


Space-Efficient Gradual Typing in Coercion-Passing Style

Yuya Tsuda 

Graduate School of Informatics, Kyoto University, Japan
tsuda@fos.kuis.kyoto-u.ac.jp

Atsushi Igarashi 

Graduate School of Informatics, Kyoto University, Japan
igarashi@kuis.kyoto-u.ac.jp

Tomoya Tabuchi

Graduate School of Informatics, Kyoto University, Japan
tabuchi@fos.kuis.kyoto-u.ac.jp

Abstract

Herman et al. pointed out that the insertion of run-time checks into a gradually typed program could hamper tail-call optimization and, as a result, worsen the space complexity of the program. To address the problem, they proposed a space-efficient coercion calculus, which was subsequently improved by Siek et al. The semantics of these calculi involves eager composition of run-time checks expressed by coercions to prevent the size of a term from growing. However, it relies also on a nonstandard reduction rule, which does not seem easy to implement. In fact, no compiler implementation of gradually typed languages fully supports the space-efficient semantics faithfully.

In this paper, we study *coercion-passing style*, which Herman et al. have already mentioned, as a technique for straightforward space-efficient implementation of gradually typed languages. A program in coercion-passing style passes “the rest of the run-time checks” around – just like continuation-passing style (CPS), in which “the rest of the computation” is passed around – and (unlike CPS) composes coercions eagerly. We give a formal coercion-passing translation from λS by Siek et al. to λS_1 , which is a new calculus of *first-class coercions* tailored for coercion-passing style, and prove correctness of the translation. We also implement our coercion-passing style transformation for the Grift compiler developed by Kuhlenschmidt et al. An experimental result shows stack overflow can be prevented properly at the cost of up to 3 times slower execution for most partially typed practical programs.

2012 ACM Subject Classification Theory of computation → Semantics and reasoning; Software and its engineering → Compilers; Theory of computation → Operational semantics

Keywords and phrases Gradual typing, coercion calculus, coercion-passing style, dynamic type checking, tail-call optimization

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.8

Related Version A full version of the paper is available at <https://arxiv.org/abs/1908.02414>.

Funding This work was partially supported by JSPS KAKENHI Grant Number JP17H01723.

Acknowledgements We thank anonymous reviewers for valuable comments and John Toman for proofreading.

1 Introduction

1.1 Space-Efficiency Problem in Gradual Typing

Gradual typing [36, 40] is one of the linguistic approaches to integrating static and dynamic typing. Allowing programmers to mix statically typed and dynamically typed fragments in a single program, it advocates the “script to program” evolution [40]. Namely, software



© Yuya Tsuda, Atsushi Igarashi, and Tomoya Tabuchi;
licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 8; pp. 8:1–8:29

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

8:2 Space-Efficient Gradual Typing in Coercion-Passing Style

development starts with simple, often dynamically typed scripts, which evolve to more robust, fully statically typed programs through intermediate stages of partially typed programs. To make this evolution work in practice, it is important that the performance of partially typed programs at intermediate stages is comparable to that of (the slower of) the two ends, that is, dynamically typed scripts and statically typed programs.

However, it has been pointed out that gradual typing suffers from serious efficiency problems from both theoretical and practical viewpoints [19, 20, 39]. In particular, Takikawa et al. [39] showed that even a state-of-the-art gradual typing implementation could show catastrophic slowdown for partially typed programs due to run-time checking to ensure safety. Worse, such slowdown is not easy to predict because it depends on implicit run-time checks inserted by the language implementation and it requires fairly deep knowledge about the underlying gradual type system to understand when and where run-time checks are inserted and how they behave. Since then, several pieces of work have investigated the performance issues [4, 27, 31, 29, 24, 12].

Earlier work by Herman et al. [19, 20] pointed out a related problem. They showed that, when values are passed between a statically typed part and a dynamically typed part many times, delayed run-time checks may accumulate and make space complexity of a program worse than an unchecked semantics.

