Space-Efficient Polymorphic Gradual Typing, Mostly Parametric

ATSUSHI IGARASHI, Kyoto University, Japan
SHOTA OZAKI, Kyoto University, Japan
TARO SEKIYAMA, National Institute of Informatics & SOKENDAI, Japan
YUDAI TANABE, Tokyo Institute of Technology, Japan

Since the arrival of gradual typing, which allows partially typed code in a single program, efficient implementations of gradual typing have been an active research topic. In this paper, we study the space-efficiency problem of gradual typing in the presence of parametric polymorphism. Based on the existing work that showed the impossibility of a space-efficient implementation that supports fully parametric polymorphism, this paper will show that a space-efficient implementation is, in principle, possible by slightly relaxing parametricity. We first develop $\lambda\mathrm{C}^{\forall}_{mp}$, which is a coercion calculus with mostly parametric polymorphism, and show its relaxed parametricity. Then, we present $\lambda\mathrm{S}^{\forall}_{mp}$, a space-efficient version of $\lambda\mathrm{C}^{\forall}_{mp}$, and prove that $\lambda\mathrm{S}^{\forall}_{mp}$ programs can be executed in a space-efficient manner and that translation from $\lambda\mathrm{C}^{\forall}_{mp}$ to $\lambda\mathrm{S}^{\forall}_{mp}$ is type- and semantics-preserving.

CCS Concepts: • **Theory of computation → Type theory**; **Lambda calculus**; **Operational semantics**; • **Software and its engineering → Functional languages**; **Polymorphism**.

Additional Key Words and Phrases: Gradual typing, Space efficiency, Parametricity

ACM Reference Format:
Atsushi Igarashi, Shota Ozaki, Taro Sekiyama, and Yudai Tanabe. 2024. Space-Efficient Polymorphic Gradual Typing, Mostly Parametric. *Proc. ACM Program. Lang.* 8, PLDI, Article 211 (June 2024), 24 pages. https://doi.org/10.1145/3656441

1 INTRODUCTION

1.1 Space-Efficient Gradual Typing

Siek and Taha [2006] coined the term *gradual typing* as a typing discipline to mix statically and dynamically typed portions of code in a single program. Gradual typing is supposed to make the interlanguage migration [Tobin-Hochstadt and Felleisen 2006]—the migration from fully dynamically typed programs to fully statically typed programs—smoother via partially typed programs. Since its arrival, both theoretical and practical aspects of gradual typing have been active research topics [Ahmed et al. 2011, 2017; Bañados Schwerter et al. 2021; Bauman et al. 2017; Garcia et al. 2016; Herman et al. 2007, 2010; Igarashi et al. 2017; Kuhlenschmidt et al. 2019; Matthews and Ahmed 2008; Miyazaki et al. 2019; Muehlboeck and Tate 2017; New et al. 2020; Ozaki et al. 2021; Rastogi et al. 2015; Richards et al. 2017; Siek and Taha 2007; Siek et al. 2015a, 2021, 2015b; Siek and Wadler 2010; Takikawa et al. 2016; Toro et al. 2019; Tsuda et al. 2020; Wadler and Findler 2009; Xie

Authors' addresses: Atsushi Igarashi, Graduate School of Informatics, Kyoto University, Japan, igarashi@kuis.kyoto-u.ac.jp; Shota Ozaki, Graduate School of Informatics, Kyoto University, Japan, ozaki@fos.kuis.kyoto-u.ac.jp; Taro Sekiyama, National Institute of Informatics & SOKENDAI, Japan, tsekiyama@acm.org; Yudai Tanabe, Department of Mathematical and Computing Science, School of Computing, Tokyo Institute of Technology, Japan, yudaitnb@prg.is.titech.ac.jp.

ACM 2475-1421/2024/6-ART211
https://doi.org/10.1145/3656441

et al. 2018] and several implementations of practical gradually typed languages, such as Typed Racket [Flatt and PLT 2010], Typed Closure [Bonnaire-Sergeant et al. 2016], Hack [Facebook 2021], and TypeScript [Bierman et al. 2014], have emerged.

Along with the success of gradual typing in practice, its efficient implementation has gained much attention. It has been revealed that the efficiency of gradually typed programs is dominated by the efficiency of type conversions that involve run-time type checking. 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]. Several attempts have been made to address the efficient implementations [Bauman et al. 2017; Kuhlenschmidt et al. 2019; Muehlboeck and Tate 2017; Rastogi et al. 2015; Richards et al. 2017].

The space-efficiency problem in gradual typing was recognized first by Herman et al. [2007, 2010]. They proposed a coercion calculus as an intermediate language to implement gradual typing, where a sequence of conversions—which are called *coercions* [Henglein 1994] in coercion calculi—can be normalized into a simpler conversion by a meta-level composition operation. They also proved that coercions emerging at run time can be composed 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. [2015a, 2021] refined their work by formalizing the notion of space-efficient coercions and proved their correctness. Specifically, they gave formal translation from an unoptimized, non-space-efficient coercion calculus λC to an optimized, space-efficient coercion calculus λS and proved that the translation is fully abstract. Kuhlenschmidt et al. [2019] developed Grift, a compiler for a gradually typed language. Grift, which uses space-efficient coercions internally, demonstrated that catastrophic slowdowns could be avoided without introducing significant average-case overhead, although it implemented the space-efficient semantics only partially because the full implementation would rely on somewhat peculiar semantics. Later, Tsuda et al. [2020] proposed coercion-passing style to make it easy to fully implement the space-efficient semantics and extended Grift. However, all the work on space efficiency discussed here has been limited to a simply typed setting.

More recently, Ozaki et al. [2021] studied this problem in a polymorphic type system and showed a negative result. They introduced a polymorphic extension $\lambda C^{\forall}$ of λC with parametric polymorphism—where parametricity is dynamically enforced by dynamic sealing [Morris 1973], following Ahmed et al. [2011]—and proved that it is impossible to make $\lambda C^{\forall}$ space-efficient as in Siek et al. [2015a, 2021], because there is a program that generates a sequence of coercions that cannot be normalized into a simpler coercion.

1.2 Our Work

This paper presents a positive result that $\lambda C^{\forall}$ can be made space-efficient with *mostly parametric* polymorphism. Along with their negative result, Ozaki et al. [2021] have observed that a problematic sequence of coercions emerges when a type abstraction is applied to the fully dynamic type $\star$ and conjectured that forbidding $\star$ as a type argument might make a space-efficient version of $\lambda C^{\forall}$ possible, with a comment that such a restriction would not be very practical. Our key idea is, instead of prohibiting $\star$ as a type argument, to ensure parametricity only if a polymorphic value is applied to non-dynamic types.[1] We will formally show that the language with the modified semantics enjoys parametricity in a certain relaxed sense and can be made space-efficient.

We make the following technical contributions in this paper:

[1]Non-dynamic types exclude only $\star$ and include types where $\star$ appear as their part, such as $\star \rightarrow \mathsf{Int}$.

- We introduce a polymorphic coercion calculus $\lambda C^{\forall}_{mp}$, which is a variant of $\lambda C^{\forall}$ by Ozaki et al. [2021], with mostly parametric semantics, in which parametricity is enforced dynamically only if the type argument to a polymorphic value is not the fully dynamic type $\star$. We prove a relaxed version of parametricity and an example free theorem, together with its basic properties, including type safety.
- We define another coercion calculus $\lambda S^{\forall}_{mp}$ with space-efficient coercions and show that $\lambda S^{\forall}$ is space efficient.
- We give a formal translation from $\lambda C^{\forall}_{mp}$ to $\lambda S^{\forall}_{mp}$ and prove that the translation preserves typing and semantics. (As a byproduct, we have fixed a flaw in previous work [Siek et al. 2015a, 2021].)

The extension of space-efficient coercions to the mostly parametric polymorphism is not trivial. First of all, in $\lambda C^{\forall}_{mp}$ (as well as $\lambda C^{\forall}$), type variables can appear in coercion—in universal coercions of the form $\forall X.c$ for conversion between universal types, where c can refer to X for conversion between X and $\star$. Thus, normalizing coercions into a simpler form has to deal with coercions with type variables bound by $\forall$. However, coercions can behave differently, depending on whether the type variable is instantiated with $\star$ or not during reduction. Thus, it appears difficult to compose universal coercions before type variables are instantiated. We solve the problem by extending the form of universal coercions to $\forall X.c_1 ⨟ c_2$ so that they keep two coercions: the one where the type variable is instantiated by non-dynamic types and the other by $\star$. Then, component-wise composition will work.

We note that the present work presents only theoretical results, although it is concerned about an efficient implementation of gradually typed languages. It is left for future work to answer empirical questions, such as whether the proposed extension of space-efficient coercions can really be implemented efficiently (in terms of both time and space) or whether the relaxed parametricity is practically useful.

Organization of the Paper. In Section 2, we review the work on space-efficient gradual typing. The definition and basic properties of $\lambda C^{\forall}_{mp}$ are presented in Section 3. Section 4 introduces $\lambda S^{\forall}_{mp}$, gives a translation from $\lambda C^{\forall}_{mp}$ to $\lambda S^{\forall}_{mp}$, and shows that $\lambda S^{\forall}_{mp}$ is space-efficient and the translation is type- and semantics-preserving. We discuss related work in Section 5 and conclude in Section 6.

This paper omits the formal definitions of some well-known notions, auxiliary lemmas, and detailed proofs. The full definitions, lemmas, and proofs are found in the supplementary material.

2 BACKGROUND

This section reviews the problem of space efficiency in gradual typing and a solution to it. We first briefly recall gradual typing and informally introduce the coercion calculus λC of Henglein [1994] and Siek et al. [2015a, 2021] as an intermediate language for gradually typed languages. We then present the space-efficiency problem, which states that coercions of unbounded size can appear at run time, and the solution originally proposed by Herman et al. [2007, 2010] and later refined by Siek et al. [2015a, 2021]. The essence of the solution is to normalize a sequence of coercions into a simpler form, whose size is statically bounded by types.

2.1 Gradual Typing and the Coercion Calculus λC

The framework for gradual typing is usually presented with two languages: a surface language in which programs are written and an intermediate language in which programs are executed. The surface language introduces a special type (called "dynamic" and often written $\star$ or ?) to signify dynamically typed portions in a program and has a type system that relaxes the usual static typing discipline. For example, consider a program

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

in the surface language. The type system of the surface language is optimistic about the use of $\star$: term M_1 can be of an arbitrary type and x can be used as any type in its scope. The notion of type consistency [Siek and Taha 2006] (or consistent subtyping [Siek and Taha 2007] if the language is equipped with subtyping) is usually used to specify how static typing is relaxed. To compensate for the relaxed static typing, the translation from the surface language to the intermediate language inserts type conversions with run-time checks. For example, M_1 is subject to conversion to $\star$ before x is bound and x is converted to Int before addition. The conversion from $\star$ to Int checks dynamically whether x is really an integer and, if not, the program execution raises an exception. We do not discuss the surface language or type consistency further, as our interest in this paper is how dynamic checks are performed in the intermediate language.

The coercion calculus λC is an intermediate language for gradual typing. It exposes where dynamic checking is performed by using *coercions*, which describe how a dynamic conversion from one type to another is performed. For example, $\mathsf{Int!}$, which is a coercion from Int to $\star$, is for adding a type-tag to a raw integer to construct a tagged value—every value of the type $\star$ is a pair, written $V\langle G!\rangle$, of a type-tag G and a value V, such as $42\langle\mathsf{Int!}\rangle$ and $\mathsf{true}\langle\mathsf{Bool!}\rangle$. Conversely, $\mathsf{Int?}^p$, which is a coercion from $\star$ to Int, is for tag-checking: It first checks that the type-tag of the input value is the one for integers, and, if the check succeeds, removes the type-tag to return a raw integer; or, if the check fails, an exception labeled p will be raised. (The superscript p, called a *blame label*, is used to identify which ?-coercion failed and is supposed to be useful in analyzing the cause of a run-time error [Wadler and Findler 2009].) In general, !- and ?-coercions (called *injections* and *projections*, respectively) are available for each kind of value (integers, Booleans, functions, etc.).

Given a coercion c, a term $M\langle c\rangle$ (called coercion application) applies coercion c to the value of term M. Using the coercions introduced thus far, the example given at the beginning of this section can be translated to the following term in λC:[2]

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

where the variable x of the type $\star$ is coerced to Int by projection $\mathsf{Int?}^p$ before applying the addition, which expects two Ints. If the value of x does not come with the type-tag for integers, the program fails before performing the addition. In general, the tag-checking step by a projection is expressed by the following two reduction rules:

$$(V\langle G!\rangle)\langle G?^p\rangle \longrightarrow V \qquad \text{and} \qquad (V\langle G_1!\rangle)\langle G_2?^p\rangle \longrightarrow \mathsf{blame}\,p \quad (G_1 \neq G_2)$$

where $\mathsf{blame}\,p$ is a special term to denote a run-time error. For example, if M_1 is $42\langle\mathsf{Int!}\rangle$, then the expression above reduces to $(42\langle\mathsf{Int!}\rangle)\langle\mathsf{Int?}^p\rangle + V$ (V is the value of M_2) and then to $42 + V$. If M_1 is $\mathsf{true}\langle\mathsf{Bool!}\rangle$, the expression reduces to $(\mathsf{true}\langle\mathsf{Bool!}\rangle)\langle\mathsf{Int?}^p\rangle + V$ and then to $\mathsf{blame}\,p + V$.

