Recursive Subtyping for All

Litao Zhou[1], Yaoda Zhou[1] and Bruno C. d. S. Oliveira

For POPL 2023

The University of Hong Kong

Two prominent features in OOP

- **Recursive Types** $\mu\alpha\ .\ A$
  - Model recursive data structures and recursive object types
- **Bounded Quantification** $\forall(\alpha \leq A)\ .\ B$
  - Addresses the interactions between parametric polymorphism and subtyping
  - Polymorphic variables can have a subtyping bound

$F_{\leq}$ with recursive types

Some landmark papers on the foundations of OOP:

- Inheritance is not subtyping (Cook et al. 1989)
- F-bounded polymorphism for object-oriented programming (Canning et al. 1989)
- An interpretation of objects and object types (Abadi et al. 1996)
- …

all essentially assumed some $F_{\leq}$ variant with recursive types

Recursive types

- Recursive types have 2 flavours:
  - Equi-recursive: $\mu\alpha\,.\,A \triangleq [\alpha \mapsto \mu\alpha\,.\,A]\,A$
  - Iso-recursive:

$$\frac{\Gamma \vdash e : \mu\alpha\,.\,A}{\Gamma \vdash \text{unfold } [\mu\alpha\,.\,A]\ e : [\alpha \mapsto \mu\alpha\,.\,A]\,A}$$

$$\frac{\Gamma \vdash e : [\alpha \mapsto \mu\alpha\,.\,A]\,A}{\Gamma \vdash \text{fold } [\mu\alpha\,.\,A]\ e : \mu\alpha\,.\,A}$$

- Amadio and Cardelli's Amber rules (1993) for recursive subtyping:

$$\frac{\Gamma, \alpha \le \beta \vdash A \le B \quad \alpha \neq \beta}{\Gamma \vdash \mu\alpha\,.\,A \le \mu\beta\,.\,B} \qquad \frac{\alpha \le \beta \in \Gamma}{\Gamma \vdash \alpha \le \beta} \qquad \frac{A = B}{\Gamma \vdash A \le B}$$

Bounded quantification

- The standard calculus with bounded quantification, $F_{\leq}$, has two common variants.
- The full $F_{\leq}$, where the bounds are allowed to be contra-variant:

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

  The full $F_{\leq}$ is undecidable (Pierce 1994).

- The kernel $F_{\leq}$, where the bounds are identical (Cardelli and Wenger 1985), is decidable:

$$\frac{\Gamma \vdash A \qquad \Gamma, \alpha \leq A \vdash B_1 \leq B_2}{\Gamma \vdash \forall(\alpha \leq A)\,.\, B_1 \leq \forall(\alpha \leq A)\,.\, B_2}$$

Conservativity

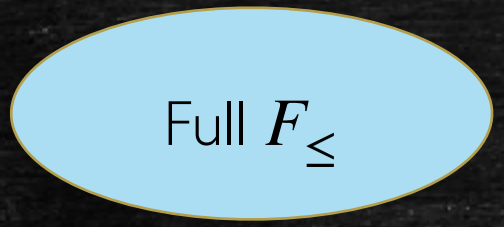
- Conservativity: whether the extended system preserves the behaviour of the original system
- Example: System $F_{\leq}$ is conservative over System F (Cardelli et al. 1994) [Note 1]

$$|\forall X.\, A| = \forall (X \leq \top).\, |A|$$
$$|\Lambda x.\, E| = \Lambda (x \leq \top).\, |E|$$

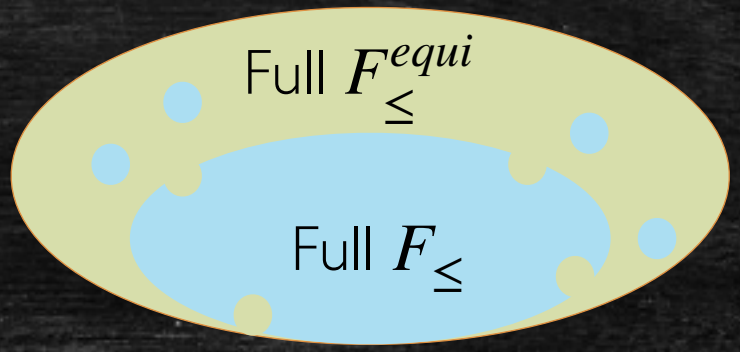
[Note 1]: System $F_{\leq}$ is not conservative over System F with top type.