To make the discussion more concrete, consider the following mutually recursive functions (written in ML-like syntax):

```
let rec even (x : int) : * =
  if x = 0 then true⟨bool!⟩ else (odd (x - 1))⟨bool!⟩
and odd (x : int) : bool =
  if x = 0 then false else (even (x - 1))⟨bool?p⟩
```

Ignoring the gray part (in angle brackets), which will be explained shortly, this is a tail-recursive definition of functions to decide whether a given integer is even or odd, except that the return type of one of the functions is written \star , which is the dynamic type, which can be any tagged value. This definition expresses a situation where a statically typed and a dynamically typed function call each other.¹ The gray part represents inserted run-time checks, written using Henglein’s coercion syntax [18]: bool! is a coercion from bool to \star and $\text{true}\langle\text{bool!}\rangle$ means that (untagged) Boolean value true will be tagged with bool to make a value of the dynamic type; bool?^p is a coercion from \star to bool and $(\text{even } (x - 1))\langle\text{bool?}^p\rangle$ means that the value returned from recursive call $\text{even } (x - 1)$ will be tested whether it is tagged with bool – if so, the run-time check removes the tag and returns the untagged Boolean value, and, otherwise, it results in *blame*, which is an uncatchable exception (with label p to indicate where the check has failed).

The crux of this example is that the insertion of run-time checks has broken tail recursion: due to $\langle\text{bool!}\rangle$ and $\langle\text{bool?}^p\rangle$, the recursive calls are not in tail positions any longer. So, according to the original semantics of coercions [18], evaluation of $\text{odd } 4$ is as follows:

$$\begin{aligned} \text{odd } 4 &\mapsto^* (\text{even } 3)\langle\text{bool?}^p\rangle \mapsto^* (\text{odd } 2)\langle\text{bool!}\rangle\langle\text{bool?}^p\rangle \\ &\mapsto^* (\text{even } 1)\langle\text{bool?}^p\rangle\langle\text{bool!}\rangle\langle\text{bool?}^p\rangle \mapsto^* (\text{odd } 0)\langle\text{bool!}\rangle\langle\text{bool?}^p\rangle\langle\text{bool!}\rangle\langle\text{bool?}^p\rangle \\ &\mapsto^* \text{false}\langle\text{bool!}\rangle\langle\text{bool?}^p\rangle\langle\text{bool!}\rangle\langle\text{bool?}^p\rangle \mapsto^* \text{false} \end{aligned}$$

¹ In this sense, the argument of even should have been \star , too, but it would clutter the code after inserting run-time checks.

$ \begin{aligned} &\text{odd } 4 \\ &\mapsto^* (\text{even } 3)\langle \text{bool}^{?^p} \rangle \\ &\mapsto (\text{odd } (3 - 1))\langle \text{bool}! \rangle\langle \text{bool}^{?^p} \rangle \\ &\mapsto (\text{odd } (3 - 1))\langle \text{bool}! \ ; \ \text{bool}^{?^p} \rangle \\ &= (\text{odd } (3 - 1))\langle \text{id}_{\text{bool}} \rangle \\ &\mapsto (\text{odd } 2)\langle \text{id}_{\text{bool}} \rangle \\ &\mapsto (\text{even } (2 - 1))\langle \text{bool}^{?^p} \rangle\langle \text{id}_{\text{bool}} \rangle \\ &\mapsto (\text{even } (2 - 1))\langle \text{bool}^{?^p} \ ; \ \text{id}_{\text{bool}} \rangle \\ &= (\text{even } (2 - 1))\langle \text{bool}^{?^p} \rangle \\ &\mapsto (\text{even } 1)\langle \text{bool}^{?^p} \rangle \\ &\mapsto \dots \end{aligned} $	$ \begin{aligned} &\text{oddk } (4, \text{id}_{\text{bool}}) \\ &\mapsto \text{evenk } (4 - 1, \text{bool}^{?^p} \ ; \ ; \ \text{id}_{\text{bool}}) \\ &\mapsto \text{evenk } (4 - 1, \text{bool}^{?^p}) \\ &\mapsto \text{evenk } (3, \text{bool}^{?^p}) \\ &\mapsto \text{oddk } (3 - 1, \text{bool}! \ ; \ ; \ \text{bool}^{?^p}) \\ &\mapsto \text{oddk } (3 - 1, \text{id}_{\text{bool}}) \\ &\mapsto \text{oddk } (2, \text{id}_{\text{bool}}) \\ &\mapsto \text{evenk } (2 - 1, \text{bool}^{?^p} \ ; \ ; \ \text{id}_{\text{bool}}) \\ &\mapsto \text{evenk } (2 - 1, \text{bool}^{?^p}) \\ &\mapsto \text{evenk } (1, \text{bool}^{?^p}) \\ &\mapsto \dots \end{aligned} $
--	---

■ **Figure 1** Reduction from $\text{odd } 4$ in $\lambda\mathcal{S}$ (left) and reduction from $\text{oddk } (4, \text{id}_{\text{bool}})$ in $\lambda\mathcal{S}_1$ (right).