Henglein [1994] also introduced coercions for first-class functions. A *function coercion*, which takes the form $c \rightarrow d$, wraps a given function so that arguments and return values are coerced by coercions c and d, respectively. For example, an application of the term $(\lambda x{:}\star.M)\langle\mathsf{Int!} \rightarrow \mathsf{Bool?}^p\rangle$ to a value V' reduces as follows:

$$((\lambda x{:}\star.M)\langle\mathsf{Int!} \rightarrow \mathsf{Bool?}^p\rangle)\ V' \longrightarrow ((\lambda x{:}\star.M)\ (V'\langle\mathsf{Int!}\rangle))\langle\mathsf{Bool?}^p\rangle\ .$$

[2]This translation to λC can be performed automatically [Siek and Taha 2006; Siek et al. 2021]: we can find where coercions are inserted from a typing derivation in the surface language.

Here, the argument V' is first coerced to the dynamic type by the argument coercion $\mathsf{Int!}$ and then passed to the function $\lambda x{:}\star.M$. Once a value of the dynamic type is returned, it is coerced to Bool by $\mathsf{Bool?}^p$.

2.2 Space-Efficiency Problems in λC

Although λC offers an elegant formalism to implement gradual typing, a straightforward implementation would cause accumulation of coercions, causing undesirable space overhead, as first observed by Herman et al. [2007, 2010]. For example, consider the following mutually recursive functions *even* and *odd* in gradual typing:

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

The functions *even* and *odd* return a Boolean value indicating whether an argument is even and odd, respectively. Because these functions are written in tail-recursive form, one might expect that a call to these functions only consumes a constant space. However, the fact that the return type of *even* is specified to be $\star$ complicates the matter.

First, they translate to the following function definitions in λC:

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

Here, injections are inserted in even_C because it has to return $\star$, whereas a projection is inserted in odd_C because it has to return Bool but even_C returns $\star$. Notice that recursive calls are *not* at tail positions any longer and unbounded growth of a term under reduction can occur. For instance, the reduction of $\mathit{odd}_\mathrm{C}\ 4$ proceeds as follows:

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

As the reduction sequence indicates, each time even_C and odd_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 execution needs a stack space that the program size cannot bound. Even worse, non-terminating programs may consume infinite spaces for storing coercions.

Aside from coercions at tail positions, function coercions also cause accumulation of coercions because function coercions can nest an arbitrary number of times on a function value. In fact, it is reported that programs that intensively use wrappers for checking the behavior of functions can exhibit catastrophic overhead [Takikawa et al. 2016].

Herman et al. solved this problem theoretically by refining the semantics of λC so that nested coercions are normalized to simpler ones at run time. For example, we can find that application of coercions $M\langle \mathsf{Bool!}\rangle\ \langle \mathsf{Bool?}^p\rangle$ is equivalent to $M\langle \mathsf{id}_{\mathsf{Bool}}\rangle$ because applying projection $\langle \mathsf{Bool?}^p\rangle$ to values injected by $\langle \mathsf{Bool!}\rangle$ always succeeds. Also, nested function coercions $M\langle \mathsf{Bool?}^{p_1} \to \mathsf{Int!}\rangle\ \langle \mathsf{Bool!} \to \mathsf{Int?}^{p_2}\rangle$ are equivalent to $M\langle (\mathsf{Bool!}\ ;\ \mathsf{Bool?}^{p_1}) \to (\mathsf{Int!}\ ;\ \mathsf{Int?}^{p_2})\rangle$, which further simplifies to $M\langle \mathsf{id}_{\mathsf{Bool}} \to \mathsf{id}_{\mathsf{Int}}\rangle$. They generalized this idea and defined coercion normalization. Once we allow the reduction $M\langle c_1\rangle\ \langle c_2\rangle \longrightarrow M\langle c_3\rangle$, which normalizes c_1 and c_2 into c_3 even before reducing M, the above example reduces as follows:

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

Because consecutively applied coercions are immediately normalized, the number of coercions emerging during the reduction can increase only by a constant factor. Furthermore, Herman et al. proved that the size of a normalized coercion is bounded by the height of the largest type in the original program before the 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. [2015a, 2021] refined the result of Herman et al., who had given coercion normalization without taking blame labels into account. Siek et al. formulated blame-aware normalization as a meta-level operation $c \fatsemi d$ that collapses the composition of c and d into a simpler coercion. For example, $\mathsf{Bool}! \fatsemi \mathsf{Bool}?^{p}$ returns $\mathsf{id}_{\mathsf{Bool}}$ and $\mathsf{Bool}?^{p} \fatsemi \mathsf{id}_{\mathsf{Bool}}$ returns $\mathsf{Bool}?^{p}$.[3] They also developed a "space-efficient" coercion calculus λS and provided a fully abstract translation from λC to λS.

2.3 Parametrically Polymorphic Gradual Typing Cannot Be Made Space-Efficient

Ozaki et al. [2021] extended λC to a parametrically polymorphic coercion calculus $\lambda\mathrm{C}^{\forall}$ by applying the idea of *dynamic sealing*, which is a standard approach to achieving parametricity in gradual typing [Ahmed et al. 2011, 2017; Igarashi et al. 2017; New et al. 2020; Toro et al. 2019] (as far as we know, there is no polymorphic gradually typed language that satisfies parametricity without resting on dynamic sealing). Ahmed et al. [2011] noticed that the usual type-level β-reduction $(\Lambda X.M)\ A \longrightarrow M[X := A]$ with type substitution $[X := A]$ breaks parametricity. For recovering parametricity, they introduced dynamic sealing [Abadi et al. 1995; Matthews and Ahmed 2008; Morris 1973; Pierce and Sumii 2000], which seals the actual type parameter A with a fresh type name α generated at run time. The generated name α is recorded and associated with the type A in a *name store* Σ. Formally, the reduction relation with dynamic sealing is expressed as a four-place relation $\Sigma \triangleright M \longrightarrow \Sigma' \triangleright M'$ and the reduction of type applications is formulated as follows:

$$\Sigma \triangleright (\Lambda X.M)\ A \longrightarrow \Sigma, \alpha := A \triangleright M[X := \alpha]$$

Ozaki et al. introduced new forms of coercions to embody parametricity ensured by dynamic sealing in a coercion calculus. For example, they extended the type-tags in injections to involve type variables and names. This extension allows a term $\Lambda X.\lambda x{:}X.x\langle X!\rangle$, which takes a type parameter X and value parameter x and then returns x coerced to $\star$ via the coercion $X!$, to be typed at $\forall X.X \to \star$. Furthermore, its application to a type A and a value V of the type A reduces as follows:

$$\Sigma \triangleright (\Lambda X.\lambda x{:}X.x\langle X!\rangle)\ A\ V \longrightarrow \Sigma, \alpha := A \triangleright (\lambda x{:}\alpha.x\langle \alpha!\rangle)\ V \longrightarrow \Sigma, \alpha := A \triangleright V\langle \alpha!\rangle \quad (\alpha \text{ is fresh})\ .$$

In the result, the value V is coerced to $\star$ via the injection $\langle \alpha!\rangle$. Their calculus $\lambda\mathrm{C}^{\forall}$ allows V to be typed at the type name α associated with its type A by the name store $\Sigma, \alpha := A$.[4] As α is fresh, the context cannot know it. Therefore, after the reduction, the context cannot remove the tag α: any attempt to remove the tag by a projection must fail. Because this behavior is independent of the type argument A, the use of the freshly generated type name α makes the behavior of $\Lambda X.\lambda x{:}X.x\langle X!\rangle$ uniform whatever type argument is passed.

While dynamic sealing is thus crucial to make polymorphic gradual typing parametric, Ozaki et al. also discovered that it prevents $\lambda\mathrm{C}^{\forall}$ from being space-efficient. They gave a type-name-free recursive program that, for any number n, can reduce to $\Sigma \triangleright M\langle \alpha_1!\rangle \cdots \langle \alpha_n!\rangle$ for some term M,

[3]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. [2021].
[4]More precisely, $\lambda\mathrm{C}^{\forall}$ provides coercions to convert a value of the type A to the type name α and vice versa.

generated type names $\alpha_1, \cdots, \alpha_n$, and name store Σ that associates each α_i with $\star$. To make $\lambda C^\forall$ space-efficient, we need to simplify the coercion sequence $\alpha_1!, \cdots, \alpha_n!$ in a meaning-preserving way; otherwise, the total size of coercions appearing during the reduction can grow unboundedly, resulting in the break of space efficiency. However, it is impossible: the only way to simplify $\alpha_1!, \cdots, \alpha_n!$ is to drop some projection $\alpha_i!$, but it changes the original meaning. It is notable that associating each α_i with $\star$ by Σ is important to typecheck $M\langle\alpha_1!\rangle \cdots \langle\alpha_n!\rangle$ because, in general, for a term $M\langle\alpha_1!\rangle\,\langle\alpha_2!\rangle$ to be well typed, α_2 is required to be associated with the type $\star$ of $M\langle\alpha_1!\rangle$.

3 POLYMORPHIC COERCION CALCULUS $\lambda C^\forall_{mp}$

The proof of the impossibility of a space-efficient implementation [Ozaki et al. 2021] relies on a sequence of coercion applications $M\langle\alpha_1!\rangle \cdots \langle\alpha_n!\rangle$ of an unbounded length with different names. It may appear that unbounded name generation is problematic but it is not: it is possible to construct a program that generates only one name α and, for any n, $M\langle\alpha!\rangle \cdots \langle\alpha!\rangle$ with n coercions $\alpha!$ and prove that $\langle\alpha!\rangle \cdots \langle\alpha!\rangle$ cannot be simplified into a smaller coercion (see the supplementary material). This fact suggests that the root cause of the problem is not name generation but *a type name associated with* $\star$. In fact, in the conclusion of their paper, Ozaki et al. [2021] have conjectured that prohibiting $\star$ as a type argument might make a space-efficient calculus with a comment that such a restriction would be too severe, as passing $\star$ would play an important role when dynamically typed and polymorphic code interact—especially when dynamically typed code accesses polymorphically typed code through the dynamic type, as argued by Ahmed et al. [2011].

We advance the insight by Ozaki et al. [2021] one step further and argue rigorously that what hampers space efficiency is a type name associated with $\star$. Our key idea is to *allow* $\star$ *to be passed but give up parametricity to some extent*, rather than to disallow $\star$ completely. More specifically, we change the semantics of $\lambda C^\forall$ so that type-level β-reduction *does not generate a type name if the type argument is* $\star$:

$$\Sigma \triangleright (\Lambda X.M)\,A \longrightarrow \begin{cases} \Sigma \triangleright M[X := \star] & (\text{if } A = \star) \\ \Sigma, \alpha := A \triangleright M[X := \alpha] & (\text{otherwise}) \end{cases}$$

The price we pay is the loss of parametricity. However, the loss is, we believe, slight—for a type that includes the dynamic type as a part of it (such as $\star \to \star$), type names will be generated and parametric behavior will be (dynamically) checked. We think this is a reasonable compromise as the use of $\star$ as a type argument would suggest that this part of code is not yet ready to enjoy the benefits static types could give.[5] As far as we know, the new semantics does not lose other important properties as a gradually typed language.

This section develops $\lambda C^\forall_{mp}$, a "mostly parametric" version of $\lambda C^\forall$, and shows its basic properties including type soundness and (a relaxed notion of) parametricity. The calculus $\lambda C^\forall_{mp}$ is defined in a manner similar to $\lambda C^\forall$ except that: the type-level β-reduction in $\lambda C^\forall$ is parametric while that in $\lambda C^\forall_{mp}$ is not; and type variables are used to represent type names in $\lambda C^\forall$ while type names and variables are distinguished in $\lambda C^\forall_{mp}$ (following Ahmed et al. [2017]). What we do *not* study is the *dynamic gradual guarantee* (DGG) [Siek et al. 2015b]. Actually, we expect neither $\lambda C^\forall$ nor $\lambda C^\forall_{mp}$

[5]There is a design space to reduce the loss further. For instance, consider a semantics where $(\Lambda X.M)\,A$ generates no type name if A is $\star$ *or* M *does not involve coercions of the form* $\langle X!\rangle$. This semantics seems space-efficient, as a coercion $\langle\alpha!\rangle$ is generated only for α not associated with $\star$, and it allows more programs to benefit from parametricity. However, it is unclear whether we could receive a satisfactory payoff of implementing such a complex semantics. The present work rather aims to establish a reasonable compromise—giving up parametricity only when $\star$ is passed—for space efficiency, not to explore "necessary" conditions of space-efficient polymorphic gradual typing.

$$
\begin{array}{rrcl}
\text{Base types} & \iota & & \text{Blame labels} \quad p, q \qquad \text{Type variables} \quad X, Y \\
\text{Types} & A, B, C & ::= & \iota \mid \star \mid A \to B \mid \forall X.A \mid X \mid \alpha \\
\text{Ground types} & G, H & ::= & \iota \mid \star \to \star \mid \forall X.\star \mid X \mid \alpha \\
\text{Coercions} & c, d & ::= & \mathsf{id}_A \mid G! \mid G?^p \mid \alpha^- \mid \alpha^+ \mid c \to d \mid \forall X.c \mid c \,;\, d \mid \bot^p_{A \leadsto B} \\
\text{Terms} & M & ::= & x \mid U \mid M\,M \mid M\,A \mid M\langle c \rangle \mid \mathsf{blame}\,p \\
\text{Uncoerced Values} & U & ::= & k \mid \lambda x\!:\!A.M \mid \Lambda X.(M : A) \\
\text{Values} & V & ::= & U \mid V\langle \alpha^- \rangle \mid V\langle G! \rangle \mid V\langle c \to d \rangle \mid V\langle \forall X.c \rangle \\
\text{Frames} & E & ::= & \square\,M \mid V\,\square \mid \square\,A \mid \square\langle c \rangle \\
\text{Type environments} & \Gamma & ::= & \emptyset \mid \Gamma, x : A \mid \Gamma, X \\
\text{Stores} & \Sigma & ::= & \emptyset \mid \Sigma, \alpha := \mathbb{A}
\end{array}
$$

Fig. 1. Polymorphic Coercion Calculus $\lambda C^{\forall}_{mp}$: Syntax

enjoys this property because they are similar to GSF, a polymorphic gradually typed language studied by Toro et al. [2019], which is proven to invalidate the DGG.

