Space-Efficient Polymorphic Gradual Typing, Mostly Parametric

Atsushi Igarashi
Kyoto University

Shota Ozaki
Kyoto University

<u>Taro Sekiyama</u>
National Institute of Informatics
& SOKENDAI

Yudai Tanabe
Tokyo Institute of Technology

Gradual Typing (GT) [Siek&Taha'06]

- Enables migration between static and dynamic typing
  - Languages and tools: TypeScript, Typed Racket, Typed Closure, C#, mypy, ...
- Introduces a special type ★ a.k.a. ***the dynamic type***
  - The type check for ★ is skipped at compile time and deferred to run time
- Example: succ

```
let succ (x:★) : ★ = x + 1

succ (42:★)   ⟶ (43:★)  // well-typed
succ (true:★) ⟶ error    // well-typed
```

Gradual Typing (GT) [Siek&Taha'06]

- Enables migration between static and dynamic typing
  - Languages and tools: TypeScript, Typed Racket, Typed Closure, C#, mypy, ...
- Introduces a special type ★ a.k.a. ***the dynamic type***
  - The type check for ★ is skipped at compile time and deferred to run time
- Example: succ

```
let succ (x:int) : int = x + 1

succ 42   ⟶ 43       // well-typed
succ true            // ill-typed
```

Theoretical Research on GT

Parametric polymorphism
[Ahmed et al.'11,'17; Igarashi et al.'17;
Toro et al'19, New et al.'20, Labrada et al.'22]

Objects
[Siek&Taha'07]

Intersection / union types
[Castagna&Lanvin'17]

Effects
[Schwerter et al.'14;
Sekiyama et al.'15, New et al.'23]

Dependent typing
[Lennon-Bertrand et al.'22; Eremondi et al.'22]

Typestate
[Wolff et al.'11]

Security typing
[Fennell&Thiemann'13;
Toro et al.'18; Chen&Siek'24]

Type inference
[Siek&Vachharajani'08;
Garcia&Cimini'15; Miyazaki et al.'19]

etc.

Theoretical Research on GT

Parametric polymorphism
[Ahmed et al.'11,'17; Igarashi et al.'17;
Toro et al'19, New et al.'20, Labrada et al.'22]

Objects
[Siek&Taha'07]

Intersection / union types
[Castagna&Lanvin'17]

Effects
[Schwerter et al.'14;
Sekiyama et al.'15, New et al.'23]

Dependent typing
[Lennon-Bertrand et al.'22; Eremondi et al.'22]

Typestate
[Wolff et al.'11]

Security typing
[Fennell&Thiemann'13;
Toro et al.'18; Chen&Siek'24]

Type inference
[Siek&Vachharajani'08;
Garcia&Cimini'15; Miyazaki et al.'19]

etc.

Polymorphic Gradual Typing (PGT) [Ahmed et al.'11,'17; others]

- Supports **polymorphic types** ∀X.T
- **Enforces parametricity at run time**
  - Run-time errors happen if programs try to break abstraction of polymorphism

```
let id★ : ★       = λx:★. x
let id∀ : ∀X.X→X = id★


id∀ [bool] true    ⟶ true
id∀ [int]  42      ⟶ 42
id∀ [★]    (42:★)  ⟶ (42:★)
```

Polymorphic Gradual Typing (PGT) [Ahmed et al.'11,'17; others]

- Supports **polymorphic types** $\forall X.T$
- **Enforces parametricity at run time**
  - Run-time errors happen if programs try to break abstraction of polymorphism

```
let succ★ : ★      = λx:int. x+1
let id∀   : ∀X.X→X = succ★

id∀ [bool] true    −→ error
id∀ [int]  42      −→ error
id∀ [★]   (42:★)  −→ error
```

Polymorphic Gradual Typing (PGT) [Ahmed et al.'11,'17; others]

