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.

1 INTRODUCTION

1.1 Space-Efficient Gradual Typing

Gradual typing [Siek and Taha 2006] is a methodology to combine static and dynamic typing and allows a single program to involve statically and dynamically typed code. This ability of gradual typing not only brings the benefits of both the two typing disciplines, but also enables smooth migration between fully dynamic and static typing. Emerging gradually typed languages include, for example, Typed Racket [Flatt and PLT 2010], Typed Closure [Bonnaire-Sergeant et al. 2016], Hack [Facebook 2021], and TypeScript [Bierman et al. 2014].

A key ingredient of gradual typing is a special type $\star$, which is called the *dynamic type*. The dynamic type allows "skipping" the static checking: at compile time, any term can be converted to the dynamic type and terms of the dynamic type are optimistically supposed to be convertible to arbitrary types. Instead, the skipped checks are performed at run time by monitoring the conversion. If conversion to an inconsistent type through the dynamic type is detected, the run-time system stops the evaluation of the program and reports an error. For example, let a term M be of the dynamic type $\star$. A static type checker accepts calling it with a Boolean argument $M\,\mathsf{true}$ and applying integer operations $M + 1$. However, M cannot be used as a function nor an integer at run time if it is, say, a Boolean value. Then, the run-time system detects that the conversion of M to a function type or the integer type fails and reports the invalid conversion by raising an exception.

Along with the success of gradual typing in practice, its efficient implementation is gaining attention. Recently it has been exposed that the efficiency of gradually typed programs is dominated by the efficiency of conversions. For example, even in simple gradual typing, a naive implementation of conversions for functions is unexpectedly costly in terms of both time [Takikawa et al. 2016] and space [Herman et al. 2007, 2010].

The space-consuming problem in gradual typing has been recognized first by Herman et al. [2007, 2010]. As an intermediate language for simple gradual typing, they proposed a coercion calculus where a sequence of conversions—which are called *coercions* [Henglein 1994] in coercion calculi—can be normalized into a simpler conversion. They also proved that coercions emerging at run time are collapsed into a coercion of bounded size. This result indicates that simple gradual typing can be implemented in a space-efficient manner, that is, the space overhead of a gradually typed program is increased only by an expected factor, compared with the space consumed by the fully dynamically typed version. Siek et al. [2015] proved correctness of this space-efficient

Authors' addresses: Shota Ozaki, Graduate School of Informatics, Kyoto University, Japan, ozaki@fos.kuis.kyoto-u.ac.jp; Taro Sekiyama, National Institute of Informatics & SOKENDAI, Japan, sekiyama@nii.ac.jp; Atsushi Igarashi, Graduate School of Informatics, Kyoto University, Japan, igarashi@kuis.kyoto-u.ac.jp.

implementation by providing a semantics-preserving translation from an unoptimized, non-space-efficient coercion calculus λC to an optimized, space-efficient coercion calculus λS. The previous work addresses gradual typing with simple types, but support for advanced typing features needs more sophisticated implementation of coercions. Efficient implementation of realistic gradual typing is, therefore, still challenging [Kuhlenschmidt et al. 2019].

1.2 Our Work

This work addresses the space-consuming problem in *polymorphic* gradual typing. As a tool to investigate this problem formally, we introduce a polymorphic (unoptimized) coercion calculus $\lambda C^{\forall}$, which is an extension of the coercion calculus λC [Siek et al. 2015] with polymorphism. A question for this extension is how universal types are represented and checked at run time. Inspired by New et al. [2020]; Toro et al. [2019]; Xie et al. [2018] and motivated by the simplicity, we add the universal type $\forall X.\star$ and type variables X as new type representations to be checked at run time and support new coercions that check consistency for these type representations.

As our calculus $\lambda C^{\forall}$ is designed for polymorphic gradual typing, we have to consider a common issue in it: how parametricity is enforced. Parametricity [Reynolds 1983; Wadler 1989] is a key property of polymorphism, guaranteeing the uniform behavior of polymorphic values and enabling useful reasoning about them. Unfortunately, a naive introduction of polymorphism to gradual typing breaks parametricity because gradual typing can inspect type arguments at run time via conversion. To protect type arguments against run-time inspection, Ahmed et al. [2011] proposed type encapsulation by dynamic sealing. In their calculus, type application generates fresh symbolic names at run time, and type arguments are encapsulated by them. If encapsulated types are inspected in a polymorphic context, the run-time system reports an error. Ahmed et al. [2017] proved that polymorphic gradual typing with dynamic sealing enjoys parametricity. Following Ahmed et al., $\lambda C^{\forall}$ is also equipped with dynamic sealing.

Support for dynamic sealing, however, makes it impossible to implement $\lambda C^{\forall}$ space-efficiently. The key idea of the space-efficient implementation for simple gradual typing is that a sequence of multiple coercions can be simplified into a single, smaller coercion. Dynamic sealing precludes this approach because, briefly speaking, a sequence of coercions with dynamically generated names cannot be simplified furthermore as all the names are necessary to represent the semantics of the coercion sequence. We formally address the problem in a specific coercion calculus $\lambda C^{\forall}$, but for the simplicity of its design, we conjecture that other polymorphic gradually typed languages with dynamic sealing cannot be made space-efficient either; this will be discussed in Section 5.

Below is a summary of our contributions.

- We introduce a type-safe polymorphic coercion calculus $\lambda C^{\forall}$ with run-time type representations for universal types and type variables, coercions for them, and dynamic sealing.
- We show that dynamic sealing allows $\lambda C^{\forall}$ to have a sequence of coercions such that it cannot be normalized into a smaller coercion and the size of the entire sequence cannot be bounded.

The rest of this paper is organized as follows. We review the coercion calculus λC of Siek et al. and how space efficiency is achieved in simple gradual typing in Section 2. The definition and basic properties, including type safety, of $\lambda C^{\forall}$ are presented in Section 3. We state and prove that $\lambda C^{\forall}$ cannot become space-efficient in Section 4. We discuss related work in Section 5 and give remarks on future directions in Section 6.

2 BACKGROUND

This section reviews space efficiency in gradual typing [Herman et al. 2007, 2010; Siek et al. 2015]. We first informally introduce a coercion calculus λC of Siek et al. [2015] and show how statically and dynamically typed code cooperate in λC. We also present that λC allows coercions of unbounded size to appear at run time and then review the approach of Herman et al. [2007, 2010] to normalizing such undesirable coercions and making the coercion calculus space-efficient.

2.1 The Coercion Calculus λC

Gradual typing can control which parts are statically typed and which are dynamically typed by type annotation. To see it, first, let's recall static typechecking. For example, consider a term $\mathsf{let}\ x : \mathsf{Int} = M\ \mathsf{in}\ x + 1$ in static typing. The term M is checked statically to be an integer because it is annotated with the static type Int. Further, the remaining term $x + 1$ is typechecked with the assumption that the variable x is assigned the type Int. Gradual typing can defer this checking process to run time by using the dynamic type $\star$. For example, consider a term

$$\mathsf{let}\ x : \star = M_1\ \mathsf{in}\ \mathsf{let}\ y : \mathsf{Int} = M_2\ \mathsf{in}\ x + y\ .$$

The dynamic type $\star$ means that M_1 can be of an arbitrary type and x can be supposed to be of any type during the typechecking of the remaining term $\mathsf{let}\ y : \mathsf{Int} = M_2\ \mathsf{in}\ x + y$. Because the operation $+$ requires arguments to be integers, we need to check that x is indeed an integer. Gradual typing performs this checking at run time. If M_1 is not an integer—say, a Boolean value true—then an exception will be raised at run time. It is notable that the typechecking in terms of term M_2 and variable y is performed statically because they are annotated with the static type Int and thus, M_2 must be evaluated to an integer (if the evaluation terminates). Gradual typing enables a spectrum between dynamic and static typing by introducing the dynamic type.

The coercion calculus λC is an intermediate language for gradual typing, and it exposes where run-time checking is performed by *coercions*. Coercions support injecting values of static types into the dynamic type and projecting values of the dynamic type to an arbitrary type with run-time checking. Below are coercions in λC (we present only the part necessary to explain the essence of λC):

$$\text{Coercions}\quad c, d \quad ::= \quad G! \mid G?^p \mid \mathsf{id}_A \mid c \to d \mid \cdots$$

Given a coercion c, coercion application $M\,\langle c \rangle$ checks whether the value of a term M can be coerced by c.

The first two coercions, injections $G!$ and projections $G?^p$, represent type conversion using the dynamic type $\star$. These coercions are equipped with *ground types* G and blame labels p. Ground types represent type tags attached to dynamic values (i.e., values of the dynamic type) at run time. They consist of base types, such as Int and Bool, and the function type $\star \to \star$, which signifies that injected values are functions. A blame label p identifies the run-time checking by projection, used for analyzing which part in a source program is the cause of a run-time error [Wadler and Findler 2009]. An injection coercion $G!$ produces a dynamic value tagged with type G. For example, $\mathsf{true}\,\langle \mathsf{Bool}! \rangle$ is a dynamic value tagged with the ground type Bool. A projection coercion $G?^p$ checks the tag of a given dynamic value. If it is the type G, then the coercion returns the underlying value of the dynamic value. In general, λC accepts the reduction $V\,\langle G! \rangle\,\langle G?^p \rangle \longmapsto V$. If the attached tag is not equal to G, then the projection coercion raises an exception $\mathsf{blame}\,p$ with label p that notifies which run-time checking fails. For example, $(\mathsf{true}\,\langle \mathsf{Bool}! \rangle)\,\langle \mathsf{Bool}?^p \rangle$ is evaluated to true, but $(\mathsf{true}\,\langle \mathsf{Bool}! \rangle)\,\langle \mathsf{Int}?^p \rangle$ is evaluated to $\mathsf{blame}\,p$ because $\mathsf{Bool} \neq \mathsf{Int}$. Using these coercions, the example given at the beginning of this section can be translated to the following term in λC:

$$\mathsf{let}\ x : \star = M_1\ \mathsf{in}\ \mathsf{let}\ y : \mathsf{Int} = M_2\ \mathsf{in}\ (x\,\langle \mathsf{Int}?^p \rangle) + y$$

where the variable x of the type $\star$ is coerced to Int. If M_1 injects an integer value (e.g., $1\ \langle \mathsf{Int!} \rangle$), then the projection would succeed and then an integer value would be produced as a final result. However, if M_1 injects a non-integer value (e.g., $\mathsf{true}\ \langle \mathsf{Bool!} \rangle$), then the projection coercion would raise an exception blame p. This translation to λC can be performed automatically [Siek and Taha 2006; Siek et al. 2015]: we can find where coercions are inserted from typing derivations of a gradual type system.