3.1 Syntax

Figure 1 presents the syntax of $\lambda C^{\forall}_{mp}$. Let ι range over *base types*, which include Int, Bool, and Str. The set of *types*, ranged over by A, B, consists of base types ι, the dynamic type $\star$, function types $A \to B$, universal types $\forall X.A$, type variables X, and type names α. A universal type $\forall X.A$ binds the type variable X in A. The set of *ground types*, ranged over by G, H, consists of base types ι, the function type $\star \to \star$, the universal type $\forall X.\star$, type variables, and type names. Ground types represent type tags attached to values of the dynamic type and used for injections and projections. The ground type $\forall X.\star$ represents the tag for type abstractions. We use metavariables $\mathbb{A}$, $\mathbb{B}$, and $\mathbb{C}$ to denote types that are not the dynamic type $\star$. Let p, q range over *blame labels*, which represent program points to identify where run-time checking fails.

The set of *coercions*, ranged over by c, d, consists of identity coercions id_A, injections $G!$, projections $G?^p$, concealment α^-, revelation α^+, function coercions $c \to d$, universal coercions $\forall X.c$, sequential compositions $c \,;\, d$, and failure coercions $\bot^p_{A \leadsto B}$. Concealment and revelation, which correspond to static casts in Ahmed et al. [2011], conversions in Ahmed et al. [2017], and sealing/unsealing operations in New et al. [2020], are conversions between α and $\Sigma(\alpha)$, the type that α is associated with in store Σ. A universal coercion $\forall X.c$, which binds the type variable X in c, works as a conversion between universal types: 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 coercion from type $\forall X.\star \to \star$ to type $\forall X.X \to X$ is written $\forall X.X! \to X?^p$. 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. 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]. As its name suggests, a failure coercion represents a failure and raises blame if it is applied to a value. As we see later, the semantics relies on the uniqueness of coercion typing; thus, a failure coercion is annotated with source type A and target type B.

The set of *terms*, ranged over by M, consists of: constants k including integers, Booleans (true and false), and first-order primitive functions; variables x; function abstractions $\lambda x\!:\!A.M$; function applications $M_1\,M_2$; type abstractions $\Lambda X.(M : A)$; type applications $M\,A$; coercion applications $M\langle c \rangle$; and run-time errors $\mathsf{blame}\,p$, which are raised by the failure of a projection $G?^p$ or application of a failure coercion $\bot^p_{A \leadsto B}$. As in $\lambda C^{\forall}$ [Ozaki et al. 2021], the body of a type abstraction is annotated with its type for making the semantics deterministic; we will see it in detail shortly. A

Coercion typing $\boxed{\Sigma \mid \Gamma \vdash c : A \rightsquigarrow B}$

$$\frac{\vdash \Sigma \quad \Sigma \vdash \Gamma \quad \Sigma \mid \Gamma \vdash A}{\Sigma \mid \Gamma \vdash \mathrm{id}_A : A \rightsquigarrow A}\ (\textsc{Ct\_Id\_C})$$

$$\frac{\vdash \Sigma \quad \Sigma \vdash \Gamma \quad \Sigma \mid \Gamma \vdash G}{\Sigma \mid \Gamma \vdash G! : G \rightsquigarrow \star}\ (\textsc{Ct\_Inj\_C}) \qquad \frac{\vdash \Sigma \quad \Sigma \vdash \Gamma \quad \Sigma \mid \Gamma \vdash G}{\Sigma \mid \Gamma \vdash G?^p : \star \rightsquigarrow G}\ (\textsc{Ct\_Proj\_C})$$

$$\frac{\begin{array}{c}\Sigma \mid \Gamma \vdash c : A' \rightsquigarrow A \\ \Sigma \mid \Gamma \vdash d : B \rightsquigarrow B'\end{array}}{\Sigma \mid \Gamma \vdash c \to d : (A \to B) \rightsquigarrow (A' \to B')}\ (\textsc{Ct\_Arr\_C}) \qquad \frac{\begin{array}{c}\Sigma \mid \Gamma \vdash c : A \rightsquigarrow B \\ \Sigma \mid \Gamma \vdash d : B \rightsquigarrow C\end{array}}{\Sigma \mid \Gamma \vdash c\,;d : A \rightsquigarrow C}\ (\textsc{Ct\_Seq\_C})$$

$$\frac{\vdash \Sigma \quad \Sigma \vdash \Gamma \quad \alpha := \mathbb{A} \in \Sigma}{\Sigma \mid \Gamma \vdash \alpha^- : \mathbb{A} \rightsquigarrow \alpha}\ (\textsc{Ct\_Conceal\_C}) \qquad \frac{\vdash \Sigma \quad \Sigma \vdash \Gamma \quad \alpha := \mathbb{A} \in \Sigma}{\Sigma \mid \Gamma \vdash \alpha^+ : \alpha \rightsquigarrow \mathbb{A}}\ (\textsc{Ct\_Reveal\_C})$$

$$\frac{\Sigma \mid \Gamma, X \vdash c : A \rightsquigarrow B}{\Sigma \mid \Gamma \vdash \forall X.c : \forall X.A \rightsquigarrow \forall X.B}\ (\textsc{Ct\_All\_C}) \qquad \frac{\vdash \Sigma \quad \Sigma \vdash \Gamma \quad \Sigma \mid \Gamma \vdash A \quad \Sigma \mid \Gamma \vdash B}{\Sigma \mid \Gamma \vdash \bot^p_{A \rightsquigarrow B} : A \rightsquigarrow B}\ (\textsc{Ct\_Fail\_C})$$

Term typing $\boxed{\Sigma \mid \Gamma \vdash M : A}$

$$\frac{\vdash \Sigma \quad \Sigma \vdash \Gamma \quad ty(k) = A}{\Sigma \mid \Gamma \vdash k : A}\ (\textsc{T\_Const\_C}) \qquad \frac{\Sigma \mid \Gamma \vdash M : A \quad \Sigma \mid \Gamma \vdash c : A \rightsquigarrow B}{\Sigma \mid \Gamma \vdash M\langle c\rangle : B}\ (\textsc{T\_Crc\_C})$$

Fig. 2. Polymorphic Coercion Calculus $\lambda C^{\forall}_{mp}$: Typing

function abstraction $\lambda x\!:\!A.M$ binds the variable x in M, and a type abstraction $\Lambda X.(M : A)$ binds the type variable X in M and A. *Values*, ranged over by V, are constants, function abstractions, type abstractions, or values with a concealment coercion $V\langle\alpha^-\rangle$, an injection coercion $V\langle G!\rangle$, a function coercion $V\langle c \to d\rangle$, or a universal coercion $V\langle\forall X.c\rangle$. The values of the first three forms are also called *uncoerced values*, ranged over by U.

Let E range over *frames*, which are *single-level* evaluation contexts [Siek et al. 2021] to express standard left-to-right evaluation. Let Γ range over *type environments*. A type environment is a sequence of (1) pairs of a variable and its type $x : A$ and (2) type variables X. Let Σ range over *stores*. A store is a sequence of pairs $\alpha := \mathbb{A}$ of a type name and a non-dynamic type. We assume that all variables (x in $\Gamma_1, x : A, \Gamma_2$) and type variables ($X$ in Γ_1, X, Γ_2) in a type environment and all type names in a store are pair-wise distinct.

The set of free variables in a term and the sets of free type variables in a term, a coercion, and a type are defined in a standard manner. We define α-conversion in the standard manner and identify α-equivalent types, coercions, and terms. We write $M[x := V]$ for capture-avoiding substitution of V for x in M and $B[X := A]$ for capture-avoiding type substitution of A for X in B. They are defined straightforwardly. The only possible forms of type substitutions applied to coercions and terms are $[X := \alpha]$ or $[X := \star]$. The latter (for coercions) needs a little ingenuity: $X?^p[X := \star]$ and $X![X := \star]$ are defined to be $\mathrm{id}_\star$. Otherwise, it is straightforward.

3.2 Type System

Figure 2 presents the type system, which consists of five judgment forms: type well-formedness $\Sigma \mid \Gamma \vdash A$, which means that type A is well formed under store Σ and type environment Γ; store well-formedness $\vdash \Sigma$, which means that store Σ is well formed; type environment well-formedness $\Sigma \vdash \Gamma$, which means that type environment Γ is well formed under Σ; coercion typing $\Sigma \mid \Gamma \vdash c : A \rightsquigarrow B$, which means that c is a well-formed coercion from *source type* A to *target type* B; and term typing $\Sigma \mid \Gamma \vdash M : A$, which means term M is given type A under store Σ and type environment Γ. We omit the well-formedness rules because they are defined in a straightforward manner. Interested readers are referred to the supplementary material.

The rules for coercion typing are a straightforward adaptation of previous work on coercions [Henglein 1994; Siek et al. 2021]. An identity coercion id_A is a coercion from A to itself. An injection is typed as a coercion from a ground type G to $\star$, and, conversely, a projection $G?^p$ is from $\star$ to G. Similarly, a concealment α^- converts $\mathbb{A}$ to α, where α is associated with $\mathbb{A}$ in Σ, and a revelation α^+ is its converse. A function coercion $c \to d$ is a coercion from a function type $A \to B$ to a function type $A' \to B'$ if c coerces A' to A and d coerces B to B'. The coercion constructor $\to$ is contravariant in the coercion c for arguments. 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. The typing for universal coercions $\forall X.c$ is not surprising; $\forall X.c$ 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. A failure coercion $\perp^p_{A \rightsquigarrow B}$ is given the declared source and target types.

The rules for term typing are also straightforward; we show only a few rules that deserve remarks. In (T_Const_C), ty is a (meta-level) function that maps a constant k to a first-order type of the form $\iota_1 \to \iota_2 \to \cdots \to \iota_n (n \geq 1)$. The rule (T_Crc_C) means that a term $M\langle c \rangle$ is of type B if M of type A is coerced by c of $A \rightsquigarrow B$.

3.3 Operational Semantics

Figure 3 defines the reduction relation, which involves stores and is written $\Sigma \triangleright M \longrightarrow \Sigma' \triangleright M'$. If the stores are the same on both sides, that is $\Sigma = \Sigma'$, we omit them. For example, the rule (R_Beta_C) is understood as $\Sigma \triangleright (\lambda x\!:\!A.M)\ V \longrightarrow \Sigma \triangleright M[x := V]$.

Most rules are a straightforward adaptation of previous coercion calculi [Henglein 1994; Siek et al. 2021]. We first explain the rules except (R_TybetaDyn_C) and (R_Tybeta_C). The rule (R_Delta_C) 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 $ty(\delta(k_1, k_2)) = A$. The rule (R_Beta_C) represents standard β-reduction. The rule (R_Id_C) means that an identity coercion is an identity function. The rule (R_Wrap_C) 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. The rules (R_Collapse_C) and (R_Conflict_C) represent type tag checking: (R_Collapse_C) is for the case that V passes the check because the ground types in the injection and projection are the same; they are removed after reduction. The rule (R_Conflict_C) is for the case that V does not pass because the ground types in the injection and projection are different; the term reduces to blame p with label p attached to the projection. The rule (R_Remove_C) represents that a concealment is canceled by a revelation; unlike tag checking, the type variables in the concealment and revelation are necessarily the same in a well-typed term. The rule (R_Split_C) splits a composition coercion into two consecutively applied coercions. The rule (R_Fail_C) means that $\perp^p_{A \rightsquigarrow B}$ raises blame. The rule (R_Ctx_C) is standard, which allows a subterm to reduce and the rule (R_Blame_C) pulls out a run-time error blame p by discarding the current frame.