Conservativity

- (Equi-)recursive types are NOT conservative over (full) $F_{\leq}$ (Ghelli 1993)
- Ghelli's counter-example:



$A \not\leq A'$



$A \leq A'$

$$B \equiv \forall(\alpha \leq \mathsf{T}) . (\forall(\alpha' \leq \alpha) . \alpha \to \mathsf{T}) \to \mathsf{T}$$

$$A' \equiv \forall(\beta \leq B) . (\forall(\beta' \leq \beta) . \beta \to \mathsf{T})$$

$$A \equiv \forall(\beta \leq B) . \beta$$

Comparing different work

	Ghelli 1993	Colazzo and Ghelli 2005	Jeffrey 2001		Abadi, Cardelli, and Viswanathan 1996
Ambient type theory	$F^{equi}_{\leq}$	Kernel $F^{equi}_{\leq}$	$F^{equi}_{\leq}$	Kernel $F^{equi}_{\leq}$	$F^{iso}_{\leq}$ (Amber rules)
Transitivity	✗	✓	✓	✓	Built-in
Decidability		✓	✓*	✓	
Conservativity	✗		✗		
Modularity	✗	✗	✗	✗	

✓* Semi-decidable

Empty entry: not studied

The calculus in this work

- Our calculus kernel $F^{\mu}_{\leq\geq}$ modularly combines:
  - Multi-field records and record types
  - Bounded quantification (kernel style, equivalent bounds)

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

  - Iso-recursive subtyping with nominal unfolding rule (Zhou et al. 2022)

$$\frac{\Gamma, \alpha \vdash [\alpha \mapsto A^{\alpha}]\ A \leq [\alpha \mapsto B^{\alpha}]\ B}{\Gamma \vdash \mu\alpha\,.\,A \leq \mu\alpha\,.\,B}$$

  - Structural folding/unfolding rules
  - Additional lower bounded quantification ($\forall(\alpha \geq A)\,.\,B$) and bottom types

Nominal unfolding rule

- The recently developed nominal unfolding rule by Zhou et al. (TOPLAS'2022):

$$\frac{\Gamma, \alpha \vdash [\alpha \mapsto A^{\alpha}]\ A \leq [\alpha \mapsto B^{\alpha}]\ B}{\Gamma \vdash \mu\alpha\,.\,A \leq \mu\alpha\,.\,B}$$

- Has advantages of
  - Enabling modular proofs
  - Having an easy transitivity proof
  - Not requiring reflexivity built-in
  - Being applicable to non-antisymmetric subtyping relations
- Has the same expressiveness as the iso-recursive Amber rules

Structural folding/unfolding rules

- We also generalize the folding/unfolding rule, and prove the structural rules type sound
- Conventional folding/unfolding rules:

$$\frac{\Gamma \vdash e : [\alpha \mapsto \mu\alpha\,.\,A]\ A}{\Gamma \vdash \text{fold}\ [\mu\alpha\,.\,A]\ e : \mu\alpha\,.\,A} \qquad \frac{\Gamma \vdash e : \mu\alpha\,.\,A}{\Gamma \vdash \text{unfold}\ [\mu\alpha\,.\,A]\ e : [\alpha \mapsto \mu\alpha\,.\,A]\ A}$$

- Structural folding/unfolding rules:

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

The calculus in this work

- Our calculus kernel $F^{\mu}_{\leq\geq}$ modularly combines:
  - Multi-field records and record types
  - Bounded quantification (kernel style, equivalent bounds)

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

  - Iso-recursive subtyping with nominal unfolding rule (Zhou et al. 2022)

$$\frac{\Gamma, \alpha \vdash [\alpha \mapsto A^{\alpha}]\ A \leq [\alpha \mapsto B^{\alpha}]\ B}{\Gamma \vdash \mu\alpha\,.\,A \leq \mu\alpha\,.\,B}$$