The calculus λC also supports other forms of coercions such as identity coercions and function coercions. An identity coercion id_A expresses an identity function of type $A \to A$; hence, it does nothing computationally. A function coercion $c \to d$ wraps given functions so that arguments and return values are coerced by coercions c and d, respectively. For example, consider a term $V\ \langle \mathsf{Int!} \to \mathsf{Bool?}^p \rangle$. The application to a value V' is reduced as follows:

$$(V\ \langle \mathsf{Int!} \to \mathsf{Bool?}^p \rangle)\ V' \ \longmapsto\ (V\ (V'\ \langle \mathsf{Int!} \rangle))\ \langle \mathsf{Bool?}^p \rangle\ .$$

Here, the argument V' is first coerced to the dynamic type by the argument coercion Int! and then passed to the function V. Once V returns a value of the dynamic type, it is coerced to Bool by $\mathsf{Bool?}^p$. In general, given coercions c from type A_2 to type A_1 and d from type B_1 to type B_2, the function coercion $c \to d$ coerces functions of type $A_1 \to B_1$ to type $A_2 \to B_2$. For example, as Int! is from Int to $\star$ and $\mathsf{Bool?}^p$ is from $\star$ to Bool, the term $V\ \langle \mathsf{Int!} \to \mathsf{Bool?}^p \rangle$ requires the function V to be of $\star \to \star$ and its type is $\mathsf{Int} \to \mathsf{Bool}$. Notice that function coercions are contravariant for arguments while covariant for return values.

2.2 Space Efficiency

Herman et al. [2007, 2010] discovered that a naive implementation of run-time checking may consume an unexpectedly huge space. For example, consider the following mutually recursive functions *even* and *odd* in gradual typing:

$$\begin{array}{lcl} \mathit{even} : \mathsf{Int} \to \star & \stackrel{\text{def}}{=} & \lambda x{:}\mathsf{Int}.\mathsf{if}\ x = 0\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ (\mathit{odd}\ (x - 1)) \\ \mathit{odd} : \mathsf{Int} \to \mathsf{Bool} & \stackrel{\text{def}}{=} & \lambda x{:}\mathsf{Int}.\mathsf{if}\ x = 0\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ (\mathit{even}\ (x - 1)) \end{array}$$

The functions *even* and *odd* return a Boolean value that indicates if an argument is even and odd, respectively. The results of *even* are dynamically typed, while those of *odd* are statically typed. Because these functions are tail-recursive, one might expect that their call only consumes a constant space. However, their translation results $\mathit{even}_{\mathrm{C}}$ and $\mathit{odd}_{\mathrm{C}}$ to λC causes unbounded growth of call stack spaces. The functions $\mathit{even}_{\mathrm{C}}$ and $\mathit{odd}_{\mathrm{C}}$ are given as:

$$\begin{array}{lcl} \mathit{even}_{\mathrm{C}} & \stackrel{\text{def}}{=} & \lambda x{:}\mathsf{Int}.\mathsf{if}\ x = 0\ \mathsf{then}\ (\mathsf{true}\ \langle \mathsf{Bool!} \rangle)\ \mathsf{else}\ ((\mathit{odd}_{\mathrm{C}}\ (x - 1))\ \langle \mathsf{Bool!} \rangle) \\ \mathit{odd}_{\mathrm{C}} & \stackrel{\text{def}}{=} & \lambda x{:}\mathsf{Int}.\mathsf{if}\ x = 0\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ ((\mathit{even}_{\mathrm{C}}\ (x - 1))\ \langle \mathsf{Bool?}^p \rangle) \end{array}$$

Then, for instance, the evaluation of $\mathit{odd}_{\mathrm{C}}\ 4$ proceeds as follows:

$$\begin{array}{ll} & \mathit{odd}_{\mathrm{C}}\ 4 \\ \longmapsto^* & (\mathit{even}_{\mathrm{C}}\ 3)\ \langle \mathsf{Bool?}^p \rangle \\ \longmapsto^* & (\mathit{odd}_{\mathrm{C}}\ 2)\ \langle \mathsf{Bool!} \rangle\ \langle \mathsf{Bool?}^p \rangle \\ \longmapsto^* & (\mathit{even}_{\mathrm{C}}\ 1)\ \langle \mathsf{Bool?}^p \rangle\ \langle \mathsf{Bool!} \rangle\ \langle \mathsf{Bool?}^p \rangle \\ \longmapsto^* & (\mathit{odd}_{\mathrm{C}}\ 0)\ \langle \mathsf{Bool!} \rangle\ \langle \mathsf{Bool?}^p \rangle\ \langle \mathsf{Bool!} \rangle\ \langle \mathsf{Bool?}^p \rangle \\ \longmapsto^* & \mathsf{false}\ \langle \mathsf{Bool!} \rangle\ \langle \mathsf{Bool?}^p \rangle\ \langle \mathsf{Bool!} \rangle\ \langle \mathsf{Bool?}^p \rangle \\ \longmapsto^* & \mathsf{false}\ . \end{array}$$

As the evaluation process indicates, each time $\mathit{even}_{\mathrm{C}}$ and $\mathit{odd}_{\mathrm{C}}$ are called, a new coercion emerges to convert the call result. Such coercions would have to be stored in call stacks because they must be applied after the function call finishes and control gets back. As a result, the evaluation needs a stack space that cannot be bounded by the program size. Even worse, non-terminating programs may consume infinite spaces for storing coercions.

Herman et al. solved this problem by normalizing nested coercions that emerge at run time. For example, we can find that application of coercions $V\,\langle \mathsf{Bool!}\rangle\,\langle \mathsf{Bool?}^p\rangle$ is equivalent to $V\,\langle \mathsf{id}_{\mathsf{Bool}}\rangle$ because applying projection $\langle \mathsf{Bool?}^p\rangle$ to values injected by $\langle \mathsf{Bool!}\rangle$ always succeeds. Herman et al. generalized this idea and defined coercion normalization. Siek et al. [2015] formulated the normalization as a meta-level operation $c \mathbin{;} d$ that collapses the composition of c and d into a simpler coercion. For example, $\mathsf{Bool!} \mathbin{;} \mathsf{Bool?}^p$ returns $\mathsf{id}_{\mathsf{Bool}}$ and $\mathsf{Bool?}^p \mathbin{;} \mathsf{id}_{\mathsf{Bool}}$ returns $\mathsf{Bool?}^p$ (recall that id_A is a no-op).[1] Once we allow the reduction $M\,\langle c\rangle\,\langle d\rangle \longmapsto M\,\langle c \mathbin{;} d\rangle$, the above example can be evaluated as follows:

$$\begin{array}{lll}
& \mathit{odd}_{\mathsf{C}}\ 4 & \\
\longmapsto^* & (\mathit{even}_{\mathsf{C}}\ 3)\,\langle \mathsf{Bool?}^p\rangle & \\
\longmapsto^* & (\mathit{odd}_{\mathsf{C}}\ (3-1))\,\langle \mathsf{Bool!}\rangle\,\langle \mathsf{Bool?}^p\rangle & \\
\longmapsto & (\mathit{odd}_{\mathsf{C}}\ (3-1))\,\langle \mathsf{Bool!} \mathbin{;} \mathsf{Bool?}^p\rangle & = (\mathit{odd}_{\mathsf{C}}\ (3-1))\,\langle \mathsf{id}_{\mathsf{Bool}}\rangle \\
\longmapsto^* & (\mathit{even}_{\mathsf{C}}\ (2-1))\,\langle \mathsf{Bool?}^p\rangle\,\langle \mathsf{id}_{\mathsf{Bool}}\rangle & \\
\longmapsto & (\mathit{even}_{\mathsf{C}}\ (2-1))\,\langle \mathsf{Bool?}^p \mathbin{;} \mathsf{id}_{\mathsf{Bool}}\rangle & = (\mathit{even}_{\mathsf{C}}\ (2-1))\,\langle \mathsf{Bool?}^p\rangle \\
\longmapsto^* & (\mathit{odd}_{\mathsf{C}}\ (1-1))\,\langle \mathsf{Bool!}\rangle\,\langle \mathsf{Bool?}^p\rangle & \\
\longmapsto & (\mathit{odd}_{\mathsf{C}}\ (1-1))\,\langle \mathsf{Bool!} \mathbin{;} \mathsf{Bool?}^p\rangle & = (\mathit{odd}_{\mathsf{C}}\ (1-1))\,\langle \mathsf{id}_{\mathsf{Bool}}\rangle \\
\longmapsto^* & \mathsf{false}\,\langle \mathsf{id}_{\mathsf{Bool}}\rangle & \\
\longmapsto & \mathsf{false}\ . &
\end{array}$$

Because a consecutive application of coercions is immediately normalized, the number of coercions emerging during the evaluation can increase only by a constant factor. Furthermore, Herman et al. proved that the size of a normalized coercion is bounded by the size of the original program before insertion of coercions. By combining the two results, a coercion calculus with the reduction of composing coercions can be proven space-efficient.

Siek et al. [2015] refined the result of Herman et al. with another space-efficient coercion calculus $\lambda\mathsf{S}$ and provided a semantics-preserving translation from $\lambda\mathsf{C}$ to $\lambda\mathsf{S}$ (notice that $\lambda\mathsf{C}$ is not space-efficient because it does not support the composition reduction). A key to providing such translation is that nested coercions can be normalized into a simpler one. However, it seems quite challenging to define coercion normalization satisfying this property for a polymorphic coercion calculus with dynamic sealing, as we will see in Section 4.

3 POLYMORPHIC COERCION CALCULUS $\lambda\mathsf{C}^\forall$

In this section, after presenting an overview of $\lambda\mathsf{C}^\forall$, we develop it formally, and state its basic properties.

3.1 Overview

We extend $\lambda\mathsf{C}$ with parametric polymorphism by introducing type abstractions and applications to terms, and universal types to types, as in System F. Main issues to consider are how ground types, which act as type tags, are extended (or not extended), what kind of coercions are supported for universal types, and how parametricity is enforced.

