Presented at
PLDI'24@Copenhagen

Space-Efficient Polymorphic Gradual Typing, Mostly Parametric

Atsushi Igarashi[1], Shota Ozaki[1],
Taro Sekiyama[2], and **Yudai Tanabe[3]**

[1]Kyoto University [2]NII/SOKENDAI **[3]Tokyo Institute of Technology**
(*Institute of Science Tokyo* from October)

JSSST'41
September 11, 2024

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

Languages and tools:
TypeScript, Typed Racket, Typed Closure, C#, Dart, Raku (Perl 6), mypy, …

👍 Enables ***migration between static and dynamic typing*** in a language

✅ well-typed

```
def succ(x):
  return x + 1

class X:
  pass

x = X()
succ(x)
```

Gradual

--> TypeError: unsupported operand type(s) for +: 'X' and 'int'

❌ **ill-typed**

```
def succ(x:int):
  return x + 1

class X:
  pass

x = X()
succ(x)
```

Gradual

```
incompatible
types:
X cannot be
converted to int
    succ(x);
        ^
```

+ Type annotation

How GT[Siek&Taha'06] Works With Unknown Types

Deferred the type check for the dynamic type $\star$ to run time

well-typed

```
def succ(x):
  return x + 1

class X:
  pass

x = X()
succ(x)
```

Error

Gradual

At compile time

The GT checker gives $\star$ to type-unknown variables.

$\star \rightsquigarrow$ int

X $\rightsquigarrow \star$

The GT checker allows ***implicit type conversions*** between $\star$ and any type.

At runtime

All values typed $\star$ are type-checked at runtime.

Theoretical Research on Gradual Typing

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.

Motivation

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

Supports **polymorphic types** ∀X.T
and **enforces parametricity *at run time***

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

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

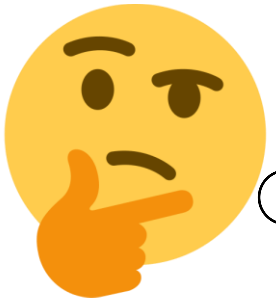
```
let succ⋆ : ⋆       = λx:int. x+1
let id∀   : ∀X.X→X = succ⋆

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

Non-parametric?

Run-time errors happen if programs try to break abstraction of polymorphism

Long-Term Goal: *Efficient PGT Implementation*



Space-efficient impl. is possible?

What low-level instruction is necessary to compile?

Good performance is achievable?

Space-Efficiency vs. *Full Parametricity* in PGT

Impossible to implement PGT space-efficiently[Ozaki'21]

(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.

Our Contribution

“*Mostly*” *parametric* PGT can be made space-efficient

Mostly parametric PGT

$\mathtt{id}_\forall$ [bool] true ⟶ error
$\mathtt{id}_\forall$ [int] 42 ⟶ error
$\mathtt{id}_\forall$ [⋆] (42:⋆) ⟶ 43:⋆ ~~error~~

Key Idea

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

Recap

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

Coercion Calculus: An IR for GT[Henglein'94]

Run-time type conversions are made explicit as ***coercions*** $\langle c \rangle$

Surface	Coercion calculus

Converted from **int** to ⋆

```
let x:⋆ = 42
in x + 1
```

Converted from ⋆ to **int**

Compile ⟹

Coercions can be *automatically* inserted [Siek'06, '21]

A value of ⋆ tagged with **int**

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

$$\to^* 42\langle \textbf{int}! \rangle\langle \textbf{int}?^p \rangle + 1$$
$$\to^* 43$$

✅ **Dynamic type checking**

$$\textbf{int} = \textbf{int}$$

Coercion Calculus: An IR for GT[Henglein'94]

Run-time type conversions are made explicit as *coercions* $\langle c \rangle$

Surface | Coercion calculus

Converted from **bool** to ⋆