The rules (R_TybetaDyn_C) and (R_Tybeta_C) are to reduce an application of a type abstraction to a type argument. In these rules, $\overline{\langle c \rangle}$ stands for a sequence of zero or more coercion applications, i.e., $\langle c_1 \rangle \cdots \langle c_n \rangle$. Thus, the left-hand side $(\Lambda X.(M : A_0))\overline{\langle \forall X.c \rangle}\ B$ represents that a type abstraction with a sequence of universal coercions is applied to type argument B. The former (R_TybetaDyn_C) means that an application to $\star$ does not generate a type name and $\star$ is substituted for the type

Coercion generation $\boxed{\mathit{coerce}^{+}_{\alpha}(A) = c, \mathit{coerce}^{-}_{\alpha}(A) = c}$

$$
\begin{array}{rclrclrcl}
\mathit{coerce}^{\pm}_{\alpha}(\iota) & = & \mathsf{id}_{\iota} & \mathit{coerce}^{\pm}_{\alpha}(\star) & = & \mathsf{id}_{\star} & \mathit{coerce}^{\pm}_{\alpha}(\forall X.A) & = & \forall X.(\mathit{coerce}^{\pm}_{\alpha}(A)) \\
\mathit{coerce}^{\pm}_{\alpha}(X) & = & \mathsf{id}_{X} & \mathit{coerce}^{\pm}_{\alpha}(\alpha) & = & \alpha^{\pm} & \mathit{coerce}^{\pm}_{\alpha}(\beta) & = & \mathsf{id}_{\beta} \quad (\text{if } \beta \neq \alpha)
\end{array}
$$
$$
\mathit{coerce}^{\pm}_{\alpha}(A \to B) = \mathit{coerce}^{\mp}_{\alpha}(A) \to \mathit{coerce}^{\pm}_{\alpha}(B)
$$

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

$$
\begin{array}{rcll}
k_1\, k_2 & \longrightarrow & \delta(k_1, k_2) & (\text{R\_Delta\_C}) \\
(\lambda x\!:\!A.M)\, V & \longrightarrow & M[x := V] & (\text{R\_Beta\_C}) \\
V\langle \mathsf{id}_A \rangle & \longrightarrow & V & (\text{R\_Id\_C}) \\
(V\langle c \to d\rangle)\, V' & \longrightarrow & (V\,(V'\langle c\rangle))\langle d\rangle & (\text{R\_Wrap\_C}) \\
V\langle G!\rangle\,\langle G?^p\rangle & \longrightarrow & V & (\text{R\_Collapse\_C}) \\
V\langle G!\rangle\,\langle H?^p\rangle & \longrightarrow & \mathsf{blame}\, p \;\; (G \neq H) & (\text{R\_Conflict\_C}) \\
V\langle \alpha^-\rangle\,\langle \alpha^+\rangle & \longrightarrow & V & (\text{R\_Remove\_C}) \\
V\langle c\,;d\rangle & \longrightarrow & V\langle c\rangle\,\langle d\rangle & (\text{R\_Split\_C}) \\
V\langle \bot^{p}_{A \rightsquigarrow B}\rangle & \longrightarrow & \mathsf{blame}\, p & (\text{R\_Fail\_C}) \\
(\Lambda X.(M : A_0))\overline{\langle \forall X.c\rangle}\,\star & \longrightarrow & (M\overline{\langle c\rangle})[X := \star] & (\text{R\_TybetaDyn\_C})
\end{array}
$$

$$
\frac{\Sigma \vdash \overline{\langle \forall X.c\rangle} : \forall X.A_0 \rightsquigarrow \forall X.A_n}{\begin{array}{l}\Sigma \triangleright (\Lambda X.(M : A_0))\overline{\langle \forall X.c\rangle}\,\mathbb{B} \\ \qquad\qquad \longrightarrow \Sigma, \alpha := \mathbb{B} \triangleright ((M\overline{\langle c\rangle})[X := \alpha])\langle \mathit{coerce}^{+}_{\alpha}(A_n[X := \alpha])\rangle\end{array}}\;(\text{R\_Tybeta\_C})
$$

$$
\frac{\Sigma \triangleright M \longrightarrow \Sigma' \triangleright M'}{\Sigma \triangleright E[M] \longrightarrow \Sigma' \triangleright E[M']}\;(\text{R\_Ctx\_C}) \qquad E[\mathsf{blame}\, p] \longrightarrow \mathsf{blame}\, p \;(\text{R\_Blame\_C})
$$

Fig. 3. Polymorphic Coercion Calculus $\lambda C^{\forall}_{mp}$: Operational Semantics

parameter X in the body of the type abstraction with coercions. The latter, which is the most complex, ensures parametric behavior dynamically. First, the premise $\Sigma \vdash \overline{\langle \forall X.c\rangle} : \forall X.A_0 \rightsquigarrow \forall X.A_n$ means that there exist some types $A_1, \cdots, A_{n-1}$ such that $\Sigma \mid \emptyset \vdash \forall X.c_i : \forall X.A_{i-1} \rightsquigarrow \forall X.A_i$, for all $i \in \{1, \ldots, n\}$. The type annotations in type abstractions and failure coercions allow the type A_n to be determined uniquely. The reduction step generates a fresh type name α, extends the store Σ by $\alpha := \mathbb{B}$, substitutes α for X in the bodies of the type abstraction and all the coercions at once. Furthermore, a coercion $\mathit{coerce}^{+}_{\alpha}(A_n[X := \alpha])$ is applied; it is a coercion from $A_n[X := \alpha]$, the type of $(M\overline{\langle c\rangle})[X := \alpha]$, to $A_n[X := \mathbb{B}]$, the type of the original term. The two auxiliary functions $\mathit{coerce}^{+}_{\alpha}(A)$ and $\mathit{coerce}^{-}_{\alpha}(A)$, defined at the top of Figure 3, generate—if $\alpha := \mathbb{B}$ is in Σ—coercions from A to $A[\alpha := \mathbb{B}]$ and from $A[\alpha := \mathbb{B}]$ to A, respectively.[6] For example, $(\Lambda X.(M_0 : \star \to \star))\langle \forall X.X! \to X?^p\rangle\ \mathsf{Int}\ 42$ of type $\forall X.X \to X$ reduces as follows:

$$
\begin{array}{rl}
 & \Sigma \triangleright (\Lambda X.(M_0 : \star \to \star))\langle \forall X.X! \to X?^p\rangle\ \mathsf{Int}\ 42 \\
\longrightarrow & \Sigma, \alpha := \mathsf{Int} \triangleright (M_0[X := \alpha])\langle \alpha! \to \alpha?^p\rangle\,\langle \alpha^- \to \alpha^+\rangle\, 42 \\
\longrightarrow^{*} & \Sigma, \alpha := \mathsf{Int} \triangleright (M_0[X := \alpha])\,(42\langle \alpha^-\rangle\langle \alpha!\rangle)\langle \alpha?^p\rangle\langle \alpha^+\rangle\ .
\end{array}
$$

If $(M_0[X := \alpha])\,(42\langle \alpha^-\rangle\langle \alpha!\rangle)$ returns $42\langle \alpha^-\rangle\,\langle \alpha!\rangle$, then the reduction ends at 42 by canceling $\alpha!$ by $\alpha?^p$ and α^- by α^+.

Remark: The whole sequence of universal coercions is processed in one step, unlike function coercions applied to an abstraction. This behavior, which can be found elsewhere [New et al. 2020;

[6]Here, we write $A[\alpha := \mathbb{B}]$ for the type obtained by replacing α in A with $\mathbb{B}$.

Toro et al. 2019], means that the bound type variables in the sequence are considered the same. Readers are referred to Ozaki et al. [2021] for more detailed discussions on why the sequence has to be processed at once.

3.4 Basic Properties

The calculus $\lambda C^{\forall}_{mp}$ enjoys determinacy of reduction (Theorem 3.1) and type safety (Theorem 3.2).

To show the determinacy of reduction, we extend α-equivalence to $\Sigma \triangleright M$ in a straightforward manner—by considering type names defined in Σ bound in M. For example, $\alpha := \mathsf{Int} \triangleright 42\langle\alpha^-\rangle$ and $\beta := \mathsf{Int} \triangleright 42\langle\beta^-\rangle$ are α-equivalent and, thus, identified.

Theorem 3.1 (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 $M_1 = M_2$.*

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

Theorem 3.2 (Type safety). *If $\Sigma \mid \emptyset \vdash M : A$, then one of the followings holds: (1) $\Sigma \triangleright M \longrightarrow^* \Sigma' \triangleright V$ for some store Σ' and value V such that $\Sigma' \mid \emptyset \vdash V : A$; (2) $\Sigma \triangleright M \longrightarrow^* \Sigma' \triangleright \mathsf{blame}\, p$ for some store Σ' and blame label p; or (3) $\Sigma \triangleright M \Uparrow$.*

3.5 Parametricity

The calculus $\lambda C^{\forall}_{mp}$ is *mostly parametric* in that, given a polymorphic value V, type applications $V\,A$ and $V\,B$ behave in the same way if neither A nor B is $\star$ (or, trivially, both are $\star$). In other words, if, say, $A \neq \star$ and $B = \star$, $V\,A$ and $V\,B$ may behave differently because $V\,A$ prevents V from inspecting the actual type A by dynamic sealing while $V\,\star$ does not. For instance, we can find such non-parametric behavior when $V = \Lambda X.0\langle\mathsf{Int!}\rangle\langle X?^p\rangle$ because then $\emptyset \triangleright V\,\mathbb{A} \longrightarrow \alpha := \mathbb{A} \triangleright 0\langle\mathsf{Int!}\rangle\langle\alpha?^p\rangle \longrightarrow \mathsf{blame}\, p$ while $\emptyset \triangleright V\,\star \longrightarrow \emptyset \triangleright 0\langle\mathsf{Int!}\rangle\langle\mathsf{id}_\star\rangle \longrightarrow 0\langle\mathsf{Int!}\rangle$.

To state this "mostly parametric" nature of $\lambda C^{\forall}_{mp}$ formally, we give a step-indexed Kripke logical relation. Due to the space limitation, the paper only explains the crux of the logical relation; see the supplementary material for details. Our logical relation is defined in a way similar to that for a polymorphic blame calculus given by Ahmed et al. [2017] except for the treatment of polymorphic values. Ahmed et al.'s logical relation relates closed values V_1 and V_2 at a polymorphic type $\forall X.A$ if, for any types B_1 and B_2, (the one-step reduction results of) type applications $V_1\,B_1$ and $V_2\,B_2$ are related at the type A under an arbitrary relational interpretation for the bound type variable X. By contrast, our logical relation relates V_1 and V_2 at $\forall X.A$ if either of the following holds: for any non-dynamic types $\mathbb{B}_1$ and $\mathbb{B}_2$, (the one-step reduction results of) type applications $V_1\,\mathbb{B}_1$ and $V_2\,\mathbb{B}_2$ are related at the type A under any relational interpretation for X; or $V_1\,\star$ and $V_2\,\star$ are related at the type $A[X := \star]$. Note that the relational interpretation in relating $V_1\,\star$ and $V_2\,\star$ is *not* arbitrary (in fact, it is fixed to the value relation of the type $\star$ implicitly). This is because some polymorphic values can distinguish between different interpretations. For instance, let $V = \Lambda X.\lambda x{:}X.x\langle X!\rangle\langle\mathsf{Int?}^p\rangle$. Then, $V\,\star\,(0\langle\mathsf{Int!}\rangle) \longrightarrow^* 0$ while $V\,\star\,(\mathsf{true}\langle\mathsf{Bool!}\rangle) \longrightarrow^* \mathsf{blame}\, p$. Therefore, the value V can distinguish between the interpretations $0\langle\mathsf{Int!}\rangle$ and $\mathsf{true}\langle\mathsf{Bool!}\rangle$ for the type variable X.

We extend the logical relation over closed terms to open terms, written as $\Sigma \mid \Gamma \vdash M_1 \approx M_2 : A$, and prove that the fundamental property and the soundness with respect to contextual equivalence $\Sigma \mid \Gamma \vdash M_1 \overset{\mathrm{ctx}}{=} M_2 : A$ (see the supplementary material for the formal definition).

Theorem 3.3 (Fundamental Property). *If $\Sigma \mid \Gamma \vdash M : A$, then $\Sigma \mid \Gamma \vdash M \approx M : A$.*

Theorem 3.4 (Soundness w.r.t. Contextual Equivalence). *If $\Sigma \mid \Gamma \vdash M_1 \approx M_2 : A$, then $\Sigma \mid \Gamma \vdash M_1 \overset{\mathrm{ctx}}{=} M_2 : A$.*

The logical relation is also useful as a tool to reason about the behavior of programs. Although no usual free theorem [Wadler 1989] is available in $\lambda C^{\forall}_{mp}$ because the behavior of polymorphic values may depend on type arguments there, we can derive theorems that predict how polymorphic values behave when they are applied to non-dynamic types. For example, the following theorem states that any closed value of type $\forall X.\forall Y.X \to Y \to X$ behaves as the K combinator, raises blame, or diverges. It can be proven using the logical relation, as Ahmed et al. [2017] did.