For the first two, we add the following three things: (1) the type $\forall X.\star$ to ground types to support injection $(\forall X.\star)!$, which is a tagging for type abstractions, and projection $(\forall X.\star)?^p$, which checks whether the value is tagged by $(\forall X.\star)!$; (2) type variables to ground type so that conversion between type variables and $\star$ is possible by $X!$ and $X?^p$; and (3) coercions of the form $\forall X.c$ between universal types such that, if c is a coercion from A to B, then $\forall X.c$ is from $\forall X.A$ to $\forall X.B$. For example, a

[1]Precisely speaking, $\mathsf{Bool!}$ and $\mathsf{Bool?}^p$ are represented by $\mathsf{id}_{\mathsf{Bool}}; \mathsf{Bool!}$ and $\mathsf{Bool?}^p; \mathsf{id}_{\mathsf{Bool}}$, respectively, in Siek et al. [2015].

coercion from type $\forall X.\star \to \star$ to type $\forall X.X \to X$ is written $\forall X.X! \to X?^p$. To coerce the value of type $\forall X.X \to X$ to the dynamic type, we compose three coercions:

(1) $\forall X.X?^p \to X!$, which is from $\forall X.X \to X$ to $\forall X.\star \to \star$;
(2) $\forall X.(\star \to \star)!$, which is from $\forall X.\star \to \star$ to $\forall X.\star$; and
(3) $(\forall X.\star)!$, which is from $\forall X.\star$ to $\star$.

This design is (partially) based on the separation of gradual typing and polymorphism as orthogonal issues—the policy advocated by Xie et al. [2018] and followed by recent calculi [New et al. 2020; Toro et al. 2019]. Thus, there is no coercion from a universal type, say $\forall X.X \to X$, to function types, say $\mathsf{Int} \to \mathsf{Int}$ or $\star \to \star$; such type conversion is possible only by type application.

To enforce parametricity, we follow the standard approach [Ahmed et al. 2011, 2017; Igarashi et al. 2017; Matthews and Ahmed 2008; New et al. 2020; Toro et al. 2019] of using dynamic sealing [Morris 1973; Pierce and Sumii 2000].[2] Instead of usual type-level β-reduction

$$(\Lambda X.V)\,A \longmapsto V[A/X]$$

that uses type substitution $[A/X]$, the correspondence between X and A is recorded in a global store Σ, which decorates the reduction relation, and X in the body V is left as it is:

$$\Sigma \triangleright (\Lambda X.V)\,A \longmapsto (\Sigma, X := A) \triangleright V$$

Inside V, X acts like a fresh base type. Thus, even when X is bound to Int in the store, injection by $X!$ is canceled only by $X?^p$, not by $\mathsf{Int}?^p$. To see why this is important for parametricity, readers are referred to Ahmed et al. [2011, 2017]; Igarashi et al. [2017].

The semantics of coercions between universal types is slightly subtle. First of all, just as coercions $c \to d$ between function types work as wrappers for λ-abstractions, coercions $\forall X.c$ work as wrappers for Λ-abstractions. Thus, for example, the term

$$M = (\Lambda X.V)\,\langle \forall X.X! \to X?^p\rangle$$

is a value, waiting for a type argument to be passed. Here, we assume that the type of $\Lambda X.V$ is $\forall X.\star \to \star$ and thus that of M is $\forall X.X \to X$. If M is applied to a type argument, say Int, and an integer, say 42, $M\,\mathsf{Int}\,42$ is reduced as follows:

$$\begin{aligned}
&\Sigma \triangleright \underbrace{\overbrace{(\Lambda X.V)}^{\forall X.\star\to\star}\langle \forall X.X! \to X?^p\rangle}_{\forall X.X\to X}\ \mathsf{Int}\ 42\\
&\longmapsto (\Sigma, X := \mathsf{Int}) \triangleright (V\,\langle X! \to X?^p\rangle)\,42\\
&\longmapsto (\Sigma, X := \mathsf{Int}) \triangleright (V\,(42\,\langle X!\rangle))\,\langle X?^p\rangle
\end{aligned}$$

The projection $X?^p$ succeeds only if V behaves like $\forall X.X \to X$—that is, V returns the argument $42\,\langle X!\rangle$ as is.

Notice that Λ and $\forall$ are removed (and the store is extended) *in one step*. This is in contrast with the behavior of function coercions, which do not cause β-reduction in one step (as is shown in the second step above). It may appear that the following reduction rule, which generates a new name for a universal coercion, is more reasonable:

$$\Sigma \triangleright (V\,\langle \forall X.c\rangle)\,A \longmapsto (\Sigma, X := A) \triangleright (V\,X)\,\langle c\rangle\ .$$

[2]In the literature, there is some confusion about what sealing/unsealing operations are. While Ahmed et al. [2011] call casts *from dynamically generated type names to the dynamic type* sealing, the subsequent work [New et al. 2020] calls type-directed coercions *from type arguments to dynamically generated type names* sealing. The present work follows the former terminology because our calculus is closer to the one in [Ahmed et al. 2011] (and conversions from type arguments to type names are implicit), but we may say "a type name seals a type" to explain the latter.

According to this rule, the example above would reduce as follows:

$$\begin{array}{rr}
 & \Sigma \triangleright ((\Lambda X.V)\,\langle \forall X.X! \to X?^p\rangle)\,\mathsf{Int}\,42 \\
= & \Sigma \triangleright ((\Lambda X.V)\,\langle \forall Y.Y! \to Y?^p\rangle)\,\mathsf{Int}\,42 \\
\longmapsto & (\Sigma, Y := \mathsf{Int}) \triangleright (((\Lambda X.V)\,Y)\,\langle Y! \to Y?^p\rangle)\,42 \\
\longmapsto & (\Sigma, Y := \mathsf{Int}, X := Y) \triangleright (V\,\langle Y! \to Y?^p\rangle)\,42 \\
\longmapsto & (\Sigma, Y := \mathsf{Int}, X := Y) \triangleright (V\,(42\,\langle Y!\rangle))\,\langle Y?^p\rangle\,.
\end{array}$$

One of the intuitive motivations for our choice is that coercions should not involve computational effects (that is, name generation) other than raising blame. In fact, the second choice would cause counter-intuitive behavior: coercion $\forall X.X?^p \to X!$ from $\forall X.X \to X$ to $\forall X.\star \to \star$ followed by the opposite one $\forall X.X! \to X?^p$ (from $\forall X.\star \to \star$ back to $\forall X.X \to X$) would *not* act as no-op.[3] In other words, the second choice would break the following equivalence $\forall X.(c; d) = (\forall X.c); (\forall X.d)$, which we think should hold. (We conjecture that these coercions are contextually equivalent under our semantics, although it has not been proved.) PolyG$^\nu$ by New et al. [2020] has closely related operational semantics, although this rather big computation step is divided into multiple steps.

3.2 Syntax and Type System

Figure 1 presents the syntax and type system of $\lambda C^\forall$.

Let ι range over base types, which include at least Int and Bool. Let A, B range over types, which are base types ι, the dynamic type $\star$, function types $A \to B$, universal types $\forall X.A$, or type variables X. Unlike recent calculi of polymorphic gradual typing [Ahmed et al. 2017; New et al. 2020; Toro et al. 2019], we do not distinguish type variables and type names for simplicity. Let G, H range over ground types, which represent type tags. Ground types are base types ι, the function type $\star \to \star$, the universal type $\forall X.\star$, or type variables X. Let p, q range over blame labels, which represent program points to identify where run-time checking fails. Let c, d range over coercions, which are identity coercions id_A, injections $G!$, projections $G?^p$, function coercions $c \to d$, sequential compositions $c; d$, or universal coercions $\forall X.c$. A universal coercion $\forall X.c$ binds the type variable X in c. Let M range over terms, which are: constants k including integers, booleans, and first-order primitive functions; variables x; function abstractions $\lambda x\!:\!A.M$; function applications $M\,M$; (recursive) type abstractions $\mathsf{fix}\,x = \Lambda X.V$; type applications $M\,A$; coercion applications $M\,\langle c\rangle$; or run-time errors $\mathsf{blame}\,p$. Let V range over values, which are constants k, function abstractions $\lambda x\!:\!A.M$, type abstractions $\mathsf{fix}\,x = \Lambda X.V$, values with a type tag $V\,\langle G!\rangle$, values wrapped by a function coercion $V\,\langle c \to d\rangle$, or values wrapped by a universal coercion $V\,\langle \forall X.c\rangle$.

Although recursive type abstractions do not really add expressive power to the calculus (as they could be expressed by using the dynamic type and a fixed-point combinator), we introduce them to present some of the examples concisely. $\mathsf{blame}\,p$ stands for a run-time error raised by the failure of a projection $G?^p$.

Let $\mathcal{E}$ range over evaluation contexts, which give standard left-to-right call-by-value semantics. Let Γ range over type environments, which are a sequence of (1) pairs of a variable and its type $x : A$, (2) type variables X, and (3) type bindings $X := A$. We assume that all variables (x in $\Gamma, x : A$) and type variables (X in Γ, X or in $\Gamma, X := A$) in a type environment are pair-wise distinct. Let Σ range over stores, which are special type environments whose elements are only type bindings. We often omit $\emptyset$ at the head of a type environment and write, e.g., $X := \mathsf{Int}, x : X$ for $\emptyset, X := \mathsf{Int}, x : X$.

Terms $\lambda x\!:\!A.M$ and $\mathsf{fix}\,x = \Lambda X.V$ bind variable x in M and V, respectively; $\mathsf{fix}\,x = \Lambda X.V$, $\forall X.A$, and $\forall X.c$ bind type variable X in V, A, and c, respectively. The set of free (type) variables in a

[3]We show a more detailed comparison in Appendix.