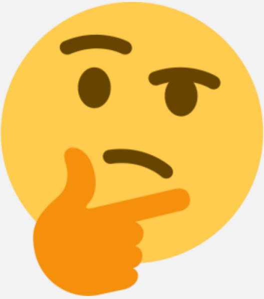
- Supports **polymorphic types** ∀X.T
- **Enforces parametricity at run time**
  - Run-time errors happen if programs try to break abstraction of polymorphism

```
(* doing dynamic analysis on abstract types *)
let id∀ : ∀X.X→X =
  ΛX. λx:X. let x' : ★ = x in
            let y  : ★ = x' + 1 in
            (y : X)

id∀ [bool] true    —→ error
id∀ [int]  42      —→ error
id∀ [★]    (42:★)  —→ error
```

Long-Term Goal

Efficient implementation of PGT



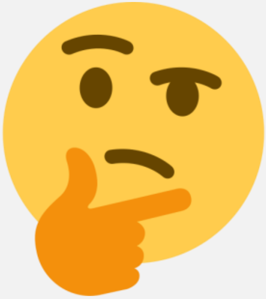
Space-efficient impl. is possible?

What low-level instruction is necessary to compile?

Good performance is achievable?

Long-Term Goal

Efficient implementation of PGT



Space-efficient impl. is possible?

What low-level instruction is necessary to compile?

Good performance is achievable?

Parametricity versus Space-Efficiency

Impossible to implement PGT space-efficiently

(at least under dynamic sealing, the standard method to enforce parametricity)

Is Space-Efficient Polymorphic Gradual Typing Possible?

SHOTA OZAKI, Graduate School of Informatics, Kyoto University, Japan
TARO SEKIYAMA, National Institute of Informatics & SOKENDAI, Japan
ATSUSHI IGARASHI, Graduate School of Informatics, Kyoto University, Japan

Gradual typing, proposed by Siek and Taha, is a way to combine static and dynamic typing in a single programming language. Since its inception, researchers have studied techniques for efficient implementation. In this paper, we study the problem of space-efficient gradual typing in the presence of parametric polymorphism. We develop a polymorphic extension of the coercion calculus, an intermediate language for gradual typing. Then, we show that it cannot be made space-efficient by following the previous approaches, due to subtle interaction with dynamic sealing, a standard technique to ensure parametricity in polymorphic gradual typing.

In Scheme and Functional Programming Workshop 2021

This work shows:

It's *possible* to implement *mostly parametric* PGT space-efficiently

This work shows:

It's *possible* to implement *mostly parametric* PGT space-efficiently

Parametricity is enforced only if
polymorphic values are instantiated with non-★ types

This work shows:

It's *possible* to implement *mostly parametric* PGT space-efficiently

Parametricity is enforced only if
polymorphic values are instantiated with non-★ types

```
let succ★ : ★      = λx:int. x+1
let id∀   : ∀X.X→X = succ★
```

Fully parametric PGT

```
id∀ [int]  42      --→ error
id∀ [★]    (42:★)  --→ error
```

Mostly parametric PGT

```
id∀ [int]  42
id∀ [★]    (42:★)
```

This work shows:

It's *possible* to implement *mostly parametric* PGT space-efficiently

Parametricity is enforced only if polymorphic values are instantiated with non-★ types

```
let succ★ : ★       = λx:int. x+1
let id∀   : ∀X.X→X = succ★
```

Fully parametric PGT

```
id∀ [int]   42      --→ error
id∀ [★]     (42:★) --→ error
```

Mostly parametric PGT

```
id∀ [int]   42      --→ error
id∀ [★]     (42:★) --→ (43:★)
```

Coercion Calculus for GT

An intermediate language where run-time type conversions are made explicit as ***coercions***

Surface language

```
let x:★ = 42 in x + 1
```

Coercion calculus

Coercion Calculus for GT

An intermediate language where run-time type conversions are made explicit as ***coercions***

Surface language

Coercion calculus

```
let x:★ = 42 in x + 1
```

Converted from int to ★

Coercion Calculus for GT

An intermediate language where run-time type conversions are made explicit as ***coercions***