  - Structural folding/unfolding rules
  - Additional lower bounded quantification ($\forall(\alpha \geq A)\,.\,B$) and bottom types

- ✓ Type soundness
- ✓ Decidability
- ✓ Conservativity over $F_{\leq}$

Type soundness

- With conventional folding/unfolding rules:

$$\mu\alpha\,.\,A \leq \mu\alpha\,.\,B \quad \Rightarrow \quad [\alpha \mapsto \mu\alpha\,.\,A]\,A \leq [\alpha \mapsto \mu\alpha\,.\,B]\,B$$

- With structural folding/unfolding rules:

$$\mu\alpha\,.\,A \leq \mu\alpha\,.\,C \leq \mu\alpha\,.\,D \leq \mu\alpha\,.\,B \quad \Rightarrow \quad [\alpha \mapsto \mu\alpha\,.\,C]\,A \leq [\alpha \mapsto \mu\alpha\,.\,D]\,B$$

- Preservation: $\Gamma \vdash e : A \wedge e \hookrightarrow e' \Rightarrow \Gamma \vdash e' : A$
- Progress: $\Gamma \vdash e : A \Rightarrow value(e) \vee \exists e', e \hookrightarrow e'$

Decidability — a measure approach

$$size_\psi(\top) = 1$$

$$size_\psi(A \to B) = 1 + size_\psi(A) + size_\psi(B)$$

$$size_\psi(\alpha) = 1 + \psi(\alpha) \text{ if } \alpha \in \psi \text{ else } 1$$

$$size_\psi(\forall(\alpha \leq A)\,.\,B) = \text{let } i = size_\psi(A) \text{ in } 1 + i + size_{\psi,\alpha\mapsto i}(B)$$

$$size_\psi(\mu\alpha\,.\,A) = \text{let } i = size_{\psi,\alpha\mapsto 1}(A) \text{ in } 1 + size_{\psi,\alpha\mapsto i}(B)$$

$$\Psi := \cdot \mid \Psi, \alpha \mapsto i$$

$$eval : \Gamma \hookrightarrow \Psi$$

$$eval(\cdot) = \cdot$$

$$eval(\Gamma', \alpha \leq A) = \text{let } \Psi' = eval(\Gamma') \text{ in } \Psi', \alpha \mapsto size_{\Psi'}(A)$$

If $size_{eval(\Gamma)}(A) + size_{eval(\Gamma)}(B) \leq k$ then $\Gamma \vdash A \leq B$ is either true or not.

Kernel $F^{\mu}_{\leq\geq}$ is conservative over kernel $F_{\leq}$

- Let Γ, A, B and e be in the language of kernel $F_{\leq}$
- Conservativity for subtyping:

$$\text{If } \vdash_{F_{\leq}} \Gamma, \Gamma \vdash_{F_{\leq}} A \text{ and } \Gamma \vdash_{F_{\leq}} B \text{ then } \Gamma \vdash_{F_{\leq}} A \leq B \Leftrightarrow \Gamma \vdash_{F^{\mu}_{\leq\geq}} A \leq B$$

- Conservativity for typing:

$$\text{If } \vdash_{F_{\leq}} \Gamma, \Gamma \vdash_{F_{\leq}} A \text{ and } \Gamma \vdash_{F_{\leq}} e \text{ then } \Gamma \vdash_{F_{\leq}} e : A \Leftrightarrow \Gamma \vdash_{F^{\mu}_{\leq\geq}} e : A$$

Application: Encodings of algebraic datatypes

- To encode algebraic datatypes:
  - Church encodings (1932)
  - Scott encodings (1962)
- With polymorphic typed lambda calculus, recursive types and subtyping, we can extend the Scott encodings to encode datatypes, and obtain
  - Subtyping between datatypes
  - A high degree of composability

Encodings of algebraic datatypes

- We define a datatype Exp_1 with numeric, addition, and subtraction constructors:

