
- Two main approaches: Equi-recursive and Iso-recursive types.
- In languages with subtyping we need to also consider **recursive subtyping**.
- Efficient algorithms for iso-recursive subtyping remain **understudied**.

Equi-recursive Types

- Treat recursive types and their unfoldings as identical.
- Example: $\mu\alpha.\alpha \to \alpha = (\mu\alpha.\alpha \to \alpha) \to (\mu\alpha.\alpha \to \alpha)$
- Advantages:
  - Convenient.
  - No need for explicit fold/unfold operations.
- Disadvantages:
  - Requires coinductive reasoning, which is costly (in terms of performance)[1].
  - Metatheory complications: $F_{<:}$ with recursive types, ML Modules.
  - Difficult to extend with more advanced type system features.

[1] Andreas Rossberg. Mutually Iso-Recursive Subtyping. OOPSLA 2023

Iso-recursive Types

- Treat recursive types and their unfoldings as different.
- Example: $\mu\alpha.\alpha \to \alpha$ and $(\mu\alpha.\alpha \to \alpha) \to \alpha$ are distinct.
- Advantages:
  - Easier to scale to more advanced features.
  - Simpler metatheory.
  - Lower computational complexity.
- Disadvantages:
  - Less convenience.
  - Operational semantics complicated by fold/unfold.

Recursive Subtyping: 3 Approaches

3 Approaches with different expressive power:

- (Inductive) **Amber**-style iso-recursive subtyping.
- (Coinductive) **Complete** iso-recursive subtyping.
- (Coinductive) **Equi**-recursive subtyping.

Expressive power comparison:

$$\textbf{Amber} < \textbf{Complete} < \textbf{Equi}$$

But **equi**-recursive subtyping can be expressed as **Amber** + equi-recursive equivalence[2]:

$$A \leq_e B \triangleq \exists C_1\ C_2.\ A \doteq C_1 \wedge C_1 \leq_i C_2 \wedge C_2 \doteq B.$$

[2] Litao Zhou, Qianyong Wan, and Bruno C. d. S. Oliveira. OOPSLA 2024.

Why QuickSub?

- An efficient algorithm for **Amber** iso-recursive subtyping is missing.

Subtyping **Amber**-style Recursive Types

- Efficient subtyping for iso-recursive types is **challenging**.
- We assume standard subtyping rules for other constructs:

$$\frac{}{A \leq \top} \qquad \frac{}{\mathsf{nat} \leq \mathsf{nat}} \qquad \frac{B_1 \leq A_1 \qquad A_2 \leq B_2}{A_1 \to A_2 \leq B_1 \to B_2}$$

- How to determine if one recursive type is a subtype of another for iso-recursive subtyping?
- We expect that recursive type unrolling preserve subtyping:

$$\text{If } \mu\alpha.A \leq \mu\alpha.B \text{ then } A\,[\alpha \mapsto \mu\alpha.A] \leq B\,[\alpha \mapsto \mu\alpha.B].$$

Example: Positive Recursive Subtyping

- $\mu\alpha.T \to \alpha \leq \mu\alpha.\ \mathsf{nat} \to \alpha$
- The left type can be regarded as a function that consumes infinite values of any type.
- The right type consumes infinite nat values.
- The left type is more general than the right type.
- **Positive subtyping is easy**: just compare the bodies in the usual way!

Example: Negative Recursive Subtyping

- $\mu\alpha.\ \alpha \to \mathsf{nat} \not\leq \mu\alpha.\ \alpha \to T$
- The left type expects an input of a specific type producing nat values.
- The right type expects an input of a specific type producing any values.
- The subtyping statement does not hold, since unrollings do not preserve subtyping.

$$((\mu\alpha.\ \alpha \to \mathsf{nat}) \to \mathsf{nat}) \to \mathsf{nat} \not\leq ((\mu\alpha.\ \alpha \to \top) \to \top) \to \top$$

- Negative subtyping holds for **reflexivity** (example $\mu\alpha.\ \alpha \to \mathsf{nat} \leq \mu\alpha.\ \alpha \to \mathsf{nat}$), and little else.

Nested Recursive Subtyping

- Example: $\mu\beta.\ \mathsf{T} \to (\mu\alpha.\ \alpha \to \beta) \leq \mu\beta.\ \mathsf{nat} \to (\mu\alpha.\ \alpha \to \beta)$?
- **Question:** Should these be subtypes?