Surface language

Coercion calculus

Converted from ★ to int

```
let x:★ = 42 in x + 1
```

Converted from int to ★

Coercion Calculus for GT

An intermediate language where run-time type conversions are made explicit as ***coercions***

Surface language

```
let x:★ = 42 in x + 1
```

Converted from int to ★ (on 42)

Converted from ★ to int (on x)

Compile ⇒

Coercion calculus

```
let x:★ = 42⟨int!⟩ in x⟨int?⟩ + 1
```

Coercion Calculus for GT

An intermediate language where run-time type conversions are made explicit as ***coercions***

Surface language

Coercion calculus

Converted from ★ to int

```
let x:★ = 42 in x + 1
```

Compile ⇒

```
let x:★ = 42⟨int!⟩ in x⟨int?⟩ + 1
```

Converted from int to ★

Coercion Calculus for GT

An intermediate language where run-time type conversions are made explicit as ***coercions***

Surface language

```
let x:★ = 42 in x + 1
```

Compile

Coercion calculus

let x:★ = $\underline{42}\langle \mathbf{int!} \rangle$ in $\underline{x}\langle \mathbf{int?} \rangle$ + 1

Coercion Calculus for GT

An intermediate language where run-time type conversions are made explicit as ***coercions***

Surface language

```
let x:★ = 42 in x + 1
```

Compile ⇒

Coercion calculus

A value of ★ tagged with **int**

let x:★ = $\underline{42\langle \mathbf{int!} \rangle}$ in $\underline{x\langle \mathbf{int?} \rangle}$ + 1

Coercion Calculus for GT

An intermediate language where run-time type conversions are made explicit as ***coercions***

Surface language

```
let x:★ = 42 in x + 1
```

Compile ⇒

Coercion calculus

A value of ★ tagged with **int**

$\texttt{let}\ x{:}\star = 42\langle \mathbf{int!} \rangle\ \texttt{in}\ x\langle \mathbf{int?} \rangle + 1$

↓

$42\langle \mathbf{int!} \rangle\langle \mathbf{int?} \rangle + 1$

Coercion Calculus for GT

An intermediate language where run-time type conversions are made explicit as ***coercions***

Surface language

```
let x:★ = 42 in x + 1
```

Compile ⇒

Coercion calculus

A value of ★ tagged with **int**

let x:★ = 42⟨**int!**⟩ in x⟨**int?**⟩ + 1

↓

Checks the tag is the same

42⟨**int!**⟩⟨**int?**⟩ + 1

Coercion Calculus for GT

An intermediate language where run-time type conversions are made explicit as ***coercions***

Surface language

```
let x:★ = 42 in x + 1
```

Compile ⇒

Coercion calculus

A value of ★ tagged with **int**

```
let x:★ = 42⟨int!⟩ in x⟨int?⟩ + 1
```

Checks the tag is the same

```
42⟨int!⟩⟨int?⟩ + 1
```

⟶

```
42 + 1
```

Coercion Calculus for GT

An intermediate language where run-time type conversions are made explicit as ***coercions***

Surface language

```
let x:★ = true in x + 1
```

Coercion calculus

Coercion Calculus for GT

An intermediate language where run-time type conversions are made explicit as ***coercions***

Surface language

```
let x:★ = true in x + 1
```

⇒

Coercion calculus

$\texttt{let x:★ = true}\langle \mathbf{bool!} \rangle\ \texttt{in x}\langle \mathbf{int?} \rangle \texttt{ + 1}$

Coercion Calculus for GT

An intermediate language where run-time type conversions are made explicit as ***coercions***

Surface language

```
let x:★ = true in x + 1
```

⇨

Coercion calculus

$\texttt{let}\ x{:}\bigstar = \texttt{true}\langle \mathbf{bool!} \rangle\ \texttt{in}\ x\langle \mathbf{int?} \rangle + 1$