Syntax

Base Types	ι	::=	$\mathsf{Int} \mid \mathsf{Bool} \mid \cdots$
Types	A, B	::=	$\iota \mid \star \mid A \to B \mid \forall X.A \mid X$
Ground types	G, H	::=	$\iota \mid \star \to \star \mid \forall X.\star \mid X$
Coercions	c, d	::=	$\mathsf{id}_A \mid G! \mid G?^p \mid c \to d \mid c; d \mid \forall X.c$
Terms	M	::=	$k \mid x \mid \lambda x{:}A.M \mid M\,M \mid \mathsf{fix}\,x = \Lambda X.V \mid M\,A \mid M\,\langle c \rangle \mid \mathsf{blame}\,p$
Values	V	::=	$k \mid \lambda x{:}A.M \mid \mathsf{fix}\,x = \Lambda X.V \mid V\,\langle G! \rangle \mid V\,\langle c \to d \rangle \mid V\,\langle \forall X.c \rangle$
Evaluation Contexts	$\mathcal{E}$	::=	$\square \mid \mathcal{E}[\square\,M] \mid \mathcal{E}[V\,\square] \mid \mathcal{E}[\square\,A] \mid \mathcal{E}[\square\,\langle c \rangle]$
Type environments	Γ	::=	$\emptyset \mid \Gamma, x : A \mid \Gamma, X \mid \Gamma, X := A$
Stores	Σ	::=	$\emptyset \mid \Sigma, X := A$

Type well-formedness $\boxed{\Gamma \vdash A}$

$$\Gamma \vdash \iota \ (\textsc{Tw\_Base}) \qquad \Gamma \vdash \star \ (\textsc{Tw\_Star}) \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B} \ (\textsc{Tw\_Arrow})$$

$$\frac{X \in \Gamma \text{ or } X := A \in \Gamma}{\Gamma \vdash X} \ (\textsc{Tw\_Var}) \qquad \frac{\Gamma, X \vdash A}{\Gamma \vdash \forall X.A} \ (\textsc{Tw\_Poly})$$

Type environment well-formedness $\boxed{\vdash \Gamma}$

$$\vdash \emptyset \ (\textsc{Tew\_Empty}) \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\vdash \Gamma, x : A} \ (\textsc{Tew\_Var})$$

$$\frac{\vdash \Gamma}{\vdash \Gamma, X} \ (\textsc{Tew\_Tyvar}) \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\vdash \Gamma, X := A} \ (\textsc{Tew\_Binding})$$

Coercion typing $\boxed{\Gamma \vdash_C c : A \rightsquigarrow B}$

$$\frac{\vdash \Gamma \qquad \Gamma \vdash A}{\Gamma \vdash_C \mathsf{id}_A : \Gamma(A) \rightsquigarrow \Gamma(A)} \ (\textsc{Ct\_Id}) \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash G}{\Gamma \vdash_C G! : \Gamma(G) \rightsquigarrow \star} \ (\textsc{Ct\_Inj})$$

$$\frac{\vdash \Gamma \qquad \Gamma \vdash G}{\Gamma \vdash_C G?^p : \star \rightsquigarrow \Gamma(G)} \ (\textsc{Ct\_Proj}) \qquad \frac{\Gamma \vdash_C c : A' \rightsquigarrow A \qquad \Gamma \vdash_C d : B \rightsquigarrow B'}{\Gamma \vdash_C c \to d : (A \to B) \rightsquigarrow (A' \to B')} \ (\textsc{Ct\_Arrow})$$

$$\frac{\Gamma \vdash_C c : A \rightsquigarrow B \qquad \Gamma \vdash_C d : B \rightsquigarrow C}{\Gamma \vdash_C c; d : A \rightsquigarrow C} \ (\textsc{Ct\_Seq}) \qquad \frac{\Gamma, X \vdash_C c : A \rightsquigarrow B}{\Gamma \vdash_C \forall X.c : \forall X.A \rightsquigarrow \forall X.B} \ (\textsc{Ct\_Cabs})$$

Term typing $\boxed{\Gamma \vdash_T M : A}$

$$\frac{\vdash \Gamma \qquad ty(k) = A}{\Gamma \vdash_T k : A} \ (\textsc{T\_Const}) \qquad \frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash_T x : A} \ (\textsc{T\_Var}) \qquad \frac{\Gamma, x : \Gamma(A) \vdash_T M : B}{\Gamma \vdash_T \lambda x{:}A.M : \Gamma(A) \to B} \ (\textsc{T\_Abs})$$

$$\frac{\Gamma \vdash_T M_1 : A \to B \qquad \Gamma \vdash_T M_2 : A}{\Gamma \vdash_T M_1\,M_2 : B} \ (\textsc{T\_App}) \qquad \frac{\Gamma, x : \forall X.A, X \vdash_T V : A}{\Gamma \vdash_T \mathsf{fix}\,x = \Lambda X.V : \forall X.A} \ (\textsc{T\_Tyfix})$$

$$\frac{\Gamma \vdash_T M : \forall X.B \qquad \Gamma \vdash A}{\Gamma \vdash_T M\,A : B[\Gamma(A)/X]} \ (\textsc{T\_Tyapp}) \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\Gamma \vdash_T \mathsf{blame}\,p : A} \ (\textsc{T\_Abort})$$

$$\frac{\Gamma \vdash_T M : A \qquad \Gamma \vdash_C c : A \rightsquigarrow B}{\Gamma \vdash_T M\,\langle c \rangle : B} \ (\textsc{T\_Crc})$$

Fig. 1. $\lambda C^{\forall}$: Polymorphic coercion calculus term is defined in a standard manner. We define α-conversion in the standard manner and identify α-equivalent terms. We abbreviate fix $x = \Lambda X.V$ to $\Lambda X.V$ if x does not occur free in V.

We write $[V/x]$ for capture-avoiding substitution of V for x and $[A/X]$ for capture-avoiding type substitution of A for X. We also use (type bindings in) a type environment as a type substitution and write $\Gamma(A)$ for the type defined by:

$$(\emptyset)(A) = A \quad (\Gamma, x : B)(A) = (\Gamma, X)(A) = \Gamma(A) \quad (\Gamma, X := B)(A) = \Gamma(A[B/X])).$$

The type system consists of four judgment forms: type well-formedness $\Gamma \vdash A$, which means that type A is well formed under type environment Γ; type environment well-formedness $\vdash \Gamma$, which means that type environment Γ is well formed; coercion typing $\Gamma \vdash_C c : A \rightsquigarrow B$, which means that c is a well-formed coercion from *source type* A to *target type* B; and term typing $\Gamma \vdash_T M : A$, which means term M is given type A under type environment Γ.

The rules for type well-formedness $\Gamma \vdash A$ and type environment well-formedness $\vdash \Gamma$ are mostly straightforward. Since a type environment contains type variables and type bindings, a type variable X is well formed under Γ if $X \in \Gamma$ or $X := A \in \Gamma$ (Tw_Var). If a type binding $X := A$ or a (typed) variable $x : A$ is added to Γ, the type A has to be well formed under Γ ((Tew_Var) and (Tew_Binding)). For example, $\vdash X := \mathsf{Int}, Y := X \to X$, but not $\vdash Y := X \to X, X := \mathsf{Int}$ because $\emptyset \vdash X \to X$ does not hold.

The rules for coercion typing and term typing are a straightforward adaptation of previous work on coercions [Henglein 1994; Siek et al. 2015] and polymorphic gradual typing [Ahmed et al. 2011; Igarashi et al. 2017]. As in the previous calculi of polymorphic gradual typing, these rules take type bindings in a type environment into account. For example, if a store has $X := A$, then $\lambda x{:}X.x$ should have type $A \to A$, not $X \to X$; otherwise type preservation breaks in $\Sigma \triangleright (\Lambda X.\lambda x{:}X.x)\ A \longmapsto \Sigma, X := A \triangleright \lambda x{:}X.x$. As a rule of thumb, whenever a type annotation appearing in a term is mentioned elsewhere in a judgment, the type environment is applied to get the “real” type by expanding type bindings.

An identity coercion id_A is a coercion from $\Gamma(A)$ to itself. An injection is typed as a coercion from $\Gamma(G)$ to $\star$; conversely, a projection $G?^p$ is from $\star$ to $\Gamma(G)$. A function coercion $c \to d$ is a coercion from $A \to B$ to a function type $A' \to B'$ if c coerces A' to A and d coerces B to B'. A composition coercion $c; d$, which applies c and d in this order, is from A to C if c coerces A to B and d coerces B to C. Finally, a universal coercion $\forall X.c$ is from a universal type $\forall X.A$ to a universal type $\forall X.B$ if c coerces A to B under a type environment augmented with X. In (T_Const), ty is a (meta-level) function which maps a constant k to a first-order type of the form $\iota_1 \to \iota_2 \to \cdots \to \iota_n (n \geq 1)$.

3.3 Operational Semantics

Figure 2 defines the operational semantics of $\lambda C^{\forall}$. Reduction rules are about basic computation steps and evaluation rules are about computation of subterms. As we already explained in Section 3.1, the reduction and evaluation relations involve stores and are written $\Sigma \triangleright M \longrightarrow \Sigma' \triangleright M'$ and $\Sigma \triangleright M \longmapsto \Sigma' \triangleright M'$, respectively. In the case that the stores are the same on both sides, that is $\Sigma = \Sigma'$, we omit them: For example, the rule (R_Beta) is understood as $\Sigma \triangleright (\lambda x{:}A.M)\ V \longrightarrow \Sigma \triangleright M[V/x]$. (R_Delta) reduces an application of a primitive function. Here, δ is a meta-level partial function from two constants to another and is supposed to preserve types in the sense that $ty(k_1) = \iota \to A$ and $ty(k_2) = \iota$ imply $ty(\delta(k_1, k_2)) = A$. (R_Beta) is standard β-reduction. (R_Id) means that an identity coercion is an identity function. (R_Wrap) reduces a function application where the function is wrapped by a function coercion $c \to d$. The coercion c is applied to the argument and the coercion d is applied to the return value. (R_Collapse) and (R_Conflict) represent type tag checking. (R_Collapse) is the case that V passes the check because the ground types in the injection and projection are the same; they are removed after reduction. (R_Conflict) is the case that V does

Reduction $\boxed{\Sigma \triangleright M \longrightarrow \Sigma' \triangleright M'}$

$$\begin{array}{rcllr}
k_1\, k_2 & \longrightarrow & \delta(k_1, k_2) & & (\text{R\_Delta}) \\
(\lambda x : A.M)\, V & \longrightarrow & M[V/x] & & (\text{R\_Beta}) \\
V\, \langle \mathsf{id}_A \rangle & \longrightarrow & V & & (\text{R\_Id}) \\
(V\, \langle c \rightarrow d \rangle)\, V' & \longrightarrow & (V\, (V'\, \langle c \rangle))\, \langle d \rangle & & (\text{R\_Wrap}) \\
V\, \langle G! \rangle\, \langle G?^p \rangle & \longrightarrow & V & & (\text{R\_Collapse}) \\
V\, \langle G! \rangle\, \langle H?^p \rangle & \longrightarrow & \mathsf{blame}\, p & \text{if } G \neq H & (\text{R\_Conflict}) \\
V\, \langle c; d \rangle & \longrightarrow & V\, \langle c \rangle\, \langle d \rangle & & (\text{R\_Split}) \\
\Sigma \triangleright ((\mathsf{fix}\, x = \Lambda X.V)\, \overrightarrow{\langle \forall X.c \rangle})\, A & \longrightarrow & \Sigma, X := A \triangleright V[\mathsf{fix}\, x = \Lambda X.V/x]\, \overrightarrow{\langle c \rangle} & & (\text{R\_Tybeta})
\end{array}$$