Theorem 3.5 (Free Theorem: K-Combinator). *If $\Sigma \mid \emptyset \vdash V : \forall X.\forall Y.X \to Y \to X$ and $\Sigma \mid \emptyset \vdash V_1 : \mathbb{A}$ and $\Sigma \mid \emptyset \vdash V_2 : \mathbb{B}$, then one of the following holds: (1) $\Sigma \triangleright V\,\mathbb{A}\,\mathbb{B}\,V_1\,V_2 \longrightarrow^{*} \Sigma' \triangleright V'_1$ and $\Sigma' \mid \emptyset \vdash V'_1 \overset{\mathrm{ctx}}{=} V_1 : \mathbb{A}$ for some Σ' and V'_1; (2) $\Sigma \triangleright V\,\mathbb{A}\,\mathbb{B}\,V_1\,V_2 \longrightarrow^{*} \Sigma' \triangleright \mathsf{blame}\,p$ for some Σ' and p; or (3) $\Sigma \triangleright V\,\mathbb{A}\,\mathbb{B}\,V_1\,V_2 \Uparrow$.*

4 SPACE-EFFICIENT POLYMORPHIC COERCION CALCULUS $\lambda S^{\forall}_{mp}$

This section presents yet another coercion calculus $\lambda S^{\forall}_{mp}$. It works as a space-efficient implementation of $\lambda C^{\forall}_{mp}$ by eagerly normalizing a sequence of consecutively applied coercions into a canonical form. As we detail later, concealment and revelation make it complicated to discuss space efficiency formally; thus, they are implicit in $\lambda S^{\forall}_{mp}$, as in some of the early calculi for polymorphic gradual typing [Ahmed et al. 2011; Igarashi et al. 2017]. We also state that $\lambda S^{\forall}_{mp}$ implements $\lambda C^{\forall}_{mp}$ correctly by providing a type- and semantics-preserving translation from $\lambda C^{\forall}_{mp}$ to $\lambda S^{\forall}_{mp}$.

4.1 Space-Efficient Coercions for Mostly Parametric Polymorphism

$\lambda S^{\forall}_{mp}$ inherits space-efficient coercions, ranged over by s and t, from λS [Siek et al. 2015a, 2021]. Space-efficient coercions can be specified by the following regular schema (regular expression constructors are displayed in red):

$$(\ G_1?^p;\ )^{?}\ (\ \bot^p \mid (g\ (\ ;G_2!\ )^{?}))\ .$$

A space-efficient coercion starts with an optional projection coercion $G_1?^p$, followed by either a failure coercion $\bot^p$ or a ground coercion g—an identity coercion, a function coercion, or a universal coercion—followed by an optional injection coercion $G_2!$.

One fundamental property of space-efficient coercions is that the composition of two space-efficient coercions s and t (where the target type of first is the source of the other) can always be expressed by another space-efficient coercion. To express such a composed coercion, we introduce (a meta-level function) $s \mathbin{⨾} t$. For example, let $s = \mathsf{id}_{\mathsf{Int}}\ ;\mathsf{Int}! : \mathsf{Int} \rightsquigarrow \star$, which adds the injection tag $\mathsf{Int}!$ to an integer, and $t = \mathsf{Int}?^p\ ;\mathsf{id}_{\mathsf{Int}} : \star \rightsquigarrow \mathsf{Int}$, which checks if the value has $\mathsf{Int}!$ and removes it. Then, $s \mathbin{⨾} t = \mathsf{id}_{\mathsf{Int}}$, by canceling tagging followed by untagging. However, we have $s \mathbin{⨾} t' = \bot^p$ for $t' = \mathsf{Bool}?^p\ ;\mathsf{id}_{\mathsf{Bool}}$, because the tags added by s and checked by t' do not match. In general,

$$(g_1\ ;G_2!) \mathbin{⨾} (H_1?^q; g_2) = \begin{cases} g_1 \mathbin{⨾} g_2 & (\text{if } G_2 = H_1) \\ \bot^q & (\text{if } G_2 \neq H_1) \end{cases}$$

holds in the simply-typed setting, where G and H are concrete types.

Polymorphic typing, however, complicates this meta-level composition because injection and projection can involve type variables. The question here is what the composition of, say, $\forall X.\mathsf{id}_X\ ;X!$ (from $\forall X.X$ to $\forall X.\star$) and $\forall X.\mathsf{Int}?^p\ ;\mathsf{id}_{\mathsf{Int}}$ (from $\forall X.\star$ to $\forall X.\mathsf{Int}$) should be. In fact, the answer depends on how these variables are instantiated at run time. Since, in $\lambda C^{\forall}_{mp}$, a type variable is instantiated with either a fresh type name or $\star$, there are two cases:

(1) In the case where X is instantiated with $\star$, the first coercion behaves as the identity coercion $\mathsf{id}_\star$ and the composition will be $\mathsf{Int}?^p\ ;\mathsf{id}_{\mathsf{Int}}$;

(2) In the case where X is instantiated with a type name α, the composition will be $\perp^p$ because α and Int cannot be the same.

It is unknown, however, which case will happen at run time and, thus, we have to keep both versions in a single coercion.

To keep different versions that arise after type substitution, we extend the syntax of the universal coercions to $\forall X.s \,,, \, t$.[7] The coercion s represents the behavior where X is instantiated with a type name and the second the behavior where X is with $\star$. For example, $\forall X.\mathsf{id}_X \,;\, X!$ and $\forall X.\mathsf{Int}?^p \,;\, \mathsf{id}_{\mathsf{Int}}$ are actually written $\forall X.((\mathsf{id}_X \,;\, X!) \,,, \, (\mathsf{id}_\star))$ and $\forall X.((\mathsf{Int}?^p \,;\, \mathsf{id}_{\mathsf{Int}}) \,,, \, (\mathsf{Int}?^p \,;\, \mathsf{id}_{\mathsf{Int}}))$, respectively, and we can compose them as follows.

$$\begin{array}{rl} & (\forall X.((\mathsf{id}_X \,;\, X!) \,,, \, (\mathsf{id}_\star))) \fatsemi (\forall X.((\mathsf{Int}?^p \,;\, \mathsf{id}_{\mathsf{Int}}) \,,, \, (\mathsf{Int}?^p \,;\, \mathsf{id}_{\mathsf{Int}}))) \\ = & \forall X.((\mathsf{id}_X \,;\, X!) \fatsemi (\mathsf{Int}?^p \,;\, \mathsf{id}_{\mathsf{Int}})) \,,, \, (\mathsf{id}_\star \fatsemi (\mathsf{Int}?^p \,;\, \mathsf{id}_{\mathsf{Int}})) \\ = & \forall X.(\perp^p \,,, \, (\mathsf{Int}?^p \,;\, \mathsf{id}_{\mathsf{Int}})) \end{array}$$

If X is instantiated with a type name later, $\perp^p$ will be chosen; otherwise, $\mathsf{Int}?^p \,;\, \mathsf{id}_{\mathsf{Int}}$ will be chosen.

The extension relies on the fact that only a type name or $\star$ is substituted for a type variable and different type variables cannot be instantiated with the same type name. If the semantics were based on type substitution $[X := A]$ of arbitrary types to discard mostly parametric semantics, we might have to list all the possible behaviors in a universal coercion, which can be infinite.

We admit that the extended syntax for universal coercions is a little naive in terms of their sizes because each universal quantifier always involves two coercions of similar forms. In fact, as we will see, the translation of $\lambda\mathrm{C}^\forall_{mp}$ coercions into $\lambda\mathrm{S}^\forall_{mp}$ ones can cause an exponential blow-up of the size of a coercion, although the blow-up is statically bounded by the number of (nested) universal coercions appearing in $\lambda\mathrm{C}^\forall_{mp}$ programs and unbounded growth *during reduction* will be prevented. An alternative design is to associate a type variable with $s \,,, \, t$ to express which type variable controls the choice between two coercions, making it possible to push "$,,$" towards leaves: For example, $\forall X.(\mathsf{Int}! \rightarrow X!) \,,, \, (\mathsf{Int}! \rightarrow \mathsf{id}_\star)$ could be represented as $\forall X.\mathsf{Int}! \rightarrow (X! \overset{X}{,,} \mathsf{id}_\star)$, where $X! \overset{X}{,,} \mathsf{id}_\star$ means either $X!$—if X is instantiated with a type name—or $\mathsf{id}_\star$—if X is instantiated with $\star$. Although exponential blow-ups can be avoided in many cases, the worst-case complexity is not clear. Moreover, we expect that this alternative would complicate the coercion composition $s \fatsemi t$. Detailed investigation for better alternatives is left for future work.

4.2 Syntax and Type System

The top of Figure 4 gives the syntax of $\lambda\mathrm{S}^\forall_{mp}$. It shows the same constructors as those in $\lambda\mathrm{C}^\forall_{mp}$ in gray.

$\lambda\mathrm{S}^\forall_{mp}$ removes type annotations from failure coercions (and also type abstractions), as the semantics does not use coercion generation functions any longer. Similarly, identity coercions omit type information for simplicity. We use b and i for coercions following the regular schemata $\perp^p \mid (g\ (\ ;\, G_2!\ )^?)$ and $g\ (\ ;\, G_2!\ )^?$ and call them possibly blaming and intermediate coercion, respectively. As we mentioned earlier, we remove concealment and revelation. In $\forall X.s \,,, \, t$, the type variable X is bound (only) in s.

Terms are defined similarly to $\lambda\mathrm{C}^\forall_{mp}$, except that a coercion application always involves a space-efficient coercion. Values are uncoerced values possibly with a *single* coercion which takes one of the following forms: $g \,;\, G!$, $s \rightarrow t$, and $\forall X.s \,,, \, t$, which are non-identity intermediate coercions. Since $\lambda\mathrm{S}^\forall_{mp}$ normalizes a sequence of consecutively applied coercions eagerly, a value is wrapped by at most one coercion and the hole in a frame never appears under coercion applications.

[7]The operator symbol "$,,$" is derived from the merge operator [Huang and d. S. Oliveira 2020; Reynolds 1988] for intersection types.

Syntax

Space-efficient coercions	s, t	$::=$	$G?^p \,;\, b \mid b$
Possibly blaming coercions	b	$::=$	$\bot^p \mid i$
Intermediate coercions	i, j	$::=$	$g \,;\, G! \mid g$
Ground coercions	g, h	$::=$	$\mathsf{id} \mid s \to t \mid \forall X.s \,„\, t$
Terms	M	$::=$	$x \mid U \mid M\,M \mid M\,A \mid M\langle s\rangle \mid \mathsf{blame}\,p$
Uncoerced values	U	$::=$	$k \mid \lambda x\!:\!A.M \mid \Lambda X.M$
Values	V	$::=$	$U \mid U\langle g \,;\, G!\rangle \mid U\langle s \to t\rangle \mid U\langle \forall X.s \,„\, t\rangle$
Frames	E	$::=$	$\square\, M \mid V\, \square \mid \square\, A$
Type environments	Γ	$::=$	$\emptyset \mid \Gamma, x : A \mid \Gamma, X$
Stores	Σ	$::=$	$\emptyset \mid \Sigma, \alpha := \mathbb{A}$

Coercion typing $\boxed{\Sigma \mid \Gamma \vdash s : A \rightsquigarrow B}$

$$\frac{\vdash \Sigma \quad \emptyset \vdash \Gamma \quad \Sigma \mid \Gamma \vdash A}{\Sigma \mid \Gamma \vdash \mathsf{id} : \Sigma(A) \rightsquigarrow \Sigma(A)}\ (\textsc{Ct\_Id\_S}) \qquad \frac{\Sigma \mid \Gamma \vdash g : A \rightsquigarrow \Sigma(G) \quad \Sigma \mid \Gamma \vdash G}{\Sigma \mid \Gamma \vdash g \,;\, G! : A \rightsquigarrow \star}\ (\textsc{Ct\_Inj\_S})$$

(A is neither a function nor a universal type)

$$\frac{\Sigma \mid \Gamma, X \vdash s : A \rightsquigarrow B \quad \Sigma \mid \Gamma \vdash t : A[X := \star] \rightsquigarrow B[X := \star]}{\Sigma \mid \Gamma \vdash \forall X.s \,„\, t : \forall X.A \rightsquigarrow \forall X.B}\ (\textsc{Ct\_All\_S})$$

Term typing $\boxed{\Sigma \mid \Gamma \vdash M : A}$

$$\frac{\Sigma \mid \Gamma, x : \Sigma(A) \vdash M : B}{\Sigma \mid \Gamma \vdash \lambda x\!:\!A.M : \Sigma(A) \to B}\ (\textsc{T\_Abs\_S}) \qquad \frac{\Sigma \mid \Gamma \vdash M_1 : A \to B \quad \Sigma \mid \Gamma \vdash M_2 : A}{\Sigma \mid \Gamma \vdash M_1\, M_2 : B}\ (\textsc{T\_App\_S})$$

Fig. 4. $\lambda\mathsf{S}^{\forall}_{mp}$: Syntax and Typing Rules (excerpt)

We extend type substitution as follows (only the interesting cases are shown):

$$(G?^p \,;\, b)[X := \star] \quad = \quad \begin{cases} b[X := \star] & (\text{if } G = X) \\ G?^p \,;\, (b[X := \star]) & (\text{if } G \neq X) \end{cases}$$

$$(g \,;\, G!)[X := \star] \quad = \quad \begin{cases} g[X := \star] & (\text{if } G = X) \\ (g[X := \star]) \,;\, G! & (\text{if } G \neq X)\,. \end{cases}$$

The type system of $\lambda\mathsf{S}^{\forall}_{mp}$ is slightly different from that of $\lambda\mathsf{C}^{\forall}_{mp}$. To dispense with concealment and revelation coercions, we identify a type name α and the type to which it is bound ($\Sigma(\alpha)$, defined below), as in Ahmed et al. [2011]; Igarashi et al. [2017]. For example, under $\Sigma = \alpha := \mathsf{Int}$, term $\lambda x\!:\!\alpha.x + 1$ is well typed and given type $\mathsf{Int} \to \mathsf{Int}$, despite that the variable x, which is declared to have type α, is directly passed to where an Int is expected. We show a few representative typing rules at the bottom of Figure 4. In what follows, $\Sigma(A)$ denotes the name-free type obtained by replacing type names in A with the corresponding types associated by store Σ. Formally, it is defined as follows:

$$\emptyset(A) = A \qquad\qquad (\Sigma, \alpha := \mathbb{B})(A) = \Sigma(A[\alpha := \mathbb{B}])\,.$$

We also write $\Sigma(\Gamma)$ for the type environment obtained by mapping every binding $x : A$ in type environment Γ to $x : \Sigma(A)$.