Nested Recursive Subtyping

- Example: $\mu\beta.T \to (\mu\alpha.\alpha \to \beta) \not\leq \mu\beta.\mathsf{nat} \to (\mu\alpha.\alpha \to \beta)$
- The variable β appears to be in a positive position.
- However, due to the variable α appearing negatively, the types are not related by subtyping.
- Complex interactions between recursive variables.
- We can see that unrollings do not preserve subtyping!

$$\mu\beta.\ \top \to ((\mu\alpha.\ \alpha \to \beta) \to \beta) \to \beta$$

$$\not\leq$$

$$\mu\beta.\ \mathsf{nat} \to ((\mu\alpha.\ \alpha \to \beta) \to \beta) \to \beta$$

Amber Rules

- Traditional Amber rules for iso-recursive subtyping.

$$\frac{\Delta, \alpha \le \beta \vdash A \le B}{\Delta \vdash \mu\alpha.A \le \mu\beta.B}\text{(Amber-rec)}$$

$$\frac{}{\Delta \vdash \mu\alpha.A \le \mu\alpha.A}\text{(Amber-self)}$$

- Amber-rec: Compares recursive types by their bodies.
- Amber-self: Handles reflexivity for negative recursive types.
- **Backtracking** is required.
- Variable renaming issues.
- Reflexivity is complex for subtyping relations that are not antisymmetric.

Nominal Unfolding Rules

- Proposed by Zhou et al. (TOPLAS 2022)
- Recursive type bodies are unfolded using labeled types.

$$\frac{\Gamma, \alpha \vdash [\alpha \mapsto A^{\alpha}]A \leq [\alpha \mapsto B^{\alpha}]B}{\Gamma \vdash \mu\alpha.A \leq \mu\alpha.B}\text{(Sn-rec)}$$

- **Exponential** blowup in size due to substitution.
  - Generally a problem for substitution-based algorithms.

Key Ideas in QuickSub

- Distinguishing **strict subtyping** from **equivalence**.
- Tracking **polarities**: Positive vs. Negative occurrences.
  - But cannot be done naively!
  - Handling negative recursive subtyping with **equality variable sets**.

Distinguishing Strict Subtyping from Equivalence

- Algorithm does not just compute **True** or **False**.
- Instead the algorithm returns **3 possible results**:
  - Two types are equivalent ($\approx$).
  - The first type is a strict subtype of the second ($<$).
  - The types have no relation.
- Example: $\mu\alpha.\alpha \to \mathsf{nat} < \mu\alpha.\alpha \to \top$
- Example: $\mu\alpha.\top \to \alpha \approx \mu\alpha.\top \to \alpha$
- Helpful to avoid backtracking for reflexivity.

Handling Negative Recursive Subtyping

- Reflexive uses of negative subtyping variables need to be tracked:

causes failure

$$\dfrac{\dfrac{\alpha^- \approx \alpha^- \qquad \mathsf{nat} < \top}{\alpha^- \to \mathsf{nat} < \alpha^- \to \top}}{\mu\alpha.\alpha \to \mathsf{nat} \not< \mu\alpha.\alpha \to \top}\ \text{✗} \qquad\qquad \dfrac{\dfrac{\alpha^- < \top \qquad \alpha^+ \approx \alpha^+}{\top \to \alpha^+ < \alpha^- \to \alpha^+}}{\mu\alpha.\top \to \alpha < \mu\alpha.\alpha \to \alpha}\ \text{✓}$$

- Furthermore, we also need to track "fake" positive variables.
- Positive variables and other uses of negative variables do not cause trouble.
- QuickSub employs **equality variable sets** for this.

The QuickSub Algorithm

Syntax:

Types	A, B	$::=$	$\mathsf{nat} \mid \top \mid A_1 \to A_2 \mid \alpha \mid \mu\alpha.\ A$
Subtyping results	$\lessapprox$	$::=$	$< \quad \mid \quad \approx_S$
Polarity modes	$\oplus$	$::=$	$+ \mid -$
Subtyping contexts	Ψ	$::=$	$\cdot \mid \Psi, \alpha^{\oplus}$
Equality variable sets	S	$::=$	$\emptyset \mid \{\alpha_1, \ldots, \alpha_n\}$