Evaluation $\boxed{\Sigma \triangleright M \longmapsto \Sigma' \triangleright M'}$

$$\frac{\mathcal{E} \neq \square}{\mathcal{E}[\mathsf{blame}\, p] \longmapsto \mathsf{blame}\, p}\ (\text{E\_Blame}) \qquad \frac{\Sigma \triangleright M \longrightarrow \Sigma' \triangleright M'}{\Sigma \triangleright \mathcal{E}[M] \longmapsto \Sigma' \triangleright \mathcal{E}[M']}\ (\text{E\_Evctx})$$

Fig. 2. Operational semantics

not pass because the ground types in the injection and projection are different; the term is reduced to blame p. (R_Split) splits a composition coercion into two consecutive coercion applications. In (R_Tybeta), $\overrightarrow{\langle c \rangle}$ stands for a sequence of 0 or more coercion applications, i.e., $M\, \overrightarrow{\langle c \rangle} = M\, \langle c_1 \rangle \cdots \langle c_n \rangle$. (R_Tybeta) reduces a type application, in which the type abstraction may be wrapped by universal coercions. The reduction generates a fresh type name, extends the store Σ by $X := A$, instead of applying a type substitution, and removes *all* binders ΛX and $\forall X$ at once, which means that the type variables X bound by them are considered the same. Moreover, since the variable x bound by the recursive type abstraction stands for the type abstraction itself, $\mathsf{fix}\, x = \Lambda X.V$ is substituted for x. (E_Blame) reduces a term containing a run-time type error blame p; the current evaluation context is discarded and the whole term is reduced to blame p. (E_Evctx) is standard, which allows a subterm to be reduced.

3.4 Properties

We state two basic properties of $\lambda C^{\forall}$: determinacy of evaluation (Theorem 1) and type safety (Theorem 4). Determinacy of evaluation is proved via determinacy of reduction. (We extend α-equivalence to $\Sigma \triangleright M$ in a straightforward manner—by considering type variables defined in Σ bound in M. For example, $X := \mathsf{Int} \triangleright 42\, \langle X! \rangle$ and $Y := \mathsf{Int} \triangleright 42\, \langle Y! \rangle$ are α-equivalent and, thus, identified.)

Lemma 1 (Determinacy of reduction). *If $\Sigma \triangleright M \longrightarrow \Sigma_1 \triangleright M_1$ and $\Sigma \triangleright M \longrightarrow \Sigma_2 \triangleright M_2$, then $\Sigma_1 = \Sigma_2$ and $M_1 = M_2$.*

Proof. By straightforward case analysis on $\Sigma \triangleright M \longrightarrow \Sigma_1 \triangleright M_1$. □

Theorem 1 (Determinacy of evaluation). *If $\Sigma \triangleright M \longmapsto \Sigma_1 \triangleright M_1$ and $\Sigma \triangleright M \longmapsto \Sigma_2 \triangleright M_2$, then $\Sigma_1 = \Sigma_2$ and $M_1 = M_2$.*

Proof. By case analysis on $\Sigma \triangleright M \longmapsto \Sigma_1 \triangleright M_1$. In the case of (E_Evctx), we use Lemma 1. □

Type safety follows from progress and preservation [Wright and Felleisen 1994]. We write $\longmapsto^*$ for the reflexive transitive closure of $\longmapsto$, and $\Sigma \triangleright M \Uparrow$ if there is an infinite evaluation sequence starting from $\Sigma \triangleright M$.

Theorem 2 (Progress). *If $\Sigma \vdash_T M : A$, then one of the followings holds:*

- *$M = V$ for some V,*
- *$M = \mathsf{blame}\, p$ for some p, or*
- *$\Sigma \triangleright M \longmapsto \Sigma' \triangleright M'$ for some Σ' and M'.*

Proof. By induction on the derivation of $\Sigma \vdash_T M : A$. □

Lemma 2 (Preservation for reduction). *If $\Sigma \vdash_T M : A$ and $\Sigma \triangleright M \longrightarrow \Sigma' \triangleright M'$, then $\Sigma' \vdash_T M' : A$.*

Proof. By case analysis on $\Sigma \triangleright M \longrightarrow \Sigma' \triangleright M'$. □

Theorem 3 (Preservation for evaluation). *If $\Sigma \vdash_T M : A$ and $\Sigma \triangleright M \longmapsto \Sigma' \triangleright M'$, then $\Sigma' \vdash_T M' : A$.*

Proof. By case analysis on $\Sigma \triangleright M \longmapsto \Sigma' \triangleright M'$. The case for (E_Evctx) is proved by induction on $\mathcal{E}$. Use Lemma 2 if $\mathcal{E}$ is $\square$. □

Theorem 4 (Type safety). *If $\Sigma \vdash_T M : A$, then one of the followings holds:*

- *$\Sigma \triangleright M \longmapsto^* \Sigma' \triangleright V$ for some store Σ' and value V,*
- *$\Sigma \triangleright M \longmapsto^* \Sigma' \triangleright \mathsf{blame}\, p$ for some store Σ' and blame label p, or*
- *$\Sigma \triangleright M \Uparrow$.*

Proof. By Theorems 2 and 3. □

4 $\lambda C^{\forall}$ CANNOT BE MADE SPACE-EFFICIENT BY COERCION NORMALIZATION

As mentioned at the end of Section 2, a key property to achieve space-efficiency is that nested coercions can be normalized into a simpler one. This property (adapted from Theorem 6 of Herman et al. [2010]) can be formally stated as follows:

Proposition 5 (Size of coercions is bounded). *For any closed well-typed term M, there exists a natural number n such that, for any term M', sequence of coercion applications $\langle c_1 \rangle \cdots \langle c_n \rangle$ and store Σ, if*

- *$\emptyset \triangleright M \longmapsto^* \Sigma \triangleright M'$ and*
- *M' contains $\langle c_1 \rangle \cdots \langle c_n \rangle$,*

then there exists a coercion c such that c is contextually equivalent to $c_1; \cdots ; c_n$ and $\mathsf{size}(c) \leq n$.

This proposition means that (1) any coercion sequence $\langle c_1 \rangle \cdots \langle c_n \rangle$ that appears during the execution of a program M can be normalized into a single coercion c in some way, and (2) c behaves as $c_1; \cdots ; c_n$ and the size of c is bounded by a natural number n depending on only (the type derivation of) the program M. By normalizing all coercions at every evaluation step, the fraction of the program state occupied by coercions can be bounded, which is what "space-efficient" means in the literature. (To give a concrete space-efficient calculus, we would have to develop an effective normalization procedure and an operational semantics that eagerly composes coercion sequences—just as Bañados Schwerter et al. [2021]; Herman et al. [2007, 2010]; Siek et al. [2015] did.)

However, Proposition 5 *does not hold*, that is, it is impossible to derive a space-efficient variant of $\lambda C^{\forall}$ (by coercion normalization). This impossibility stems from the fact that a sequence $X_1!; \cdots ; X_n!$ of injection coercions[4] cannot be made smaller and there is a program that creates a growing sequence of such injection coercions. In the rest of this section, we disprove Proposition 5 as Theorem 6.

[4]This coercion is well typed if $X_i := \star \in \Sigma$ for any i.

We will define contextual equivalence of coercions to formally discuss whether a coercion can be replaced by a smaller one without affecting the semantics of programs. However, as in the example of $even_C/odd_C$ in Section 2, what grows large is not a coercion but a sequence of coercion applications. Therefore, we define contextual equivalence over sequences of coercion applications.

First, we define coercion sequences and coercion sequence typing.

Definition 1 (Coercion sequences). *Coercion sequences cs are defined as follows:*

$$cs \quad ::= \quad \langle c \rangle ; cs \mid \langle c \rangle$$

Definition 2 (Coercion sequence typing). *Coercion sequence typing rules are defined as follows:*

$$\frac{\Gamma \vdash_C c : A \rightsquigarrow B}{\Gamma \vdash_{CS} \langle c \rangle : A \rightsquigarrow B}\ (\text{Cs\_One}) \qquad \frac{\Gamma \vdash_C c : A \rightsquigarrow B \qquad \Gamma \vdash_{CS} cs : B \rightsquigarrow C}{\Gamma \vdash_{CS} \langle c \rangle ; cs : A \rightsquigarrow C}\ (\text{Cs\_More})$$

Coercion sequences are sequences of $\langle c \rangle$'s connected by semicolons. $\langle c \rangle$ coerces A to B if c coerces A to B. $\langle c \rangle; cs$ coerces A to C if c coerces A to B and cs coerces B to C.

Next, we define coercion contexts and contextual equivalence for coercion sequences to discuss whether replacement of them affects program semantics.

Definition 3 (Contexts). *Contexts $\mathcal{C}$ are defined as follows:*

$$\mathcal{C} \quad ::= \quad \square \mid \lambda x : A.\mathcal{C} \mid \mathcal{C}\, M \mid M\, \mathcal{C} \mid \mathsf{fix}\, x = \Lambda X.\mathcal{C} \mid \mathcal{C}\, A \mid \mathcal{C}\, \langle c \rangle$$

Definition 4 (Coercion contexts). *Coercion contexts are defined as pairs of term M and context $\mathcal{C}$. Then, $(M, \mathcal{C})[cs]$ is defined as follows:*

$$(M, \mathcal{C})[\langle c \rangle ; cs] = (M\, \langle c \rangle, \mathcal{C})[cs]$$
$$(M, \mathcal{C})[\langle c \rangle] = \mathcal{C}[M\, \langle c \rangle]$$

As usual, a context is a term with a single hole and $\mathcal{C}[M]$ represents a term obtained by textually substituting M for the hole in $\mathcal{C}$. If the hole is surrounded by $\mathsf{fix}\, x = \Lambda X.\square$, $\mathcal{C}[M]$ is well defined only if M is a value V. A coercion context is a pair of a term and a context, $(M, \mathcal{C})$. Intuitively, $(M, \mathcal{C})[\langle c_1 \rangle; \cdots ; \langle c_n \rangle]$ represents the term $\mathcal{C}[M\, \langle c_1 \rangle \cdots \langle c_n \rangle]$.