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

$$\begin{array}{rcll}
k_1\, k_2 & \longrightarrow & \delta(k_1, k_2) & \text{(R\_DELTA\_S)} \\
(\lambda x{:}A.M)\; V & \longrightarrow & M[x := V] & \text{(R\_BETA\_S)} \\
U\langle \mathsf{id} \rangle & \longrightarrow & U & \text{(R\_ID\_S)} \\
(U\langle s \to t\rangle)\; V & \longrightarrow & (U\,(V\langle s\rangle))\langle t\rangle & \text{(R\_WRAP\_S)} \\
U\langle \bot^p \rangle & \longrightarrow & \mathsf{blame}\, p & \text{(R\_FAIL\_S)} \\
M\langle s\rangle\,\langle t\rangle & \longrightarrow & M\langle s \fatsemi t\rangle & \text{(R\_MERGE\_S)} \\
(\Lambda X.M)\,\star & \longrightarrow & M[X := \star] & \text{(R\_TYBETADYN\_S)} \\
(\Lambda X.M)\langle \forall X.s \,,, t\rangle\,\star & \longrightarrow & (M\langle t\rangle)[X := \star] & \text{(R\_TYBETADYNC\_S)} \\
\Sigma \triangleright (\Lambda X.M)\;\mathbb{A} & \longrightarrow & \Sigma, \alpha := \mathbb{A} \triangleright M[X := \alpha] & \text{(R\_TYBETA\_S)} \\
 & & \quad \text{where } \alpha \notin \mathrm{dom}(\Sigma) & \\
\Sigma \triangleright (\Lambda X.M)\langle \forall X.s \,,, t\rangle\;\mathbb{A} & \longrightarrow & \Sigma, \alpha := \mathbb{A} \triangleright (M\langle s\rangle)[X := \alpha] & \text{(R\_TYBETAC\_S)} \\
 & & \quad \text{where } \alpha \notin \mathrm{dom}(\Sigma) & \\
E[\mathsf{blame}\, p] & \longrightarrow & \mathsf{blame}\, p & \text{(R\_BLAMEE\_S)} \\
(\mathsf{blame}\, p)\langle s\rangle & \longrightarrow & \mathsf{blame}\, p & \text{(R\_BLAMEC\_S)}
\end{array}$$

$$\frac{\Sigma \triangleright M \longrightarrow \Sigma' \triangleright M'}{\Sigma \triangleright E[M] \longrightarrow \Sigma' \triangleright E[M']}\ \text{(R\_CTXE\_S)} \qquad \frac{\Sigma \triangleright M \longrightarrow \Sigma' \triangleright M'}{\Sigma \triangleright M\langle s\rangle \longrightarrow \Sigma' \triangleright M'\langle s\rangle}\ \text{(R\_CTXC\_S)}$$

(M is not a coercion application.)

Fig. 5. $\lambda S^{\forall}_{mp}$: Operational Semantics

The typing for space-efficient coercions is mostly straightforward. The only interesting rule is (CT_ALL_S) for universal coercions: The second coercion t, which represents the case where X is $\star$, is required to be from $A[X := \star]$ to $B[X := \star]$. The typing judgment for terms takes the same form $\Sigma \mid \Gamma \vdash M : A$ as $\lambda C^{\forall}_{mp}$ but type names do not appear in Γ nor A (while they *do* appear in M). A similar convention is applied to coercion typing judgments as well.

4.3 Operational Semantics

The semantics of $\lambda S^{\forall}_{mp}$ is defined by the four-place reduction relation $\Sigma_1 \triangleright M_1 \longrightarrow \Sigma_2 \triangleright M_2$, which means that the one-step reduction of term M_1 with store Σ_1 produces term M_2 and store Σ_2. Formally, they are the smallest relations satisfying the reduction rules in Figure 5.

Most of the reduction rules are the same as those of $\lambda C^{\forall}_{mp}$ (Figure 3) except for the syntactic difference explained in Section 4.2. The reduction rules for type applications assume that the value applied to a type is a type abstraction possibly with a single universal coercion because nested coercion applications are not values in $\lambda S^{\forall}_{mp}$. Note that, unlike $\lambda C^{\forall}_{mp}$, coercion generation is not needed because concealment and revelation are implicit in $\lambda S^{\forall}_{mp}$.

The new reduction rule is (R_MERGE_S), which normalizes the composition $s\,;\,t$ of consecutively applied coercions s and t into a single coercion $s \fatsemi t$. It does not require coerced terms to be values because coercions are normalized before they reduce. The coercion normalization is implemented by the meta-level composition operation $s \fatsemi t$, which is defined below.

Definition 4.1 (Coercion Composition). For space-efficient coercions s and t, a space-efficient coercion $s \fatsemi t$ is defined as follows.

$$\begin{array}{rcl@{\qquad}rcl}
(G?^p\,;b) \fatsemi t & = & G?^p\,;(b \fatsemi t) & (g\,;G!) \fatsemi (G?^p\,;b) & = & g \fatsemi b \\
\bot^p \fatsemi t & = & \bot^p & (g\,;G!) \fatsemi (H?^p\,;b) & = & \bot^p \quad (\text{if } G \neq H) \\
i \fatsemi \bot^{p'} & = & \bot^{p'} & \multicolumn{3}{l}{\mathsf{id} \fatsemi t = t \quad (\text{if } t \neq \bot^{p'} \wedge t \neq (h\,;H!) \wedge t \neq \mathsf{id})} \\
i \fatsemi (h\,;H!) & = & (i \fatsemi h)\,;H! & (s \to t) \fatsemi (s' \to t') & = & (s' \fatsemi s) \to (t \fatsemi t') \\
i \fatsemi \mathsf{id} & = & i & (\forall X.s_1 \,„\, s_2) \fatsemi (\forall X.t_1 \,„\, t_2) & = & \forall X.(s_1 \fatsemi t_1) \,„\, (s_2 \fatsemi t_2) \\
 & & & s \fatsemi t & & \text{is undefined otherwise.}
\end{array}$$

The normalization of $s \fatsemi t$ proceeds as follows. (Here, we assume that the target type of s and the source type of t agree.) First, the projection in s must remain in the resulting coercion, because it is the first check to be performed when the coercion is applied. If s is of the form $G?^p\,;\bot^q$, applying the composition of s and t fails at either the projection $G?^p$ or blame q, whatever t is. Thus, $s \fatsemi t$ results in s. If s is an intermediate coercion i, there are four cases depending on the form of t:

(1) if t is a failure $\bot^p$, the composition $i \fatsemi t$ is $\bot^p$ because i does not raise blame immediately;
(2) if t is an intermediate coercion ending with an injection $H!$, the intermediate coercion on the left is pushed through the formal composition, leaving $H!$ as it is;
(3) if t is an identity coercion, i is naturally the result of composition; and
(4) otherwise, t is $H?^p\,;b$, $t_1 \to t_2$, or $\forall X.t_0$; and we proceed with case analysis on i.
   (a) If i is an identity, the composition returns t;
   (b) If i is $g\,;G!$, then t must be $H?^p\,;b$ and the composition results in either $g \fatsemi b$ (if G and H are equal) or $\bot^p$ (otherwise), corresponding to (R_Collapse_C) and (R_Conflict_C) of $\lambda C^{\forall}_{mp}$;
   (c) If i is a function coercion, t must also be a function coercion and the composition is recursively called (the order of composition is reversed for coercions for arguments).
   (d) If i is a universal coercion, t must also be universal and the composition is recursively called.

Because frames in $\lambda S^{\forall}_{mp}$ are restricted, blame lifting and subterm reduction are split into four rules. (R_BlameE_S) and (R_BlameC_S) mean that blame is lifted, no matter whether it is surrounded by an ordinary frame or a coercion application. The direct subterm of a coercion application is reduced by (R_CtxC_S), if it is not a (nested) coercion application. Otherwise, a subterm is reduced by (R_CtxE_S). The reduction of a nested coercion application starts with normalizing the sequence of the applied coercions, and then proceeds to the reduction of the coerced term.

Example. We illustrate an example of reduction in $\lambda S^{\forall}_{mp}$. Here, we add type annotations to id (as in $\lambda C^{\forall}_{mp}$), so that their source/target types are obvious.

Consider space-efficient coercions

$$\begin{aligned}
s_1 &= \forall X.(\mathsf{id}_X \to (\mathsf{id}_X\,;X!)) \,„\, (\mathsf{id}_\star \to \mathsf{id}_\star) \\
s_2 &= \forall X.(\mathsf{id}_X \to (\mathsf{Int}?^p\,;\mathsf{id}_{\mathsf{Int}})) \,„\, (\mathsf{id}_\star \to (\mathsf{Int}?^p\,;\mathsf{id}_{\mathsf{Int}}))\,,
\end{aligned}$$

which correspond to $\lambda C^{\forall}_{mp}$ coercions $\forall X.\mathsf{id}_X \to (\mathsf{id}_X\,;X!)$ and $\forall X.\mathsf{id}_X \to (\mathsf{Int}?^p\,;\mathsf{id}_{\mathsf{Int}})$, respectively, and $V = \Lambda X.\lambda x\!:\!X.x$. Let V_1 be a term of type A. Since

$$\begin{array}{rl}
 & (\forall X.(\mathsf{id}_X \to (\mathsf{id}_X\,;X!)) \,„\, (\mathsf{id}_\star \to \mathsf{id}_\star)) \\
 & \fatsemi (\forall X.(\mathsf{id}_X \to (\mathsf{Int}?^p\,;\mathsf{id}_{\mathsf{Int}})) \,„\, (\mathsf{id}_\star \to (\mathsf{Int}?^p\,;\mathsf{id}_{\mathsf{Int}}))) \\
= & \forall X.(\quad (\mathsf{id}_X \to (\mathsf{id}_X\,;X!)) \fatsemi (\mathsf{id}_X \to (\mathsf{Int}?^p\,;\mathsf{id}_{\mathsf{Int}})) \\
 & \qquad „\, ((\mathsf{id}_\star \to \mathsf{id}_\star) \fatsemi (\mathsf{id}_\star \to (\mathsf{Int}?^p\,;\mathsf{id}_{\mathsf{Int}})))) \\
= & \forall X.((\mathsf{id}_X \to \bot^p) \,„\, (\mathsf{id}_\star \to \mathsf{Int}?^p\,;\mathsf{id}_{\mathsf{Int}}))\,,
\end{array}$$

the term $V\langle s_1\rangle\ \langle s_2\rangle\ A\ V_1$ (under the empty store) reduces as follows:

$$\begin{array}{l}
\emptyset \triangleright V\langle s_1\rangle\langle s_2\rangle\, A\, V_1 \longrightarrow \emptyset \triangleright V\langle s_1 \fatsemi s_2\rangle\, A\, V_1 \\
\qquad\qquad\qquad = \ \emptyset \triangleright V\langle \forall X.(\mathsf{id}_X \to \bot^p) \mathbin{,\!,} (\mathsf{id}_\star \to (\mathsf{Int}?^p\, ;\, \mathsf{id}_{\mathsf{Int}}))\rangle\, A\, V_1
\end{array}$$

$$\begin{array}{ll}
\qquad\qquad (\text{if } A = \star) \swarrow & \qquad\qquad \searrow (\text{if } \exists \mathbb{A}. A = \mathbb{A}) \\
\quad \emptyset \triangleright (\lambda x{:}\star.x)\langle \mathsf{id}_\star \to (\mathsf{Int}?^p\, ;\, \mathsf{id}_{\mathsf{Int}})\rangle\ V_1 & \quad \alpha := \mathbb{A} \triangleright (\lambda x{:}\alpha.x)\langle \mathsf{id}_\alpha \to \bot^p\rangle\ V_1 \\
\longrightarrow \emptyset \triangleright ((\lambda x{:}\star.x)\ (V_1\langle \mathsf{id}_\star\rangle))\langle \mathsf{Int}?^p\, ;\, \mathsf{id}_{\mathsf{Int}}\rangle & \longrightarrow \alpha := \mathbb{A} \triangleright ((\lambda x{:}\alpha.x)\ (V_1\langle \mathsf{id}_\alpha\rangle))\langle \bot^p\rangle \\
\longrightarrow \emptyset \triangleright ((\lambda x{:}\star.x)\ V_1)\langle \mathsf{Int}?^p\, ;\, \mathsf{id}_{\mathsf{Int}}\rangle & \longrightarrow \alpha := \mathbb{A} \triangleright ((\lambda x{:}\alpha.x)\ V_1)\langle \bot^p\rangle \\
\longrightarrow \emptyset \triangleright V_1\langle \mathsf{Int}?^p\, ;\, \mathsf{id}_{\mathsf{Int}}\rangle & \longrightarrow \alpha := \mathbb{A} \triangleright V_1\langle \bot^p\rangle \\
\longrightarrow^* \begin{cases} \emptyset \triangleright k & (\text{if } \exists k. V_1 = k\langle \mathsf{id}_{\mathsf{Int}}\, ;\, \mathsf{Int}!\rangle) \\ \emptyset \triangleright \mathsf{blame}\, p & (\text{otherwise}) \end{cases} & \longrightarrow \alpha := \mathbb{A} \triangleright \mathsf{blame}\, p\,.
\end{array}$$

If V_1 is an integer cast to $\star$ and A is $\star$, it returns the integer; otherwise, it raises blame p. Notably, the coercion $s_1 \fatsemi s_2$ is smaller than the sum of the sizes of s_1 and s_2.