↓

$\texttt{true}\langle \mathbf{bool!} \rangle\langle \mathbf{int?} \rangle + 1$

Coercion Calculus for GT

An intermediate language where run-time type conversions are made explicit as ***coercions***

Surface language

$\texttt{let}\ x : \star = \texttt{true}\ \texttt{in}\ x + 1$

Coercion calculus

$\texttt{let}\ x : \star = \texttt{true}\langle \mathbf{bool!} \rangle\ \texttt{in}\ x\langle \mathbf{int?} \rangle + 1$

$\texttt{true}\langle \mathbf{bool!} \rangle\langle \mathbf{int?} \rangle + 1 \longrightarrow \texttt{error}$

Coercion Calculus for Parametricity Enforcement

- Sealing abstraction by **type names** α generated at type application
  - Intuition: type names can be considered as fresh base types

$$(\Lambda X.M)\ [A] \longrightarrow M[\alpha/X] \quad \text{(where } \alpha \text{ is fresh)}$$

- New coercions for type variables and names

$\langle X! \rangle : X \to \star$ $\quad$ $\langle X? \rangle : \star \to X$ $\quad$ $\langle \alpha! \rangle : A \to \star$ $\quad$ $\langle \alpha? \rangle : \star \to A$

Surface language

```
let id∀ : ∀X.X→X =
  ΛX.λx:X. let x' : ★ = x in
           let y  : ★ = x' + 1 in
           (y : X)
```

Coercion Calculus for Parametricity Enforcement

- Sealing abstraction by **type names** α generated at type application
  - Intuition: type names can be considered as fresh base types

$$(\Lambda X.M)\ [A] \longrightarrow M[\alpha/X]$$

(wh

A is the type argument in generating α

- New coercions for type variables and names

⟨X!⟩ : X→★ ⟨X?⟩ : ★→X ⟨α!⟩ : A→★ ⟨α?⟩ : ★→A

Surface language

```
let id∀ : ∀X.X→X =
  ΛX.λx:X. let x' : ★ = x in
           let y  : ★ = x' + 1 in
           (y : X)
```

Coercion Calculus for Parametricity Enforcement

- Sealing abstraction by **type names** α generated at type application
  - Intuition: type names can be considered as fresh base types

$$(\Lambda X.M)\ [A] \longrightarrow M[\alpha/X] \quad \text{(wh}$$

A is the type argument in generating α

- New coercions for type variables and names

⟨X!⟩ : X→★ ⟨X?⟩ : ★→X ⟨α!⟩ : A→★ ⟨α?⟩ : ★→A

Surface language

```
let id∀ : ∀X.X→X =
  ΛX.λx:X. let x' : ★ = x in
           let y  : ★ = x' + 1 in
           (y : X)
```

⇒

Coercion calculus

```
let id∀ : ∀X.X→X =
  ΛX.λx:X. let x' = x⟨X!⟩ in
           let y  = (x'⟨int?⟩ + 1)⟨int!⟩ in
           y⟨X?⟩
```

Coercion Calculus for Parametricity Enforcement

- Sealing abstraction by **type names** α generated at type application
  - Intuition: type names can be considered as fresh base types

$$(\Lambda X.M)\ [A] \longrightarrow M[\alpha/X] \qquad \text{(wh}$$

> A is the type argument in generating α

- New coercions for type variables and names

$\langle X! \rangle : X \to \star$ $\langle X? \rangle : \star \to X$ $\langle \alpha! \rangle : A \to \star$ $\langle \alpha? \rangle : \star \to A$

Coercion calculus

```
idᵥ [int] 42
```

Coercion calculus

```
let id∀ : ∀X.X→X =
  ΛX.λx:X. let x’ = x⟨X!⟩ in
           let y  = (x’⟨int?⟩ + 1)⟨int!⟩ in
           y⟨X?⟩
```