Definition 5 (Contextual equivalence for coercion sequences). *Two coercion sequences cs_1, cs_2 are* contextually equivalent, *written $cs_1 \overset{\text{ctx}}{=} cs_2$, if, for any coercion context $(M, \mathcal{C})$, type A, and store Σ, $\Sigma \vdash_T (M, \mathcal{C})[cs_1] : A$ and $\Sigma \vdash_T (M, \mathcal{C})[cs_2] : A$ imply one of the followings:*

- $\Sigma \triangleright (M, \mathcal{C})[cs_1] \longmapsto^* \Sigma_1 \triangleright V_1$ *and* $\Sigma \triangleright (M, \mathcal{C})[cs_2] \longmapsto^* \Sigma_2 \triangleright V_2$, *for some values V_1 and V_2, and stores Σ_1 and Σ_2,*
- $\Sigma \triangleright (M, \mathcal{C})[cs_1] \longmapsto^* \Sigma_1 \triangleright \mathsf{blame}\, p$ *and* $\Sigma \triangleright (M, \mathcal{C})[cs_2] \longmapsto^* \Sigma_2 \triangleright \mathsf{blame}\, p$, *for some blame label p and stores Σ_1 and Σ_2, or*
- $\Sigma \triangleright (M, \mathcal{C})[cs_1] \Uparrow$ *and* $\Sigma \triangleright (M, \mathcal{C})[cs_2] \Uparrow$.

Contextual equivalence, $cs_1 \overset{\text{ctx}}{=} cs_2$, means that for any coercion context $(M, \mathcal{C})$, if $(M, \mathcal{C})[cs_1]$ and $(M, \mathcal{C})[cs_2]$ have the same type, both result in a value, raise the same blame, or diverge.

Next, we define the sizes of a coercion and a coercion sequence in order to discuss space consumption.

Definition 6 (Size of coercions). *The size* $\mathrm{size}(c)$ *of a coercion* c *is defined as follows:*

$$\begin{aligned} \mathrm{size}(\mathsf{id}_A) &= 1 \\ \mathrm{size}(G!) &= 1 \\ \mathrm{size}(G?^p) &= 1 \\ \mathrm{size}(c \to d) &= \mathrm{size}(c) + \mathrm{size}(d) + 1 \\ \mathrm{size}(c; d) &= \mathrm{size}(c) + \mathrm{size}(d) + 1 \\ \mathrm{size}(\forall X.c) &= \mathrm{size}(c) + 1 \end{aligned}$$

Definition 7 (Size of coercion sequences). *The size* $\mathrm{size}(cs)$ *of a coercion sequence* cs *is defined as follows:*

$$\begin{aligned} \mathrm{size}(\langle c\rangle; cs) &= \mathrm{size}(c) + \mathrm{size}(cs) + 1 \\ \mathrm{size}(\langle c\rangle) &= \mathrm{size}(c) \end{aligned}$$

These sizes of a coercion and a coercion sequence are defined as the number of constructors included in them. The size of a coercion sequence is defined so that $\langle c\rangle; \langle d\rangle$ and $c; d$ has the same size.

Finally, we prove the main theorem: There is a term that generates a coercion sequence of unbounded size, which cannot shrink to a single coercion without modifying the program behavior. Let $CS(M)$ be the set of coercion sequences in M.

Theorem 6 (Not space-efficient). *There exists a closed well-typed term* M *such that, for any natural number* n*, there exist term* M' *and store* Σ *such that*

1. $\emptyset \triangleright M \longmapsto^* \Sigma \triangleright M'$*, and*
2. *there exist a coercion sequence* $cs \in CS(M')$*, type* B_1*, and type* B_2 *such that*
   - (a) $\Sigma \vdash_{CS} cs : B_1 \rightsquigarrow B_2$*,*
   - (b) $\mathrm{size}(cs) > n$*, and*
   - (c) *there does not exist a coercion* c *such that* $\Sigma \vdash_C c : B_1 \rightsquigarrow B_2$*,* $cs \overset{\mathrm{ctx}}{=} \langle c\rangle$*, and* $\mathrm{size}(\langle c\rangle) < \mathrm{size}(cs)$*.*

Proof. Let $M = (\mathsf{fix} f = \Lambda X.\lambda x{:}X.f \star (x\,\langle X!\rangle)) \star (0\,\langle \mathsf{Int}!\rangle)$. Fix n. Let

$$\begin{aligned} M' &= (\mathsf{fix} f = \Lambda X.\lambda x{:}X.f \star (x\,\langle X!\rangle)) \star (0\,\langle \mathsf{Int}!\rangle\,\langle X_1!\rangle \cdots \langle X_n!\rangle), \text{ and} \\ \Sigma &= \emptyset, X_1 := \star, \ldots, X_n := \star \end{aligned}$$

where $X_1, \ldots, X_n$ are distinct type variables. (1) can be proved easily, so we omit its proof. We prove (2). Let

$$\begin{aligned} cs &= \langle \mathsf{Int}!\rangle; \langle X_1!\rangle; \cdots ; \langle X_n!\rangle, \\ B_1 &= \mathsf{Int}, \text{ and} \\ B_2 &= \star. \end{aligned}$$

(a) and (b) can be proved easily, so we omit proofs about them. We prove (c), which is equivalent to "for any coercion c, if $\Sigma \vdash_C c : B_1 \rightsquigarrow B_2$ and $cs \overset{\mathrm{ctx}}{=} \langle c\rangle$, then $\mathrm{size}(\langle c\rangle) \geq \mathrm{size}(cs)$". Fix c and suppose $\Sigma \vdash_C c : B_1 \rightsquigarrow B_2$ and $cs \overset{\mathrm{ctx}}{=} \langle c\rangle$. Let $C = \square\,\langle X_n?^{p_n}\rangle \cdots \langle X_1?^{p_1}\rangle\langle \mathsf{Int}?^p\rangle$. Then, we have

$$\begin{aligned} (0, C)[\langle \mathsf{Int}!\rangle; \langle X_1!\rangle; \cdots ; \langle X_n!\rangle] &= 0\,\langle \mathsf{Int}!\rangle\,\langle X_1!\rangle \cdots \langle X_n!\rangle\,\langle X_n?^{p_n}\rangle \cdots \langle X_1?^{p_1}\rangle\,\langle \mathsf{Int}?^q\rangle \text{ and} \\ (0, C)[\langle c\rangle] &= 0\,\langle c\rangle\,\langle X_n?^{p_n}\rangle \cdots \langle X_1?^{p_1}\rangle\,\langle \mathsf{Int}?^q\rangle. \end{aligned}$$

Both

$$\Sigma \vdash_T 0\,\langle \mathsf{Int}!\rangle\,\langle X_1!\rangle \cdots \langle X_n!\rangle\,\langle X_n?^{p_n}\rangle \cdots \langle X_1?^{p_1}\rangle\,\langle \mathsf{Int}?^q\rangle : \mathsf{Int}$$

and

$$\Sigma \vdash_T 0 \langle c \rangle \langle X_n ?^{p_n} \rangle \cdots \langle X_1 ?^{p_1} \rangle \langle \mathsf{Int}?^q \rangle : \mathsf{Int}$$

are easily proved because $\Sigma \vdash_C c : \mathsf{Int} \rightsquigarrow \star$. Therefore, by $\langle \mathsf{Int}! \rangle ; \langle X_1! \rangle ; \cdots ; \langle X_n! \rangle \overset{\text{ctx}}{=} \langle c \rangle$, terms $0 \langle \mathsf{Int}! \rangle \langle X_1! \rangle \cdots \langle X_n! \rangle \langle X_n ?^{p_n} \rangle \cdots \langle X_1 ?^{p_1} \rangle \langle \mathsf{Int}?^q \rangle$ and $0 \langle c \rangle \langle X_n ?^{p_n} \rangle \cdots \langle X_1 ?^{p_1} \rangle \langle \mathsf{Int}?^q \rangle$ have the same result (a value, a blame, or divergence). $\Sigma \triangleright 0 \langle \mathsf{Int}! \rangle \langle X_1! \rangle \cdots \langle X_n! \rangle \langle X_n ?^{p_n} \rangle \cdots \langle X_1 ?^{p_1} \rangle \langle \mathsf{Int}?^q \rangle \longmapsto^* \Sigma \triangleright 0$ is clear, so $0 \langle c \rangle \langle X_n ?^{p_n} \rangle \cdots \langle X_1 ?^{p_1} \rangle \langle \mathsf{Int}?^q \rangle$ is also evaluated to a value, i.e., there exist store Σ_1 and value V_1 such that $\Sigma \triangleright 0 \langle c \rangle \langle X_n ?^{p_n} \rangle \cdots \langle X_1 ?^{p_1} \rangle \langle \mathsf{Int}?^q \rangle \longmapsto^* \Sigma_1 \triangleright V_1$. Thus, subterm $0 \langle c \rangle$ is also evaluated to a value, i.e., there exist store Σ_2 and value V_2 such that $\Sigma \triangleright 0 \langle c \rangle \longmapsto^* \Sigma_2 \triangleright V_2$. Hence by Theorem 1,

$$\Sigma \triangleright 0 \langle c \rangle \langle X_n ?^{p_n} \rangle \cdots \langle X_1 ?^{p_1} \rangle \langle \mathsf{Int}?^q \rangle \longmapsto^* \Sigma_2 \triangleright V_2 \langle X_n ?^{p_n} \rangle \cdots \langle X_1 ?^{p_1} \rangle \langle \mathsf{Int}?^q \rangle \longmapsto^* \Sigma_1 \triangleright V_1.$$

To reach V_1, the value V_2 must be of the form $V_3 \langle X_n! \rangle$ for some V_3 and

$$\Sigma \triangleright 0 \langle c \rangle \langle X_n ?^{p_n} \rangle \cdots \langle X_1 ?^{p_1} \rangle \langle \mathsf{Int}?^q \rangle \longmapsto^* \Sigma_2 \triangleright V_3 \langle X_n! \rangle \langle X_n ?^{p_n} \rangle \cdots \langle X_1 ?^{p_1} \rangle \langle \mathsf{Int}?^q \rangle .$$