4.4 Basic Properties

$\lambda\mathrm{S}^{\forall}_{mp}$ satisfies the same basic properties, namely determinacy of reduction and type safety, as $\lambda\mathrm{C}^{\forall}_{mp}$. We omit their statements for brevity; interested readers are referred to the supplementary material.

4.5 Space Efficiency

Following Herman et al. [2007, 2010], we can show that $\lambda\mathrm{S}^{\forall}_{mp}$ is space-efficient in the sense that the size of every coercion emerging at run time is bounded by a certain constant derived from the initial term. A key property of space-efficient coercions to prove this property is that the composition operator $s \fatsemi t$ does not produce a coercion larger than s and t. The theorem can be proved by combining this property with the fact that coercions that emerge during reduction are the result of either composition or substitution of a type name or $\star$ for a type variable.

We first formally define the functions size(s) and height(s) to denote the size and height of a space-efficient coercion s, respectively, and state the theorem.

$$\begin{array}{rlrl}
\mathrm{size}(G?^p\, ;\, b) &= \mathrm{size}(b) + 2 & \mathrm{height}(G?^p\, ;\, b) &= \mathrm{height}(b) \\
\mathrm{size}(\bot^p) &= \mathrm{size}(\mathsf{id}) = 1 & \mathrm{height}(\bot^p) &= \mathrm{height}(\mathsf{id}) = 1 \\
\mathrm{size}(g\, ;\, G!) &= \mathrm{size}(g) + 2 & \mathrm{height}(g\, ;\, G!) &= \mathrm{height}(g) \\
\mathrm{size}(s \to t) &= \mathrm{size}(s) + \mathrm{size}(t) + 1 & \mathrm{height}(s \to t) &= \max(\mathrm{height}(s), \mathrm{height}(t)) + 1 \\
\mathrm{size}(\forall X.s \mathbin{,\!,} t) &= \mathrm{size}(s) + \mathrm{size}(t) + 1 & \mathrm{height}(\forall X.s \mathbin{,\!,} t) &= \max(\mathrm{height}(s), \mathrm{height}(t)) + 1
\end{array}$$

Theorem 4.2 ($\lambda\mathrm{S}^{\forall}_{mp}$ is Space-Efficient). *If $\emptyset \mid \emptyset \vdash M : A$ and $\emptyset \triangleright M \longrightarrow^* \Sigma' \triangleright M'$, then for any s' appearing in M', there exists some space-efficient coercion s in M such that* size$(s') \leq 5(2^{\mathrm{height}(s)} - 1)$.

Proof. The theorem follows from the following two lemmas and the fact that the size of a space-efficient coercion s is bounded by its height as size$(s) \leq 5(2^{\mathrm{height}(s)} - 1)$:

(1) If $s \fatsemi t$ is well defined, then height$(s \fatsemi t) \leq \max(\mathrm{height}(s), \mathrm{height}(t))$,
(2) If $\Sigma \triangleright M \longrightarrow^* \Sigma' \triangleright M'$, then for any s' that occurs in M', there exists some s that occurs in M and height$(s') \leq$ height(s).

The former is proved by induction on the sum of size(s) and size(t) and the latter is by induction on the derivation of $\Sigma \triangleright M \longrightarrow^* \Sigma' \triangleright M'$. □

Coercion translation $\boxed{|c|_\Gamma = s}$

$$\begin{array}{rcll}
|\mathrm{id}_A|_\Gamma & = & \mathrm{id} & (\text{if } A \text{ is } \iota, \star, X, \text{ or } \alpha) \\
|\mathrm{id}_{A\to B}|_\Gamma & = & |\mathrm{id}_A|_\Gamma \to |\mathrm{id}_B|_\Gamma & \\
|\mathrm{id}_{\forall X.A}|_\Gamma & = & \forall X.|\mathrm{id}_A|_{\Gamma,X} \mathbin{\text{„}} |\mathrm{id}_A|_\Gamma & \\
|G!|_\Gamma & = & \begin{cases} \mathrm{id} & \text{if } G = X \notin \mathrm{dom}(\Gamma) \\ |\mathrm{id}_G|_\Gamma \mathbin{;} G! & \text{otherwise} \end{cases} & \\
|G?^p|_\Gamma & = & \begin{cases} \mathrm{id} & \text{if } G = X \notin \mathrm{dom}(\Gamma) \\ G?^p \mathbin{;} |\mathrm{id}_G|_\Gamma & \text{otherwise} \end{cases} &
\end{array}
\qquad
\begin{array}{rcl}
|\alpha^-|_\Gamma & = & \mathrm{id} \\
|\alpha^+|_\Gamma & = & \mathrm{id} \\
|\bot^p_{A\rightsquigarrow B}|_\Gamma & = & \bot^p \\
|c \to d|_\Gamma & = & |c|_\Gamma \to |d|_\Gamma \\
|c \mathbin{;} d|_\Gamma & = & |c|_\Gamma \fatsemi |d|_\Gamma \\
|\forall X.c|_\Gamma & = & \forall X.|c|_{\Gamma,X} \mathbin{\text{„}} |c|_\Gamma
\end{array}$$

Term translation $\boxed{|M|_\Gamma \ = \ M'}$

$$\begin{array}{rcl}
|k|_\Gamma & = & k \\
|x|_\Gamma & = & x \\
|\lambda x\!:\!A.M|_\Gamma & = & \lambda x\!:\!A.|M|_{\Gamma,x:A}
\end{array}
\qquad
\begin{array}{rcl}
|M_1\,M_2|_\Gamma & = & |M_1|_\Gamma\,|M_2|_\Gamma \\
|\Lambda X.(M : A)|_\Gamma & = & \Lambda X.|M|_{\Gamma,X} \\
|M\,A|_\Gamma & = & |M|_\Gamma\,A
\end{array}
\qquad
\begin{array}{rcl}
|M\langle c\rangle|_\Gamma & = & |M|_\Gamma\langle |c|_\Gamma\rangle \\
|\mathsf{blame}\,p|_\Gamma & = & \mathsf{blame}\,p
\end{array}$$

Fig. 6. Translation from $\lambda C^\forall_{mp}$ to $\lambda S^\forall_{mp}$.

The removal of concealment and revelation coercions simplifies the technical development a lot: If we had concealment and revelation coercions, the reduction from $(\Lambda X.M)\langle s\rangle\, A$ would generate a coercion involving α^- and α^+, whose size is not related to the size of coercions in the initial term. Thus, we would have to investigate the size of types of subterms during reduction, which would be complicated in the presence of type substitution.

4.6 Translation from $\lambda C^\forall_{mp}$ to $\lambda S^\forall_{mp}$

This section defines a translation from $\lambda C^\forall_{mp}$ to $\lambda S^\forall_{mp}$ and proves that it is type- and semantics-preserving.

The translation $|c|_\Gamma$ for coercions is parameterized by a type environment Γ. It is defined similarly to Siek et al. [2021], except the cases involving type variables and names. Type variables that appear free in c but are not declared in Γ represent ones assumed to be bound to the dynamic type. Thus, $X!$, and $X?^p$ translate to id (from $\star$ to $\star$) if $X \notin \mathrm{dom}(\Gamma)$. The translation of a universal coercion $\forall X.c$ combines $|c|_{\Delta,X}$ and $|c|_\Delta$. Concealments and revelations are translated to identity coercions. For example, if $c_0 = Y! \to X!$ and $c = \forall X.\forall Y.c_0$, we have (again, we add type annotations to id)

$$\begin{aligned}
|c|_\emptyset &= \forall X.(|\forall Y.c_0|_X \mathbin{\text{„}} |\forall Y.c_0|_\emptyset) \\
&= \forall X.((\forall Y.|c_0|_{X,Y} \mathbin{\text{„}} |c_0|_X) \mathbin{\text{„}} (\forall Y.|c_0|_Y \mathbin{\text{„}} |c_0|_\emptyset)) \\
&= \forall X.((\forall Y.(\mathrm{id}_Y \mathbin{;} Y! \to \mathrm{id}_X \mathbin{;} X!) \mathbin{\text{„}} (\mathrm{id}_\star \to \mathrm{id}_X \mathbin{;} X!)) \mathbin{\text{„}} (\forall Y.(\mathrm{id}_Y \mathbin{;} Y! \to \mathrm{id}_\star) \mathbin{\text{„}} (\mathrm{id}_\star \to \mathrm{id}_\star)))\ .
\end{aligned}$$

Term translation is straightforward; it just translates all coercions in the given term.

We state that the translation preserves typing and semantics in the following theorems. In what follows, metavariables for terms and values in $\lambda C^\forall_{mp}$ are subscripted by c and those in $\lambda S^\forall_{mp}$ are by s. Similarly for reduction and typing judgments.

Theorem 4.3 (Translation Preserves Typing). *If* $\Sigma \mid \Gamma \vdash_C M_c : A$, *then* $\Sigma \mid \Sigma(\Gamma) \vdash_S |M_c|_\Gamma : \Sigma(A)$.

Theorem 4.4 (Correctness of Translation). *Suppose* $\Sigma \mid \emptyset \vdash_C M_c : A$.

(1) *If* $\Sigma \triangleright M_c \longrightarrow^*_C \Sigma' \triangleright V_c$, *then there exists some* V_s *such that* $\Sigma \triangleright |M_c|_\emptyset \longrightarrow^*_S \Sigma' \triangleright V_s$. *Furthermore, if* $A = \iota$, *then there exists some* k *such that* $V_c = V_s = k$.

(2) *If* $\Sigma \triangleright |M_c|_\emptyset \longrightarrow_S^* \Sigma' \triangleright V_s$*, then there exists some* V_c *such that* $\Sigma \triangleright M_c \longrightarrow_C^* \Sigma' \triangleright V_c$*. Furthermore, if* $A = \iota$*, then there exists some* k *such that* $V_c = V_s = k$*.*
(3) $\Sigma \triangleright M_c \longrightarrow_C^* \Sigma' \triangleright \mathsf{blame}\, p$ *iff* $\Sigma \triangleright |M_c|_\emptyset \longrightarrow_S^* \Sigma' \triangleright \mathsf{blame}\, p$*.*
(4) $\Sigma \triangleright M_c \Uparrow$ *iff* $\Sigma \triangleright |M_c|_\emptyset \Uparrow$*.*

To show the latter theorem, we define a relation $\Sigma \mid \Gamma \vdash M_c \approx M_s : A$. Intuitively, it means that the behavior of term M_c under $\lambda\mathrm{C}^\forall_{mp}$'s semantics is equivalent to that of M_s under $\lambda\mathrm{S}^\forall_{mp}$'s semantics. Formally, it is the smallest relation that satisfies the following rules and is compatible with respect to term constructors except for coercion applications.

$$\frac{\Sigma \mid \Gamma \vdash M_c \approx M_s : B \quad \Sigma \mid \Gamma \vdash_C c : B \rightsquigarrow A}{\Sigma \mid \Gamma \vdash M_c\langle c\rangle \approx M_s\langle |c|_\Gamma\rangle : A} \qquad \frac{\Sigma \mid \Gamma \vdash M_c \approx M_s : A \quad \Sigma \mid \emptyset \vdash_C \mathsf{id}_A : A \rightsquigarrow A}{\Sigma \mid \Gamma \vdash M_c \approx M_s\langle |\mathsf{id}_A|_\emptyset\rangle : A}$$

$$\frac{\mathrm{ftv}(s) = \emptyset \quad \Sigma \mid \Gamma \vdash M_c \approx M_s\langle s\rangle : A \quad \Sigma \mid \emptyset \vdash_C c : A \rightsquigarrow B}{\Sigma \mid \Gamma \vdash M_c\langle c\rangle \approx M_s\langle s \mathbin{⨟} |c|_\emptyset\rangle : B} \qquad \frac{\Sigma \mid \Gamma \vdash M_c \approx M_s : B \quad \Sigma \mid \emptyset \vdash_C c^I : B \rightsquigarrow A}{\Sigma \mid \Gamma \vdash M_c\langle c^I\rangle \approx M_s : A}$$

The first rule is for congruence (except that the $\lambda\mathrm{C}^\forall_{mp}$ coercion is related to its translation). The second and third rules allow a nested coercion application $M_c\langle c_1\rangle \cdots \langle c_n\rangle$ in $\lambda\mathrm{C}^\forall_{mp}$ to be related to a $\lambda\mathrm{S}^\forall_{mp}$ term $|M_c|_\emptyset\langle |c_1|_\emptyset \mathbin{⨟} \cdots \mathbin{⨟} |c_n|_\emptyset\rangle$, in which the coercion sequence $\langle c_1\rangle \cdots \langle c_n\rangle$ is normalized into a single coercion. (Note that $|\mathsf{id}_A|_\emptyset$ is the identity element of $⨟$.) The metavariable c^I used in the last rule ranges over the set of *no-op coercions*, defined by: $c^I, d^I ::= \mathsf{id}_A \mid \alpha^- \mid \alpha^+ \mid c^I \to d^I \mid \forall X.c^I \mid c^I ; d^I$. As its name suggests, no-op coercions do nothing significant because they are constructed from identity, concealments, and revelations. They are translated to coercions constructed only from the identity and, thus, in $\lambda\mathrm{S}^\forall_{mp}$, (the translation of) a no-op coercion disappears earlier than in $\lambda\mathrm{C}^\forall_{mp}$: for example, $42\langle\alpha^-\rangle$, which is a $\lambda\mathrm{C}^\forall_{mp}$ value, is translated to $42\langle\mathsf{id}\rangle$, which further reduces to 42. Thus, the last rule allows the translation of no-op coercions not to appear in $\lambda\mathrm{S}^\forall_{mp}$. Here, the coercions in the last three rules have to be typed under the empty type environment, because the coercion composition and early removal of no-op coercions take place only at the top-level.