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

- The encoding in $F^{\mu}_{\leq\geq}$ of this datatype can be defined as follows:

$$\mathsf{Exp}_1 \triangleq \mu \mathrm{E}\,.\,\forall \mathrm{A}\,.\,\{\mathsf{num} : \mathsf{Int} \to \mathrm{A},\ \mathsf{add} : \mathrm{E} \to \mathrm{E} \to \mathrm{A},\ \mathsf{sub} : \mathrm{E} \to \mathrm{E} \to \mathrm{A}\} \to \mathrm{A}$$

- To construct concrete instances of the datatype:

```
function Num₁ (n: Int) = fold [Exp₁] (Λ A. λe. (e.num n))
function Add₁ (e₁: Exp₁, e₂: Exp₁) = fold [Exp₁] (Λ A. λe. (e.add e₁ e₂))
function Sub₁ (e₁: Exp₁, e₂: Exp₁) = fold [Exp₁] (Λ A. λe. (e.sub e₁ e₂))
```

Encodings of algebraic datatypes

- Encode 1+2:

```
Add₁ (Num₁ 1) (Num₁ 2)
```

- A case analysis-based evaluation function: eval : $Exp_1 \rightarrow Int$

```
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)
}
```

- Evaluate 1+2:

```
eval (Add₁ (Num₁ 1) (Num₁ 2))
```

Subtyping between datatypes

- Consider a larger datatype Exp_2, which extends the Exp_1 datatype with a new constructor *Neg*, for denoting negative numbers

```
data Exp₂ = Num Int | Add Exp₂ Exp₂ | Sub Exp₂ Exp₂ | Neg Exp₂
```

- This datatype is encoded in $F^{\mu}_{\leq\geq}$ as:

$$\text{Exp}_2 \triangleq \mu \text{E} \,.\, \forall \text{A} \,.\, \{\text{num} : \text{Int} \to \text{A},\ \text{add} : \text{E} \to \text{E} \to \text{A},\ \text{sub} : \text{E} \to \text{E} \to \text{A}, \text{neg} : \text{E} \to \text{A}\} \to \text{A}$$

- Note that Exp_2 has more constructors than Exp_1, so it should be safe to coerce Exp_1 expressions into Exp_2 expressions, i.e.

$$\text{Exp}_1 \leq \text{Exp}_2$$

Code Reuse

- If we have already defined a pretty-printer of Exp_2:

```
function print (e: Exp₂) = (unfold [Exp₂] e) [string] {
  num=λn.(int_to_string n),
  add=λe₁ e₂.((print e₁) ++ "+" ++ (print e₂)),
  sub=λe₁ e₂.((print e₁) ++ "-" ++ (print e₂)),
  neg=λe.("-" ++ (print e))
}
```

  then we do not need to write a pretty printer for Exp_1 again

- Since $Exp_1 \leq Exp_2$, the pretty-printer function can also be applied to Exp_1

Challenge: polymorphic constructors

- To reduce code duplication, polymorphic constructors are desired
- With $\mathrm{Exp}_1 \leq \mathrm{Exp}_2$, we expect a polymorphic constructor that works for both datatypes:

We need lower bounded quantification (not available in traditional $F_{\leq}$)

$$\mathrm{Add}_\forall : \forall(E \geq \mathrm{Exp}_1)\,.\, E \to E \to E$$

```
function Add∀ [E ≥ Exp1] (e1: E, e2: E) =
  fold [E] (ΛA. λe. (e.add e1 e2))
```

We need structural folding rules so that we can fold a type to a type variable bounded by a recursive type

Precise typing

- Now assume we wish to implement a desugar function $Exp_2 \rightarrow Exp_1$ by using subtractions $0 - n$ to represent negative numbers $-n$
- In a typical functional language, avoiding code duplication conflicts with precise static typing guarantees

```haskell
data Exp1 =
    Num1 Int | Add1 Exp1 Exp1 | Sub1 Exp1 Exp1

data Exp2 =
    Num2 Int | Add2 Exp2 Exp2 | Sub2 Exp2 Exp2 | Neg Exp2

-- code duplication !
printExp1 :: Exp1 -> String
......
printExp2 :: Exp2 -> String
......

desugar :: Exp2 -> Exp1
......
```

```haskell
data Exp =
  Num Int | Add Exp Exp | Sub Exp Exp | Neg Exp

printExp :: Exp -> String
......

-- no code duplication,
-- but loses typing information
desugar :: Exp -> Exp
......
```