Coercion Calculus for Parametricity Enforcement

- Sealing abstraction by **type names** α generated at type application
  - Intuition: type names can be considered as fresh base types

$$(\Lambda X.M)\ [A] \longrightarrow M[\alpha/X] \quad \text{(wh}$$

A is the type argument in generating α

- New coercions for type variables and names

$\langle X! \rangle : X \to \star$ | $\langle X? \rangle : \star \to X$ | $\langle \alpha! \rangle : A \to \star$ | $\langle \alpha? \rangle : \star \to A$

Coercion calculus

```
    id∀ [int] 42
--> let x' = 42⟨α!⟩ in …
```

Coercion calculus

```
let id∀ : ∀X.X→X =
  ΛX.λx:X. let x' = x⟨X!⟩ in
           let y  = (x'⟨int?⟩ + 1)⟨int!⟩ in
           y⟨X?⟩
```

Coercion Calculus for Parametricity Enforcement

- Sealing abstraction by **type names** α generated at type application
  - Intuition: type names can be considered as fresh base types

$$(\Lambda X.M)\ [A] \longrightarrow M[\alpha/X] \quad \text{(wh}$$

A is the type argument in generating α

- New coercions for type variables and names

$\langle X! \rangle : X \to \star$ | $\langle X? \rangle : \star \to X$ | $\langle \alpha! \rangle : A \to \star$ | $\langle \alpha? \rangle : \star \to A$

Coercion calculus

```
   id∀ [int] 42
⟶ let x' = 42⟨α!⟩ in …
⟶ let y = (42⟨α!⟩⟨int?⟩ + 1)⟨int!⟩ in …
```

Coercion calculus

```
let id∀ : ∀X.X→X =
  ΛX.λx:X. let x' = x⟨X!⟩ in
           let y  = (x'⟨int?⟩ + 1)⟨int!⟩ in
           y⟨X?⟩
```

Coercion Calculus for Parametricity Enforcement

- Sealing abstraction by **type names** α generated at type application
  - Intuition: type names can be considered as fresh base types

$$(\Lambda X.M)\ [A] \longrightarrow M[\alpha/X] \quad \text{(wh}$$

A is the type argument in generating α

- New coercions for type variables and names

$\langle X! \rangle : X \to \star$ $\langle X? \rangle : \star \to X$ $\langle \alpha! \rangle : A \to \star$ $\langle \alpha? \rangle : \star \to A$

Coercion calculus

```
   id∀ [int] 42
—→ let x' = 42⟨α!⟩ in …
—→ let y = (42⟨α!⟩⟨int?⟩ + 1)⟨int!⟩ in …
```

Coercion calculus

```
let id∀ : ∀X.X→X =
  ΛX.λx:X. let x' = x⟨X!⟩ in
           let y  = (x'⟨int?⟩ + 1)⟨int!⟩ in
           y⟨X?⟩
```

Coercion Calculus for Parametricity Enforcement

- Sealing abstraction by **type names** α generated at type application
  - Intuition: type names can be considered as fresh base types

$$(\Lambda X.M)\ [A] \longrightarrow M[\alpha/X] \quad \text{(wh}$$

A is the type argument in generating α

- New coercions for type variables and names

$\langle X! \rangle : X \to \star$ $\langle X? \rangle : \star \to X$ $\langle \alpha! \rangle : A \to \star$ $\langle \alpha? \rangle : \star \to A$

Coercion calculus

```
    id∀ [int] 42
--> let x' = 42⟨α!⟩ in …
--> let y = (42⟨α!⟩⟨int?⟩ + 1)⟨int!⟩ in …
--> error
```

Coercion calculus

```
let id∀ : ∀X.X→X =
  ΛX.λx:X. let x' = x⟨X!⟩ in
           let y  = (x'⟨int?⟩ + 1)⟨int!⟩ in
           y⟨X?⟩
```

Space-Efficiency [Herman et al.'07,'10]

Any consecutively applied coercions appearing at run time can be compressed into a coercion whose size is bounded statically