The QuickSub Algorithmic rules

$\boxed{\Psi \vdash_{\oplus} A \lesseqapprox B}$ (QuickSub *Subtyping*)

$$\textsc{QS-nat}\ \frac{}{\Psi \vdash_{\oplus} \mathsf{nat} \approx_{\emptyset} \mathsf{nat}} \qquad \textsc{QS-TopEq}\ \frac{}{\Psi \vdash_{\oplus} \top \approx_{\emptyset} \top} \qquad \textsc{QS-TopLt}\ \frac{A \neq \top}{\Psi \vdash_{\oplus} A < \top}$$

$$\textsc{QS-VarPos}\ \frac{\alpha^{\oplus} \in \Psi}{\Psi \vdash_{\oplus} \alpha \approx_{\emptyset} \alpha} \qquad \textsc{QS-VarNeg}\ \frac{\alpha^{\overline{\oplus}} \in \Psi}{\Psi \vdash_{\oplus} \alpha \approx_{\{\alpha\}} \alpha}$$

$$\textsc{QS-RecLt}\ \frac{\Psi, \alpha^{\oplus} \vdash_{\oplus} A_1 < A_2}{\Psi \vdash_{\oplus} \mu\alpha.\ A_1 < \mu\alpha.\ A_2} \qquad \textsc{QS-RecEq}\ \frac{\Psi, \alpha^{\oplus} \vdash_{\oplus} A_1 \approx_S A_2 \qquad \alpha \notin S}{\Psi \vdash_{\oplus} \mu\alpha.\ A_1 \approx_S \mu\alpha.\ A_2}$$

$$\textsc{QS-RecEqIn}\ \frac{\Psi, \alpha^{\oplus} \vdash_{\oplus} A_1 \approx_S A_2 \qquad \alpha \in S}{\Psi \vdash_{\oplus} \mu\alpha.\ A_1 \approx_{((S \cup \mathsf{FV}(A_1)) \setminus \{\alpha\})} \mu\alpha.\ A_2} \qquad \textsc{QS-Arrow}\ \frac{\Psi \vdash_{\overline{\oplus}} A_2 \lesseqapprox_1 A_1 \qquad \Psi \vdash_{\oplus} B_1 \lesseqapprox_2 B_2}{\Psi \vdash_{\oplus} A_1 \to A_2\ (\lesseqapprox_1 \bullet \lesseqapprox_2)\ B_1 \to B_2}$$

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

Functional QuickSub

QuickSub rules in a functional style:

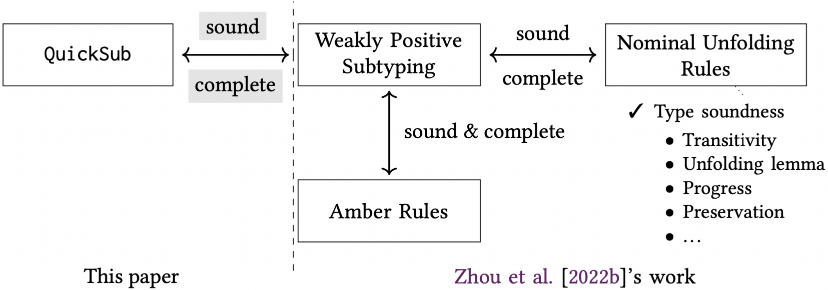
$$
\begin{array}{lcll}
\mathrm{Sub}_\Psi(\mathsf{nat}, \mathsf{nat}, \oplus) & = & \approx_\emptyset & \\
\mathrm{Sub}_\Psi(\top, \top, \oplus) & = & \approx_\emptyset & \\
\mathrm{Sub}_\Psi(A, \top, \oplus) & = & < & (\text{if } A \neq \top) \\
\mathrm{Sub}_\Psi(\alpha, \alpha, \oplus) & = & \approx_\emptyset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

Efficiently Updating Equality Variable Sets

- Use imperative data structures for efficiency.
- Boolean arrays to represent equality variable sets.
- Set union operation is linear with respect to the number of variables.
- Overall complexity: $O(mn)$, where m is the size of the type and n is the number of recursive variables.
- Optimized QuickSub maintains linear complexity for common cases.

Equivalence to Amber Rules and Type Soundness

Equivalence proof to several to the Amber rules + type soundness:



Evaluation

- Implement QuickSub in OCaml.
- Compare performance with existing algorithms:
  - Amber rules
  - Nominal unfolding
  - Complete iso-recursive subtyping[3]
  - Equi-recursive subtyping[4]
- Benchmarks for different recursive type patterns and depths.

[3] Jay Ligatti, Jeremy Blackburn, and Michael Nachtigal. 2017. On subtyping-relation completeness, with an application to iso-recursive types. TOPLAS (2017).