Desugar function in Kernel $F^{\mu}_{\leq\geq}$

```
function desugar (e: Exp₂) = (unfold [Exp₂] e) [Exp₁] {
  num = λ n. Num∀ [Exp₁] n,
  add = λ e₁ e₂. Add∀ [Exp₁] (desugar e₁) (desugar e₂),
  sub = λ e₁ e₂. Sub∀ [Exp₁] (desugar e₁) (desugar e₂),
  neg = λ e. Sub∀ [Exp₁] (Num∀ [Exp₁] 0) (desugar e)
}
```

With polymorphic constructors,
we can avoid code duplication and obtain
static typing guarantees at the same time.

Summary

- Our calculus kernel $F^{\mu}_{\leq\geq}$ modularly combines:
  - Multi-field records and record types
  - Bounded quantification
  - Iso-recursive subtyping with nominal unfolding rule
  - Structural folding/unfolding rules
  - Additional lower bounded quantification and bottom types

✓ Type soundness
✓ Decidability
✓ Conservativity over $F_{\leq}$

Application:
✓ Subtyping between algebraic datatypes

All theorems are formalized in the Coq prover.

Thanks for your listening

Typing for polymorphic constructor

$$\text{TYPING-SFOLD}\ \frac{\dots \qquad \dots}{\vdash \mathsf{Add}_\forall : \forall(\mathsf{E} \geq \mathsf{Exp}_1).\ \mathsf{E} \to \mathsf{E} \to \mathsf{E}}$$

$$\mathsf{E} \geq \mathsf{Exp}_1,\ e_1 : \mathsf{E},\ e_2 : \mathsf{E} \vdash \mathsf{fold}\ [\mathsf{E}]\ (\Lambda A.\ \lambda e.\ (e.\mathsf{add}\ e_1\ e_2)) : \mathsf{E}$$

$$\mathsf{E} \geq \mathsf{Exp}_1,\ e_1 : \mathsf{E},\ e_2 : \mathsf{E} \vdash \Lambda A.\ \lambda e.(e.\mathsf{add}\ e_1\ e_2) : \forall A. \left\{ \begin{array}{l} \mathsf{num} : \mathsf{Int} \to A, \\ \mathsf{add} : \mathsf{E} \to \mathsf{E} \to A, \\ \mathsf{sub} : \mathsf{E} \to \mathsf{E} \to A \end{array} \right\} \to A$$

Without structural rules, the annotated type should not be E, which is a supertype of Exp_1.

Comparing with DOT

	Our calculus $F^{\mu}_{\leq\geq}$	DOT		
		Rompf and Amin 2016	Hu and Lhoták 2020	Mackay et al. 2020
Path-dependent types	no	yes	yes	yes
Bounded quantification	$\alpha \leq A$ or $\alpha \geq A$	$A \leq \alpha \leq B$	$A \leq \alpha \leq B$	$\alpha \leq A$ or $\alpha \geq A$
Recursive types	iso	equi	no	equi
Intersection types	no, but possible[1]	yes	no	limited
Bottom types	yes	yes	yes	yes
Quantifier subtyping	equiv F≤	full F≤	equiv F≤	kernel F≤

[1] Yaoda Zhou, Bruno C. d. S. Oliveira, Andong Fan. *A Calculus with Recursive Types, Record Concatenation and Subtyping.* (APLAS 2022)

Comparing with DOT

	Our calculus $F^{\mu}_{\leq\geq}$	DOT		
		Rompf and Amin 2016	Hu and Lhoták 2020	Mackay et al. 2020
Reflexivity	yes	yes	yes	yes
Transitivity	yes	built-in	missing	missing
Conservativity	yes	no	unknown	unknown
Decidablity	measure based approach	no	stare-at subtyping	material/shape separated approach
Mechanized proofs	yes	yes	yes	yes