```
let x:⋆ = true
in x + 1
```

Converted from ⋆ to **int**

Compile ⇒

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

$$\rightarrow^{*} \mathrm{true}\langle \mathbf{bool!} \rangle\langle \mathbf{int?} \rangle + 1$$

$$\rightarrow^{*} \mathbf{error}$$

✗**Dynamic type checking**

$$\mathbf{bool} \neq \mathbf{int}$$

Parametricity Enforcement

Intuition: ***type names α*** can be considered as fresh base types

Key Idea: ***dynamic sealing***

$$(\Lambda X.\, M)\; A \;\to\; M[X \mapsto \boldsymbol{\alpha}]$$ (where $\boldsymbol{\alpha}$ is fresh)

✗ Nonparametric semantics

✅ Parametric semantics w/ dynamic sealing

$$\Lambda X.\, 42\langle \mathrm{int}!\rangle\langle X?\rangle$$

Apply int

$\to^{*}\; 42\langle \mathrm{int}!\rangle\langle \mathrm{int}?\rangle$

$\to^{*}\; 42$

Apply bool

$\to^{*}\; 42\langle \mathrm{int}!\rangle\langle \mathrm{bool}?\rangle$

$\to^{*}\; \mathrm{error}$

Apply int/bool

$\to^{*}\; 42\langle \mathrm{int}!\rangle\langle \alpha?\rangle$

$\to^{*}\; \mathrm{error}$

Sealing abstraction by type names α

Coercion calculus

Parametricity Enforcement

Key Idea: *dynamic sealing*

$(\Lambda X.M)\ A \ \to M[X \mapsto \boldsymbol{\alpha}]$ (where $\boldsymbol{\alpha}$ is fresh)

Surface | Coercion calculus

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

Compile ⇒

$\mathbf{let}\ id_\forall : \forall X.X \to X = \Lambda X.\lambda x : X.$
$\quad \mathbf{let}\ x' = x\langle X!\rangle\ \mathbf{in}$
$\quad \mathbf{let}\ y\ = (x'\langle \mathbf{int}?\rangle + 1)\langle \mathbf{int}?\rangle\ \mathbf{in}$
$\quad y\langle X?\rangle$

New coercions:

$\langle X!\rangle : X \rightsquigarrow \star$ $\quad \langle \alpha!\rangle : A \rightsquigarrow \star$

$\langle X?\rangle : \star \rightsquigarrow X$ $\quad \langle \alpha?\rangle : \star \rightsquigarrow A$

A is the type argument in generating α

Parametricity Enforcement

Key Idea: ***dynamic sealing***

$(\Lambda X.M)\ A\ \rightarrow M[X \mapsto \boldsymbol{\alpha}]$ (where $\boldsymbol{\alpha}$ is fresh)

Surface | Coercion calculus

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

Compile ⇒

$\mathbf{let}\ id_\forall : \forall X.X \rightarrow X = \Lambda X.\lambda x : X.$

$\quad \mathbf{let}\ x' = x\langle X!\rangle\ \mathbf{in}$

$\quad \mathbf{let}\ y = (x'\langle \mathbf{int}?\rangle + 1)\langle \mathbf{int}?\rangle\ \mathbf{in}$

$\quad y\langle X?\rangle$

$id_\forall\ \mathrm{int}\ 42$

$\rightarrow^* \mathbf{let}\ x' = 42\langle \alpha!\rangle\ \mathbf{in}\ \ldots$

$\rightarrow^* \mathbf{let}\ y = (42\underline{\langle \alpha!\rangle\langle \mathbf{int}?\rangle} + 1)\langle \mathbf{int}?\rangle\ \mathbf{in}\ \ldots$

$\rightarrow^* \mathrm{error}$

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

The space consumed by coercions is statically predictable

$$\forall M. \exists n \in \mathbb{N}. M \to^* M'\langle\overline{c_i}\rangle$$
$$\Rightarrow \exists c. \langle c \rangle =_{\mathrm{ctx}} \langle\overline{c_i}\rangle \land \mathrm{size}(c) \leq n$$

- Any $\langle c_1 \rangle \cdots \langle c_n \rangle$ appearing at run time can be compressed into $\langle c \rangle$ whose size is bounded statically
- They introduce ***eager composition semantics*** to coercion calculus

$$M\langle \mathrm{int!} \rangle\langle \mathrm{int?} \rangle \to M\langle \mathrm{id}_{\mathrm{int}} \rangle$$

Key Idea

[Ozaki et al.'21]

Impossibility of Space-Efficient, *Fully* Parametric PGT

Shown by the following facts:

①There is a well-typed program that generates $\langle\alpha_1!\rangle\cdots\langle\alpha_n!\rangle$ for arbitrary n

Coercion calculus

$$\begin{aligned}&\textbf{let}\ F = \textbf{fix}\ f = \Lambda X.\lambda x : X.\ f \star (x\langle X!\rangle)\\ &\textbf{in}\ F \star (0\langle \mathrm{Int}!\rangle)\\ \to^*\ & F \star (0\langle \mathrm{Int}!\rangle\langle\alpha_1!\rangle\cdots\langle\alpha_n!\rangle)\end{aligned}$$