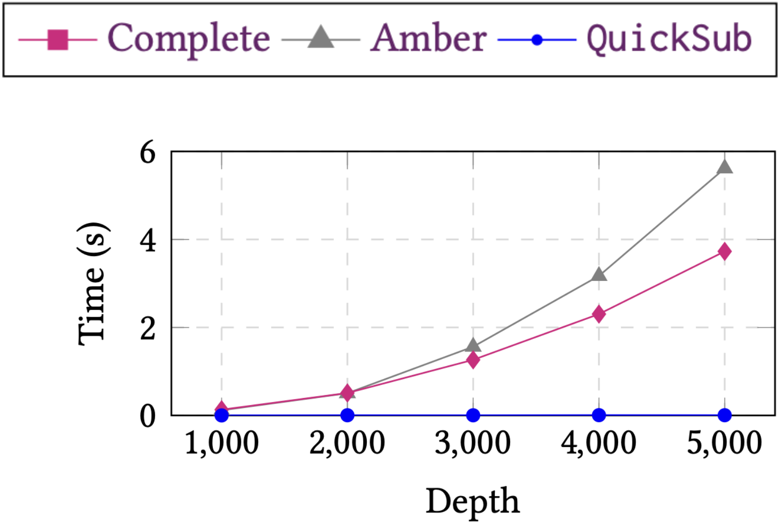
[4] Vladimir Gapeyev, Michael Y Levin, and Benjamin C Pierce. Recursive subtyping revealed. JFP (2002).

Benchmark Results

No.	QuickSub	Amber Cardelli	Complete Ligatti	Nominal Zhou	Equi Gapeyev	$\lvert S \rvert_{max}$
1	**0.0045**	1.7230	2.0541	5.6194	42.0146	1
2	0.0079	**0.0004**	1.9483	6.3181	41.6360	1
3	**0.0085**	7.3775	3.7602	12.6697	Timeout	0
4	**0.0221**	5.7502	3.4782	91.0706	Timeout	0
5	0.0054	**0.0006**	3.8383	22.2383	Timeout	0
6	**0.0038**	0.1829	1.2995	0.6027	Timeout	1
7	**0.0082**	5.7185	3.5229	30.0276	Timeout	0
8	0.0817	**0.0057**	3.8423	Timeout	Timeout	500 (worst)

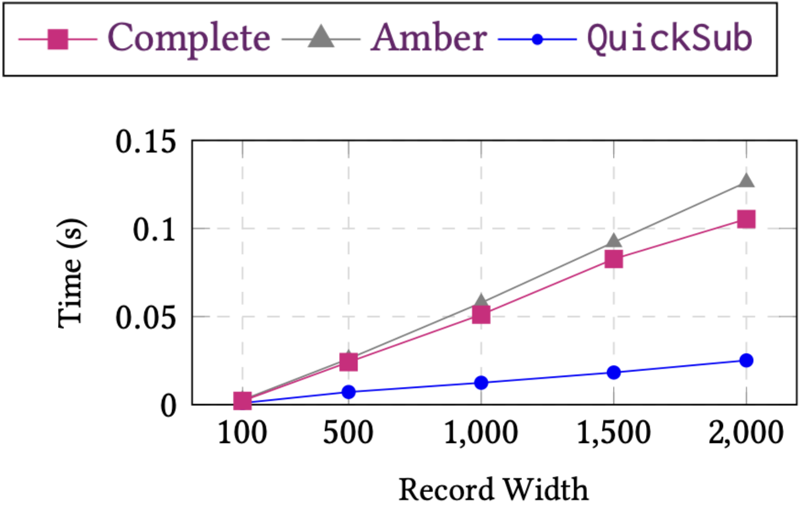
- Various tests with comparing recursive types with depth 5000 (1-7) or 500 (8). Time in seconds.
- QuickSub fastest in 5 out of 8.
- Amber faster for reflexivity (as expected).

Benchmark Results



Test (7) Proving nested positive subtyping

Benchmark Results



Depth = 10

Benchmark Results

- QuickSub outperforms other algorithms in most cases.
- Handles both simple and nested recursive types efficiently.
- Linear performance in practical scenarios.

Open Challenges

- Proof of equivalence to Amber is complex.
- QuickSub is a straightforward recursive functional program. Can we calculate QuickSub from one of the possible specifications?

Conclusion

- QuickSub provides an efficient solution for iso-recursive subtyping.
- Equivalence to Amber rules ensures correctness.
- Direct type soundness proof simplifies extensions and adaptations.
- QuickSub handles record types efficiently, broadening applicability.