Thus, $X_n!$ is a subcoercion of c (this proof is omitted). Because

$$\Sigma_2 \triangleright V_3 \langle X_n! \rangle \langle X_n ?^{p_n} \rangle \cdots \langle X_1 ?^{p_1} \rangle \langle \mathsf{Int}?^q \rangle \longmapsto \Sigma_2 \triangleright V_3 \langle X_{n-1} ?^{p_{n-1}} \rangle \cdots \langle X_1 ?^{p_1} \rangle \langle \mathsf{Int}?^q \rangle ,$$

we have

$$\Sigma \triangleright 0 \langle c \rangle \langle X_n ?^{p_n} \rangle \cdots \langle X_1 ?^{p_1} \rangle \langle \mathsf{Int}?^q \rangle \longmapsto^* \Sigma_2 \triangleright V_3 \langle X_{n-1} ?^{p_{n-1}} \rangle \cdots \langle X_1 ?^{p_1} \rangle \langle \mathsf{Int}?^q \rangle \longmapsto^* \Sigma_1 \triangleright V_1 .$$

The same argument can be made for V_3. Therefore, by induction on n, $X_n!, \ldots, X_1!$ and $\mathsf{Int}!$ are all subcoercions of c, which means that c has at least $(n+1)$ leaves when it is viewed as an abstract syntax tree. It is easy to show

$$\mathrm{size}(c) \geq 2(n+1) - 1,$$

and

$$\begin{aligned} \mathrm{size}(\langle c \rangle) &= \mathrm{size}(c) \\ &\geq 2(n+1) - 1 \\ &= 2n + 1 \\ &= \mathrm{size}(\langle \mathsf{Int}! \rangle ; \langle X_1! \rangle ; \cdots ; \langle X_n! \rangle). \end{aligned}$$

□

Remark: In fact, it is impossible to distinguish $X_1!; \cdots ; X_n!$ and $X_1!; \cdots ; X_m!$ unless the context has corresponding projections $X_i?^p$ for all i. Since the term M used in the proof does not contain projections $X_i?^p$ for any i, coercion $\mathsf{Int}!; X_1!; \cdots ; X_n!$ can be normalized to a smaller coercion, say $\mathsf{Int}!; X_1!$ without changing the behavior. In general, however, it is not the case because we can write a function that, given n, returns $0\langle \mathsf{Int}! \rangle \langle X_1! \rangle \cdots \langle X_n! \rangle$ and a list of projections $\langle X_i?^p \rangle$. Then, a caller can apply $X_n?^p, \ldots, X_1?^p$ to observe if $0 \langle \mathsf{Int}! \rangle$ is wrapped by injections $\langle X_1! \rangle \cdots \langle X_n! \rangle$ in this order.

5 RELATED WORK

Polymorphic Gradual Typing. Protection of type arguments by dynamic sealing is the standard approach to enforcing parametricity in gradual typing [Ahmed et al. 2011, 2017; Igarashi et al. 2017; Matthews and Ahmed 2008; New et al. 2020; Toro et al. 2019]. We, therefore, believe that other calculi of polymorphic gradual typing suffer the same problem of accumulated type names in a sequence of dynamic checks, as we will informally discuss below. Notice that some previous work [Ahmed et al. 2011, 2017; New et al. 2020; Toro et al. 2019] distinguishes between type variables and dynamically generated type names, unlike our calculus $\lambda C^{\forall}$. We use α, β, γ for denoting type names.

Dynamic sealing has been introduced first by Ahmed et al. [2011] to polymorphic gradual typing—although the general idea of dynamic sealing as an alternative for statically enforced type abstraction can date back to [Matthews and Ahmed 2008; Morris 1973; Pierce and Sumii 2000]—and their calculus is refined to a polymorphic calculus λB by Ahmed et al. [2017]. The calculus λB implements run-time coercion (except for sealing and unsealing) by *cast* $M : A \overset{p}{\Longrightarrow} B$, which coerces a value of type A to type B, and dynamic sealing and unsealing by *conversion* $M : A \overset{\phi}{\Longrightarrow} B$, where a conversion label ϕ signifies that the conversion seals ($\phi = -\alpha$) or unseals ($\phi = +\alpha$) a type argument with type name α. Using these constructors, as in our calculus $\lambda C^{\forall}$, the calculus λB also allows the accumulation of sealing as: $V : \mathsf{Int} \overset{-\alpha}{\Longrightarrow} \alpha : \alpha \overset{p_1}{\Longrightarrow} \star : \star \overset{-\beta}{\Longrightarrow} \beta : \beta \overset{p_2}{\Longrightarrow} \star : \star \overset{-\gamma}{\Longrightarrow} \gamma : \cdots$, where α seals Int, and β and γ seal $\star$. Removing intermediate conversions (e.g., one for β) from this accumulated sealing does not preserve the semantics. The calculus System $\mathrm{F_C}$ proposed by Igarashi et al. [2017] also enforces parametricity by the same approach. Thus, the space necessary to run programs in λB and those in System $\mathrm{F_C}$ cannot be bounded in general, as in our calculus $\lambda C^{\forall}$.

Toro et al. [2019] derived a polymorphic calculus GSF_ε by Abstracting Gradual Typing (AGT) [Garcia et al. 2016], which is a methodology to design gradually typed calculi systematically. Languages derived by AGT use *evidence* to justify coercion. Evidence is a pair of types $\langle A, B \rangle$ which intuitively represents that coercion from A to B might be successful. Evidence is gradually refined into more "precise" types [Wadler and Findler 2009] (i.e., types in which there are fewer occurrences of the dynamic type) as the program execution proceeds, and if A and B become inconsistent, the run-time system reports an error. For parametricity, Toro et al. extended evidence to pairs of *evidence types* which can accumulate type names. For example, an evidence type $\alpha^{\beta^{\mathsf{Int}}}$ means that α seals β and β seals Int. Thus, we can describe also in GSF_ε a sealing sequence that works as a counterexample to space-efficiency.

The calculus PolyC^{ν} proposed by New et al. [2020] supports term-level sealing and unsealing operations. Thus, PolyC^{ν} enables us to choose which terms are sealed, unlike other calculi, where sealing is controlled by the semantics. It is possible to run programs in PolyC^{ν} space-efficiently if no sealing is used, but it results in sacrificing parametricity.

Space-efficient Gradual Typing. Herman et al. [2007, 2010] pointed out the space consumption problem in gradual typing. They defined coercion composition and dynamic semantics for eager composition. Moreover, they proved that the space that a program in their calculus consumes is bounded. However, an algorithm for composition normalization is not clear in their work because it is defined as a set of equations.

Siek et al. [2015] proposed a space-efficient coercion calculus λS, which also has both coercion composition and eagerly composing semantics. They defined *space-efficient coercions*, which is a kind of canonical forms of coercions. Their composition is defined on space-efficient coercions as a recursive function so that it can be computed algorithmically. Kuhlenschmidt et al. [2019] implemented the Grift compiler for a gradually typed λ-calculus, by using space-efficient coercions. The Grift compiler does not fully benefit from space-efficient coercions in that coercions at tail positions do not normalize. Later, Tsuda et al. [2020] proposed an implementation technique to normalize coercions at tail positions by using coercion passing translation.

As another approach to the space-efficiency problem, Siek and Wadler [2010] introduced threesome casts to a cast (rather than coercion) calculus, another common intermediate language for gradual typing. More recently, Bañados Schwerter et al. [2021] discussed space-efficiency of AGT using Runtime Language (RL) of $\mathrm{GTFL}_{\lesssim}$, which is a gradually typed calculus with records and subtyping, proposed by Garcia et al. [2016]. In the AGT setting, type conversion is represented by evidence and composition of evidence is called *consistent transitivity* [Garcia et al. 2016]. Bañados

Schwerter et al. proved that a sufficient condition that RL can be made space-efficient is that the composition is associative and bounded. As far as we know, all the previous work for space-efficient gradual typing has been limited to a simply typed setting (with subtyping).

Coercions on Polymorphic Types. Coercions are studied as a technical device to give the semantics of subtyping [Breazu-Tannen et al. 1991; Mitchell 1984] or more general type conversion [Luo 1999; Swamy et al. 2009]. A sophisticated coercion language for parametric polymorphism has been studied by Cretin and Rémy [2012], who unify previous work [Breazu-Tannen et al. 1991; Mitchell 1988; Rémy and Yakobowski 2010] on coercions for parametric polymorphism. Our setting does not need such sophistication, though, mainly due to the separation of gradual typing and polymorphism.

6 DISCUSSION

We have proposed a polymorphic coercion calculus $\lambda C^{\forall}$, a polymorphic extension of the existing coercion calculus λC with dynamic sealing to enforce parametricity. We have proved that some coercions do not have compact forms due to interaction with dynamic sealing and the dynamic type and thus conclude that a space-efficient variant of $\lambda C^{\forall}$ cannot be derived—at least, by following the same approach by Herman et al. [2007, 2010]; Siek et al. [2015]. We believe that this result can be adapted to other existing calculi for polymorphic gradual typing.

Theorem 6 relies on the fact that a sequence of injection coercions of the form $X!$ can be arbitrarily large. We conjecture that such sequences are the *only* obstacle to space-efficient coercions and there is a version of $\lambda C^{\forall}$, which is *mostly space-efficient*. We expect that we can formally show that a calculus is mostly space-efficient by using a tweaked size function in which the size of a sequence of injections is set to be a constant.

Another observation about Theorem 6 is that, to form a sequential composition of injections, we need an injection whose source type is $\star$, as the target type of an injection is always $\star$. Since $\star$ is not a ground type, such an injection has to be of the form $X!$ where $X := \star$ in the store. Similarly, a sequence of projections can be formed only by $X?^p$ with $X := \star$.

Thus, we also conjecture that forbidding the dynamic type as a type argument will lead to a space-efficient calculus. Although it is theoretically interesting future work to prove such a conjecture, we expect that the restriction is too severe in practice—especially when dynamically typed code accesses polymorphically typed code through the dynamic type. It is likely that dynamically typed code would have no idea what type argument to pass and, as argued by Ahmed et al. [2011], passing the dynamic type to polymorphic functions seems to be a reasonable choice.

It is also an interesting question whether we can strike a balance between (space) efficiency and parametricity by reducing the overhead of coercions and dynamic sealing by other means. A possible approach is hinted by Igarashi et al. [2017], who suggested that type arguments should be sealed only when they interact with the dynamic type.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers at the 2021 Scheme and Functional Programming Workshop for their encouraging comments. This work was supported in part by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603) JST, and JSPS KAKENHI Grant Numbers JP19K20247 (Sekiyama) and JP20H00582 (Igarashi).