$$\mathrm{size}(\langle\alpha_1!\rangle\cdots\langle\alpha_n!\rangle) > n$$

③The size of $\langle\alpha_1!\rangle\cdots\langle\alpha_n!\rangle$ is not less than n

$$\nexists c \text{ s.t. } \langle c\rangle =_{\mathrm{ctx}} \langle\alpha_1!\rangle\cdots\langle\alpha_n!\rangle$$

②$\langle\alpha_1!\rangle\cdots\langle\alpha_n!\rangle$ cannot be compressed into a smaller coercion with the same semantics

Key Observations from The Impossibility

The impossibility arises from ***the type name generation at*** $\boldsymbol{M}\ \star$

Erroneous but well-typed coercion sequence

$$\langle \boldsymbol{\alpha_1} ! \rangle \cdots \langle \boldsymbol{\alpha_n} ! \rangle : A \rightsquigarrow \star$$

$\langle \boldsymbol{\alpha_1} ! \rangle \cdots \langle \boldsymbol{\alpha_n} ! \rangle$ is well typed
only when $\Rightarrow \langle \alpha_i ! \rangle : \star \rightsquigarrow \star \quad (2 \leq i \leq n)$

… and such α_i is generated by type application $\boldsymbol{M}\ \star$

Recap

$\langle \alpha ! \rangle : A \rightsquigarrow \star$

A is the type argument in generating α

Key Idea

"Mostly" Parametric Semantics, Informally

Space-efficiency is possible if M $\star$ ***generates no type name***

Dynamic type analysis for type arguments

$$(\Lambda X.\, M)\ A \;\rightarrow\; \begin{cases} M[X \mapsto \star] & (\text{if } A = \star) \\ M[X \mapsto \alpha] & (\text{if } A \neq \star) \end{cases}$$

Dynamic type checking ***does not perform for*** $\star$

$$\langle X! \rangle[X \mapsto \star] = \langle \mathrm{id}_\star \rangle \qquad \langle X? \rangle[X \mapsto \star] = \langle \mathrm{id}_\star \rangle$$

Method

(Informally)

"Mostly" Parametric PGT becomes Space-Efficient

$$\textbf{let}\ F = \textbf{fix}\ f = \Lambda X.\lambda x : X.\ f \star (x\langle X!\rangle)$$
$$\textbf{in}\ \underline{F} \star (0\langle \mathrm{Int}!\rangle)$$
$$\rightarrow^* (\textbf{fix}\ f = \Lambda X.\lambda x : \underline{X}.\ f \star (x\underline{\langle X!\rangle})) \underline{\star} (0\langle \mathrm{Int}!\rangle)$$
$$\rightarrow^* (\lambda x : \star.\ F \star (\underline{x}\langle \mathrm{id}_\star\rangle))(\underline{0\langle \mathrm{Int}!\rangle})$$
$$\rightarrow^* F \star (0\langle \mathrm{Int}!\rangle\langle \mathrm{id}_\star\rangle)$$
$$\rightarrow^* F \star (0\langle \mathrm{Int}!\rangle)$$

Refined type substitution

$$\langle X!\rangle[X := \star] = \langle \mathrm{id}_\star\rangle$$

What We Achieve

New polymorphic coercions for type analysis on whether type arguments are ⋆

$\lambda C^{\forall}_{mp}$ → Translation preserving types and semantics → $\lambda S^{\forall}_{mp}$

$\lambda C^{\forall}_{mp}$:
- ***Mostly parametric*** semantics
- Not space-efficient

$\lambda S^{\forall}_{mp}$:
- ***Mostly parametric*** semantics
- Eager coercion composition

✅ Type safety | ✅ Bisimulation | ✅ Type safety — From [Siek'15]

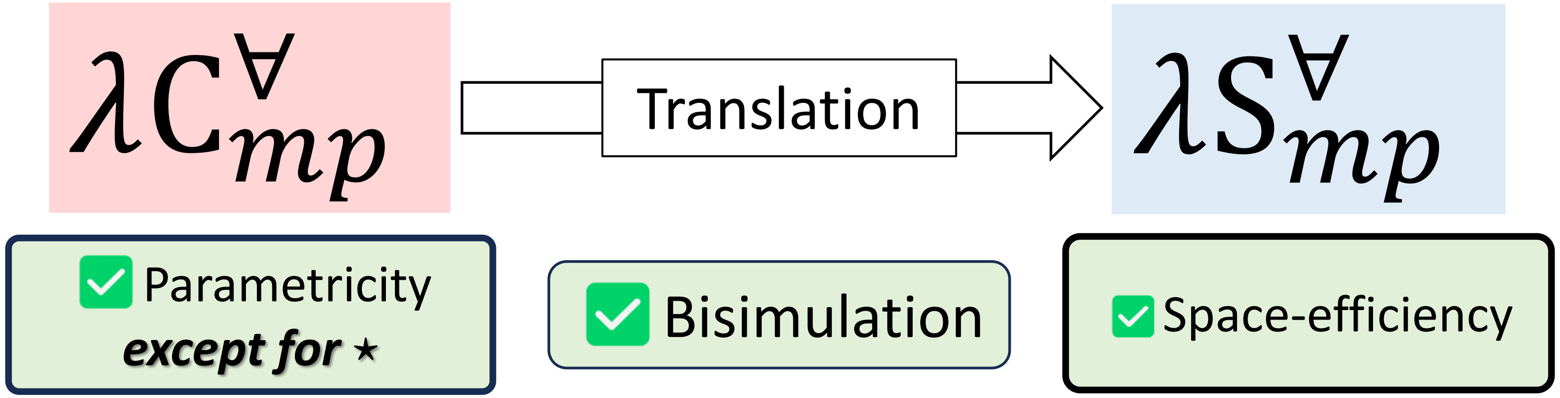
✅ Parametricity ***except for*** ⋆ — Refined from [Ahmed'17]

✅ *Is* Space-efficient — From [Herman'10]

(By handwritten proof, 200+ pages)

Conclusion

"Mostly" parametric PGT can be made space-efficient



Future work:

- Practical evaluation in terms of both *time* and *space*
- Explore optimization opportunities for "*mostly*"-parametricity

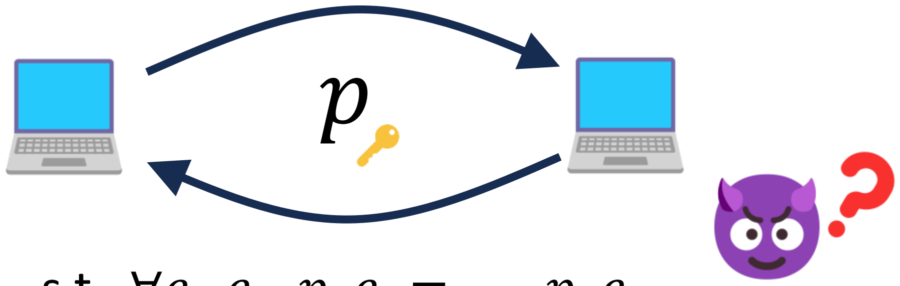
Why Parametricity?[Reynold’86, Wadler’89]

Security

- Non-interference
  [Sumii&Pierce’00,’03, Chen&Siek’22,’24]

Data abstraction

- State encapsulation
  [Launchbury’94]



s.t. $\forall e_i, e_j.\ p\ e_i =_{\mathrm{ctx}}\ p\ e_j$

Program optimization

- Deforestation (a.k.a. fusion)
  [Wadler’90, Gill’93,’96, Marlow’96]