Below are the key properties of the relation $\approx$, from which the theorem above easily follows.

Lemma 4.5 (Relating Terms in $\lambda\mathrm{C}^\forall_{mp}$ to Their Translations). *If* $\Sigma \mid \Gamma \vdash_C M_c : A$*, then* $\Sigma \mid \Gamma \vdash M_c \approx |M_c|_\Gamma : A$*.*

Lemma 4.6 (Bisimulation up to Reduction). *Suppose that* $\Sigma \mid \emptyset \vdash M_c \approx M_s : A$*.*

(1) *If* $\Sigma \triangleright M_c \longrightarrow_C \Sigma' \triangleright M_c'$*, then* $\Sigma' \triangleright M_c' \longrightarrow_C^* \Sigma'' \triangleright M_c''$ *and* $\Sigma \triangleright M_s \longrightarrow_S^* \Sigma'' \triangleright M_s''$ *and* $\Sigma'' \mid \emptyset \vdash M_c'' \approx M_s'' : A$ *for some* Σ''*,* M_c''*, and* M_s''*.*
(2) *If* $\Sigma \triangleright M_s \longrightarrow_S \Sigma' \triangleright M_s'$*, then* $\Sigma' \triangleright M_s' \longrightarrow_S^* \Sigma'' \triangleright M_s''$ *and* $\Sigma \triangleright M_c \longrightarrow_C^* \Sigma'' \triangleright M_c''$ *and* $\Sigma'' \mid \emptyset \vdash M_c'' \approx M_s'' : A$ *for some* Σ''*,* M_c''*, and* M_s''*.*
(3) *If* $M_c = V_c$*, then* $\Sigma \triangleright M_s \longrightarrow_S^* \Sigma \triangleright V_s$ *and* $\Sigma \mid \emptyset \vdash V_c \approx V_s : A$ *for some* V_s*.*
(4) *If* $M_s = V_s$*, then* $\Sigma \triangleright M_c \longrightarrow_C^* \Sigma \triangleright V_c$ *and* $\Sigma \mid \emptyset \vdash V_c \approx V_s : A$ *for some* V_c*.*
(5) *If* $M_c = \mathsf{blame}\, p$*, then* $\Sigma \triangleright M_s \longrightarrow_S^* \Sigma \triangleright \mathsf{blame}\, p$*.*
(6) *If* $M_s = \mathsf{blame}\, p$*, then* $\Sigma \triangleright M_c \longrightarrow_C^* \Sigma \triangleright \mathsf{blame}\, p$*.*

Comparison with Siek et al. [2021]. Siek et al. even proved full abstraction of the translation from a simply typed coercion calculus $\lambda\mathrm{C}$ to a simply typed, space-efficient coercion calculus $\lambda\mathrm{S}$ by providing a relation $M_c \approx M_s$ to relate terms M_c in $\lambda\mathrm{C}$ to their translation results M_s in $\lambda\mathrm{S}$. However, their technical developments for full abstraction and ours differ in two points. First, whereas we proved that the relation $\approx$ is a bisimulation up to reduction [Sangiorgi et al. 2007], Siek et al. showed that their relation $\approx$ is a weak bisimulation, which implies that, if $M_c \approx M_s$ and $M_c \longrightarrow_C M_c'$, then $M_s \longrightarrow_S^* M_s'$ for some M_s' such that $M_c' \approx M_s'$ (Proposition 19 in their paper). We first tried to show that a straightforward extension of Siek et al.'s relation is a weak bisimulation, but in the course of it, we discovered that Siek et al.'s relation is not a weak bisimulation actually; we provide a counterexample in Section J of the supplementary material. Second, Siek et al.'s relation $\approx$ is defined with an extra inference rule, named (iii) in Figure 8 of their paper, that decomposes function coercions, but our relation is defined without such a rule. We can dispense with (iii) because we focus on a bisimulation up to reduction, not a weak bisimulation.

Unfortunately, full abstraction would not hold in our setting. For full abstraction to hold, every space-efficient coercion has to have a counterpart in $\lambda C^{\forall}$—in other words, the translation has to be surjective—but it is not the case. An example is $\forall X.((\forall Y.(\bot^p \fatsemi X!)) \fatsemi (\forall Y.(Y?^p \fatsemi \bot^p)))$. We may be able to exclude such a coercion from $\lambda S^{\forall}_{mp}$ by requiring every space-efficient coercion is in the image of the translation but we have not found good (syntactic) characterization of the image.

5 RELATED WORK

Polymorphic Gradual Typing. Polymorphic gradual typing has been studied first by Ahmed et al. [2011]. They force polymorphic values to behave independently of type parameters using dynamic sealing [Abadi et al. 1995; Matthews and Ahmed 2008; Morris 1973; Pierce and Sumii 2000], albeit no formal treatment of parametricity. Since their pioneering work, polymorphic gradual typing has been gaining attention [Ahmed et al. 2011, 2017; Igarashi et al. 2017; Miyazaki et al. 2019; New et al. 2020; Ozaki et al. 2021; Toro et al. 2019; Xie et al. 2018], especially to its theoretical aspects such as parametricity and dynamic gradual guarantee [Siek et al. 2015b]. Among them, all the calculi with parametricity employ dynamic sealing, that is, the run-time generation of fresh type names. Therefore, although the ingredients to incorporate dynamic sealing are varying, the parametric calculi in the prior works on polymorphic gradual typing do not seem implementable space-efficiently due to the problem of accumulating an unbounded number of type conversions involving dynamically generated type names. The polymorphic cast calculus of Miyazaki et al. [2019] does not support dynamic sealing, so it might be made space-efficient but pays the cost of the complete loss of parametricity. Our work shows that it is overpaying: a polymorphic gradually typed language can be made space-efficient if it is "mostly" parametric.

Space-Efficient Gradual Typing. Herman et al. [2007, 2010] were the first to address the space-efficiency problem in gradual typing and gave a space-efficient coercion calculus that eagerly combines and normalizes two adjacent coercions into their composition. However, the computation of the normalization is ambiguous since the normalization is defined as an equational system. Siek et al. [2015a, 2021] addressed this issue by proposing a coercion calculus λS that restricts coercions to be a kind of canonical forms called *space-efficient coercions* and gives the composition on space-efficient coercions in a computational manner. Our calculus $\lambda S^{\forall}_{mp}$ extends their idea to polymorphic gradual typing. Siek and Wadler [2010] introduced a space-efficient cast calculus with another cast form called *threesomes*, but it is proven to be isomorphic to λS [Siek et al. 2021]. Bañados Schwerter et al. [2021] addressed space efficiency in GTFL$_{\lesssim}$ [Garcia et al. 2016], a gradually typed language with records and subtyping. They showed that the run-time language of GTFL$_{\lesssim}$ can be made space-efficient if the composition of run-time checks, called *evidence*, in the language associative and bounded, and provided new representations of evidence that meet these conditions. As far as we know, all the previous works for space-efficient gradual typing have focused only on a simply typed setting (with subtyping) except for Ozaki et al. [2021], who discovered and formalized the problem of dynamic sealing and space-efficiency, but did not consider alternatives to make polymorphic gradually typed languages space-efficient.

Although the work discussed above is concerned about theoretical aspects of space-efficient gradual typing, there is some work on its real implementation. As we mentioned in the introduction, Kuhlenschmidt et al. [2019] implemented the Grift compiler for a gradually typed λ-calculus (with mutable arrays and recursive types), 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, proved its correctness formally, and extended Grift.

Coercions on Polymorphic Types. Breazu-Tannen et al. [1991]; Mitchell [1984] study coercions to give the semantics of subtyping; Luo [1999]; Swamy et al. [2009] study them for more general type conversion. Along this line of work, Cretin and Rémy [2012] propose a sophisticated language of coercions and 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. The issue of space efficiency is not studied.

6 CONCLUSION

We have shown that polymorphic gradual typing can be made space-efficient by mostly parametric polymorphism—limiting parametricity to type arguments that are not the dynamic type—at least, theoretically. We have formalized a polymorphic coercion calculus $\lambda \mathrm{C}^{\forall}_{mp}$ with dynamic sealing to enforce parametricity, another calculus $\lambda \mathrm{S}^{\forall}_{mp}$ with space-efficient coercions, and a type-preserving translation from the former to the latter and shown that the latter is space-efficient (in the sense of Herman et al. [2007, 2010]) and the translation is semantics-preserving.

As we noted in the introduction, our work presented here focuses on the theory of space-efficient coercions. Although we suppose that our design is a simple but reasonable compromise for the design of programming languages with polymorphic gradual typing—because passing $\star$ as a type argument can be seen as a sign that that part of code is not yet ready to enjoy the benefits of static typing, including parametricity—it is left for future work to support such a claim by empirical study. If our restriction turns out to be too severe, it will be interesting work to develop a sophisticated static or dynamic analysis to detect the use of $\star$ that does not cause coercions of an unbounded size.

It is also left for future work to implement $\lambda \mathrm{S}^{\forall}_{mp}$. We expect that the coercion-passing-style implementation [Tsuda et al. 2020] can be adapted to the polymorphic setting straightforwardly.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers of the PLDI 2024 program committee for their fruitful comments. This work was supported in part by JSPS KAKENHI Grant Numbers JP20H00582 (Igarashi) and JP19K20247 and JP22K17875 (Sekiyama).

REFERENCES

Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Didier Rémy. 1995. Dynamic Typing in Polymorphic Languages. *J. Funct. Program.* 5, 1 (1995), 111–130. https://doi.org/10.1017/S095679680000126X

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

Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy G. Siek, and Sam Tobin-Hochstadt. 2017. Sound gradual typing: only mostly dead. *PACMPL* 1, OOPSLA (2017), 54:1–54:24. https://doi.org/10.1145/3133878

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

Xuejing Huang and Bruno C. d. S. Oliveira. 2020. A Type-Directed Operational Semantics For a Calculus with a Merge Operator. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 26:1–26:32. https://doi.org/10.4230/LIPICS.ECOOP.2020.26

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

Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. 2019. Dynamic Type Inference for Gradual Hindley–Milner Typing. *PACM on Programming Languages* 3, POPL (Jan. 2019), 18:1–18:29. https://doi.org/10.1145/3290331 Presented at ACM POPL 2019.

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

Fabian Muehlboeck and Ross Tate. 2017. Sound gradual typing is nominally alive and well. *PACMPL* 1, OOPSLA (2017), 56:1–56:30. https://doi.org/10.1145/3133880

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

Shota Ozaki, Taro Sekiyama, and Atsushi Igarashi. 2021. Is Space-Efficient Polymorphic Gradual Typing Possible?. In *Scheme and Functional Programming Workshop*.

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

Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 167–180. https://doi.org/10.1145/2676726.2676971

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. 1988. *Preliminary Design of the Programming Language Forsythe*. Technical Report CS-CMU-88-159. Carnegie Mellon University, Pittsburgh, PA. https://doi.org/10.1184/R1/6608582.v1

Gregor Richards, Ellen Arteca, and Alexi Turcotte. 2017. The VM already knew that: leveraging compile-time knowledge to optimize gradual typing. *PACMPL* 1, OOPSLA (2017), 55:1–55:27. https://doi.org/10.1145/3133879

Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. 2007. Environmental Bisimulations for Higher-Order Languages. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*. IEEE Computer Society, 293–302. https://doi.org/10.1109/LICS.2007.17

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

Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4609)*, Erik Ernst (Ed.). Springer, 2–27. https://doi.org/10.1007/978-3-540-73589-2_2

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

Jeremy G. Siek, Peter Thiemann, and Philip Wadler. 2021. Blame and coercion: Together again for the first time. *J. Funct. Program.* 31 (2021), e20. https://doi.org/10.1017/S0956796821000101 A preliminary version is presented at ACM PLDI'15.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015b. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA (LIPIcs, Vol. 32)*, Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

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

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage migration: from scripts to programs. In *Proc. of Dynamic Languages Symposium*. 964–974. https://doi.org/10.1145/1176617.1176755

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

Received 2023-11-16; accepted 2024-03-31