REFERENCES

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011. Blame for all. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 201–214. https://doi.org/10.1145/1926385.1926409

Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for free for free: parametricity, with and without types. *PACMPL* 1, ICFP (2017), 39:1–39:28. https://doi.org/10.1145/3110283

Felipe Bañados Schwerter, Alison M. Clark, Khurram A. Jafery, and Ronald Garcia. 2021. Abstracting Gradual Typing Moving Forward: Precise and Space-Efficient. *Proc. ACM Program. Lang.* 5, POPL, Article 61 (Jan. 2021), 28 pages. https://doi.org/10.1145/3434342

Gavin M. Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8586)*, Richard E. Jones (Ed.). Springer, 257–281. https://doi.org/10.1007/978-3-662-44202-9_11

Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. 2016. Practical Optional Types for Clojure. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9632)*, Peter Thiemann (Ed.). Springer, 68–94. https://doi.org/10.1007/978-3-662-49498-1_4

Val Breazu-Tannen, Thierry Coquand, Carl A Gunter, and Andre Scedrov. 1991. Inheritance as implicit coercion. *Information and Computation* 93, 1 (July 1991), 172–221. https://doi.org/10.1016/0890-5401(91)90055-7

Julien Cretin and Didier Rémy. 2012. On the power of coercion abstraction. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 361–372. https://doi.org/10.1145/2103656.2103699

Facebook. 2021. Hack. http://hacklang.org

Matthew Flatt and PLT. 2010. *Reference: Racket.* Technical Report PLT-TR-2010-1. PLT Design Inc. https://racket-lang.org/tr1/.

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 429–442. https://doi.org/10.1145/2837614.2837670

Fritz Henglein. 1994. Dynamic Typing: Syntax and Proof Theory. *Sci. Comput. Program.* 22, 3 (1994), 197–230. https://doi.org/10.1016/0167-6423(94)00004-2

David Herman, Aaron Tomb, and Cormac Flanagan. 2007. Space-Efficient Gradual Typing. In *Proceedings of the Eighth Symposium on Trends in Functional Programming, TFP 2007, New York City, New York, USA, April 2-4. 2007 (Trends in Functional Programming, Vol. 8)*, Marco T. Morazán (Ed.). Intellect, 1–18.

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher-Order and Symbolic Computation* 23, 2 (2010), 167–189. https://doi.org/10.1007/s10990-011-9066-z

Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017. On polymorphic gradual typing. *PACMPL* 1, ICFP (2017), 40:1–40:29. https://doi.org/10.1145/3110284

Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. 2019. Toward Efficient Gradual Typing for Structural Types via Coercions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 517–532. https://doi.org/10.1145/3314221.3314627

Zhaohui Luo. 1999. Coercive Subtyping. *Journal of Logic and Computation* 9, 1 (1999), 105–130.

Jacob Matthews and Amal Ahmed. 2008. Parametric Polymorphism through Run-Time Sealing or, Theorems for Low, Low Prices!. In *Proc. of European Symposium on Programming (ESOP'08)*. Springer Berlin Heidelberg, 16–31. https://doi.org/10.1007/978-3-540-78739-6_2

John C Mitchell. 1984. Coercion and type inference. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (Salt Lake City, Utah, USA) *(POPL '84)*. Association for Computing Machinery, New York, NY, USA, 175–185. https://doi.org/10.1145/800017.800529

John C. Mitchell. 1988. Polymorphic Type Inference and Containment. *Inf. Comput.* 76, 2/3 (1988), 211–249. https://doi.org/10.1016/0890-5401(88)90009-0

James H Morris, Jr. 1973. Protection in programming languages. *Commun. ACM* 16, 1 (Jan. 1973), 15–21. https://doi.org/10.1145/361932.361937

Max S. New, Dustin Jamner, and Amal Ahmed. 2020. Graduality and parametricity: together again for the first time. *Proc. ACM Program. Lang.* 4, POPL (2020), 46:1–46:32. https://doi.org/10.1145/3371114

Benjamin Pierce and Eijiro Sumii. 2000. Relating Cryptography and Polymorphism. Manuscript. http://www.kb.ecei.tohoku.ac.jp/~sumii/pub/infohide.pdf.

Didier Rémy and Boris Yakobowski. 2010. A Church-Style Intermediate Language for MLF. In *Functional and Logic Programming*. Springer Berlin Heidelberg, 24–39. https://doi.org/10.1007/978-3-642-12251-4_4

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*. 513–523.

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *In Scheme and Functional Programming Workshop*. 81–92.

Jeremy G. Siek, Peter Thiemann, and Philip Wadler. 2015. Blame and coercion: together again for the first time. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 425–435. https://doi.org/10.1145/2737924.2737968

Jeremy G Siek and Philip Wadler. 2010. Threesomes, with and without blame. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Vol. 45. ACM, 365–376. https://doi.org/10.1145/1706299.1706342

Nikhil Swamy, Michael Hicks, and Gavin M Bierman. 2009. A Theory of Typed Coercions and Its Applications. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (Edinburgh, Scotland) *(ICFP '09)*. ACM, New York, NY, USA, 329–340. https://doi.org/10.1145/1596550.1596598

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is sound gradual typing dead?. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 456–468. https://doi.org/10.1145/2837614.2837630

Matías Toro, Elizabeth Labrada, and Éric Tanter. 2019. Gradual parametricity, revisited. *Proc. ACM Program. Lang.* 3, POPL (2019), 17:1–17:30. https://doi.org/10.1145/3290330

Yuya Tsuda, Atsushi Igarashi, and Tomoya Tabuchi. 2020. Space-Efficient Gradual Typing in Coercion-Passing Style. In *34th European Conference on Object-Oriented Programming (ECOOP2020) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2020.8

Philip Wadler. 1989. Theorems for free!. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture - FPCA '89* (Imperial College, London, United Kingdom). ACM Press, New York, New York, USA, 347–359. https://doi.org/10.1145/99370.99404

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings.* 1–16. https://doi.org/10.1007/978-3-642-00590-9_1

A K Wright and M Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (Nov. 1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Ningning Xie, Xuan Bi, and Bruno C d S Oliveira. 2018. Consistent Subtyping for All. In *Programming Languages and Systems*. Springer International Publishing, 3–30. https://doi.org/10.1007/978-3-319-89884-1_1

A COMPARISON OF TWO SEMANTICS FOR UNIVERSAL COERCIONS

Let

$$\begin{aligned} V_1 &= \Lambda X.\lambda x\!:\!X.x \\ M_2 &= V_1\,\langle \forall Y.Y?^{p_1} \to Y! \rangle\,\langle \forall Z.Z! \to Z?^{p_2} \rangle\,. \end{aligned}$$

Under our semantics

$$\Sigma \triangleright (\Lambda X.V)\,\langle \forall X.c_1 \rangle \cdots \langle \forall X.c_n \rangle\,A \longmapsto (\Sigma, X := A) \triangleright V\,\langle c_1 \rangle \cdots \langle c_n \rangle$$

term $M_2\,\mathsf{Int}\,0$ reduces to 0 as follows.

$$\begin{aligned} \emptyset \triangleright M_2\,\mathsf{Int}\,0 &= \emptyset \triangleright (V_1\,\langle \forall X.X?^{p_1} \to X! \rangle\,\langle \forall X.X! \to X?^{p_2} \rangle)\,\mathsf{Int}\,0 \\ &\longmapsto X := \mathsf{Int} \triangleright (\lambda x\!:\!X.x)\,\langle X?^{p_1} \to X! \rangle\,\langle X! \to X?^{p_2} \rangle\,0 \\ &\longmapsto X := \mathsf{Int} \triangleright ((\lambda x\!:\!X.x)\,\langle X?^{p_1} \to X! \rangle\,(0\,\langle X! \rangle))\,\langle X?^{p_2} \rangle \\ &\longmapsto X := \mathsf{Int} \triangleright ((\lambda x\!:\!X.x)\,(0\,\langle X! \rangle\,\langle X?^{p_1} \rangle))\,\langle X! \rangle\,\langle X?^{p_2} \rangle \\ &\longmapsto X := \mathsf{Int} \triangleright ((\lambda x\!:\!X.x)\,0)\,\langle X! \rangle\,\langle X?^{p_2} \rangle \\ &\longmapsto X := \mathsf{Int} \triangleright 0\,\langle X! \rangle\,\langle X?^{p_2} \rangle \\ &\longmapsto X := \mathsf{Int} \triangleright 0 \end{aligned}$$

The same term would reduce to blame p_1 under the following reduction rule

$$\Sigma \triangleright (V\,\langle \forall X.c \rangle)\,A \longmapsto (\Sigma, X := A) \triangleright (V\,X)\,\langle c \rangle$$

as follows.

$$\begin{array}{lcl}
\emptyset \triangleright M_2\,\mathsf{Int}\,0 & = & \emptyset \triangleright (V_1\,\langle \forall Y.Y?^{p_1} \to Y! \rangle\,\langle \forall Z.Z! \to Z?^{p_2} \rangle)\,\mathsf{Int}\,0 \\
& \longmapsto & Z := \mathsf{Int} \triangleright (V_1\,\langle \forall Y.Y?^{p_1} \to Y! \rangle\,Z)\,\langle Z! \to Z?^{p_2} \rangle\,0 \\
& \longmapsto & Z := \mathsf{Int}, Y := Z \triangleright (V_1\,Y)\,\langle Y?^{p_1} \to Y! \rangle\,\langle Z! \to Z?^{p_2} \rangle\,0 \\
& \longmapsto & Z := \mathsf{Int}, Y := Z, X := Y \triangleright (\lambda x\!:\!X.x)\,\langle Y?^{p_1} \to Y! \rangle\,\langle Z! \to Z?^{p_2} \rangle\,0 \\
& \longmapsto & Z := \mathsf{Int}, Y := Z, X := Y \triangleright ((\lambda x\!:\!X.x)\,\langle Y?^{p_1} \to Y! \rangle\,(0\,\langle Z! \rangle))\,\langle Z?^{p_2} \rangle \\
& \longmapsto & Z := \mathsf{Int}, Y := Z, X := Y \triangleright ((\lambda x\!:\!X.x)\,(0\,\langle Z! \rangle\,\langle Y?^{p_1} \rangle))\,\langle Y! \rangle\,\langle Z?^{p_2} \rangle \\
& \longmapsto & Z := \mathsf{Int}, Y := Z, X := Y \triangleright \mathsf{blame}\,p_1
\end{array}$$