```
  map f (map g xs)
= map (f . g) xs
```

[Siek'12,Herman et al.'07,'10]

Difficulties in *Space-Efficient* GT Implementation

Need to capture the ***context-sensitivity***
to trigger eager-composition

Case. $E = \square \langle c \rangle$

Subcase. M is a coercion application

$$M\langle c_1 \rangle\langle c_2 \rangle \rightarrow M'\langle c_1 \fatsemi c_2 \rangle$$

Case. $E = \square\, M \mid V\, \square \mid \square\, A$

$$\frac{M \rightarrow M'}{E[M] \rightarrow E[M']}$$

Note: No compilers *fully* solve(d) the growing-coercion problem.

[Feltey'18, Kuhlenschmidt'19]

Defunctionalized C^2PS Interpreter

Originally stated in [Herman'10], Formalized in [Tsuda'20]

***"Coercion"*-passing style interpreter**
(& continuation)

```
type val =
    IntV of int
  | Fun of (val -> cont -> val)
  | TFun of (ty -> cont -> val)
  | CAppV of coercion * val
and cont =
    CFId
  | CFAppFun of term * env * cont
  | CFAppArg of val * cont
  | CFTApp of ty * cont
  | CFCApp of coercion * cont
```

Defunctionalize continuation closures
[Reynolds'98, Danvy'01]

```
let rec apply_k k v env =
  match k with
    CFId -> v
  | CFAppFun(exp,env,k') ->
      eval exp env (CFAppArg(v,k'))
  | CFCApp(c1,k') -> …
and eval exp env k =
  match exp with
    Var(r,id) ->
      apply_k k (lookup id env) env
  | …
```

k captures nearest **evaluation context** & **coercion**

Implemented in ~5000LOC, though still WIP