➔ **The space consumed by coercions at run time is statically predictable**

More precisely

For any well-typed program M, there exists some $n \in \mathbb{N}$ s.t. for any coercion sequence cs appearing during executing M, cs can be compressed into some coercion c s.t. it preserves the semantics of cs and $\mathrm{size}(c) \leq n$

$$\exists n \in \mathbb{N}.\ \mathtt{M} \longrightarrow \mathtt{M'}\langle c_1 \rangle \cdots \langle c_n \rangle \implies \exists c.\ \langle c \rangle =_{\mathrm{ctx}} \langle c_1 \rangle \cdots \langle c_n \rangle \wedge \mathrm{size}(c) \leq n$$

Impossibility of Space-Efficient, *Fully* Parametric PGT

[Ozaki et al.'21]

Shown by the following facts:

1. There is a program that generates coercion sequences $\langle\alpha_1!\rangle \cdots \langle\alpha_n!\rangle$ for arbitrary n
2. The sequence $\langle\alpha_1!\rangle \cdots \langle\alpha_n!\rangle$ cannot be compressed into a simpler coercion with the same semantics
3. The size of $\langle\alpha_1!\rangle \cdots \langle\alpha_n!\rangle$ is not less than n

Key Observations from The Impossibility

1. The sequence $\langle \alpha_1 ! \rangle \cdots \langle \alpha_n ! \rangle$ is well typed only when, for every α_i,

$$\langle \alpha_i ! \rangle \ : \ \star \to \star$$

   - The program "M$\langle \alpha_1 ! \rangle \langle \alpha_2 ! \rangle$" is ill-typed if $\langle \alpha_1 ! \rangle : \star \to \star$ and $\langle \alpha_2 ! \rangle$: int→★

2. Such α_i is generated by type application `M` [★]

Key Observations from The Impossibility

1. The sequence $\langle\alpha_1 !\rangle \cdots \langle\alpha_n !\rangle$ is well typed only when, for every α_i,

$$\langle\alpha_i !\rangle \;:\; \bigstar \to \bigstar$$

   - ◊ The program "M$\langle\alpha_1 !\rangle\langle\alpha_2 !\rangle$" is ill-typed if $\langle\alpha_1 !\rangle : \bigstar \to \bigstar$ and $\langle\alpha_2 !\rangle : \text{int} \to \bigstar$

2. Such α_i is generated by type application `M [ ★ ]`

➔ The impossibility is due to the type name generation at `M [ ★ ]`

Key Observations from The Impossibility

1. The sequence $\langle\alpha_1!\rangle\cdots\langle\alpha_n!\rangle$ is well typed only when, for every α_i,

$$\langle\alpha_i!\rangle \;:\; \star\rightarrow\star$$

   - The program “$\mathtt{M}\langle\alpha_1!\rangle\langle\alpha_2!\rangle$” is ill-typed if $\langle\alpha_1!\rangle : \star\rightarrow\star$ and $\langle\alpha_2!\rangle : \mathrm{int}\rightarrow\star$

2. Such α_i is generated by type application `M [★]`

➔ The impossibility is due to the type name generation at `M [★]`

➔ **Space-efficiency is possible if `M [★]` generates no type name**

Summary: What We Achieved

Translation preserving typing and semantics

$\lambda C^{\forall}_{mp}$ → $\lambda S^{\forall}_{mp}$

$\lambda C^{\forall}_{mp}$

- Mostly parametric semantics
- Not space-efficient

✅ Type safety
✅ Parametricity **(except for ★)**

$\lambda S^{\forall}_{mp}$

- Mostly parametric semantics
- Coercion sequences are compressed eagerly at run time
- **New coercions for type analysis on whether type arguments are ★**

✅ Type safety
✅ **Space-efficiency**

Conclusion

- Space-efficient PGT is possible if we give up full parametricity
- We show **mostly parametric PGT can be made space-efficient**

Future Work

- Implementation
  - We plan to modify the Grift compiler [Kuhlenschmidt et al.'19] to implement $\lambda S_{mp}^{\forall}$
- Practical evaluation
  - Can the impl be executed efficiently in terms of both time and space?