Thus, the size of a term being evaluated is proportional to the argument n at its longest, whereas unchecked semantics (without coercions) allows for tail-call optimization and constant-space execution. This is the space-efficiency problem of gradual typing.

1.2 Space-Efficient Gradual Typing

Herman et al. [19, 20] also presented a solution to this problem. In the evaluation sequence of $\text{odd } n$ above, we could immediately “compress” nested coercion applications $M\langle \text{bool}! \rangle\langle \text{bool}^{?^p} \rangle$ before computation of the target term M ends, because $\langle \text{bool}! \rangle\langle \text{bool}^{?^p} \rangle$ – tagging immediately followed by untagging – is equivalent to the identity function. By doing so, we can maintain that the order of the size of a term in the middle of evaluation is constant. This idea is formalized in terms of a “space-efficient” extension of the coercion calculus [18]. Since then, a few space-efficient coercion/cast calculi have been proposed [37, 38, 35].

Among them, Siek et al. [37] have proposed a space-efficient coercion calculus $\lambda\mathcal{S}$. $\lambda\mathcal{S}$ is equipped with a composition function that compresses consecutive coercions in certain canonical forms. The coercion composition is achieved as a simple recursive function thanks to the canonical forms. We show evaluation of $\text{odd } 4$ according to the $\lambda\mathcal{S}$ semantics in the left of Figure 1.² Here, $s \ ; \ t$ is a meta-level operation that composes two coercions s, t (in canonical forms) and yields another canonical coercion that semantically corresponds to their sequential composition. This composition function enables us to prevent the size of a term from growing.

However, in order to ensure that nested coercion applications are always merged, the operational semantics of $\lambda\mathcal{S}$ relies on a nonstandard reduction rule and nonstandard evaluation contexts. Although it does not cause any theoretical problems, it does not seem easy to implement – in particular, its compilation method seems nontrivial. In fact, none of the existing compiler implementations that address the space-efficiency problem [24, 12] solves the problem of growing coercions at tail positions (an exception is recent work by Castagna et al. [5] – See Section 6 for more comparison).

² Strictly speaking, $\text{bool}!$ and $\text{bool}^{?^p}$ are abbreviations of $\text{id}_{\text{bool}}; \text{bool}!$ and $\text{bool}^{?^p}; \text{id}_{\text{bool}}$, respectively, in $\lambda\mathcal{S}$.

1.3 Our Work: Coercion-Passing Style

In this paper, we study coercion-passing style for space-efficient gradual typing. Just as continuation-passing style, in which “the rest of the computation” is passed around as first-class functions and every function call is at a tail position, a program in coercion-passing style passes “the rest of the run-time checks” around. Actually, the idea of coercion-passing style has already been listed as one of the possible implementation techniques by Herman et al. [19, 20] but it has been neither well studied nor formalized.

We use the even/odd example above to describe our approach to the problem. Here are the even/odd functions in coercion-passing style. (We omit type declarations for simplicity.)

```
let rec evenk (x, κ) =
  if x = 0 then true⟨bool! ;; κ⟩ else oddk (x - 1, bool! ;; κ)
and oddk (x, κ) =
  if x = 0 then false⟨κ⟩ else evenk (x - 1, bool?p ;; κ)
```

Additional parameters named κ are for *first-class coercions*, which are supposed to be applied – as in $\text{false}\langle\kappa\rangle$ – to values that are returned in the original function definition. We often call these coercions *continuation coercions*. Coercion applications such as $\text{true}\langle\text{bool!}\rangle$ and $(\text{oddk } (x - 1))\langle\text{bool!}\rangle$ at tail positions in the original program are translated to coercion compositions such as $\text{true}\langle\text{bool!} ;; \kappa\rangle$ and $\text{oddk } (x - 1, \text{bool!} ;; \kappa)$, respectively. When κ is bound to a concrete coercion, it will be composed with bool! *before it is applied*. Similarly to programs in CPS, function calls pass (composed) coercions.

With these functions in coercion-passing style, the evaluation of $\text{oddk } (4, \text{id}_{\text{bool}})$ (where id_{bool} is an identity coercion, which does nothing) proceeds as in the right of Figure 1. Since tagging followed by untagging (with the same tag) actually does nothing, $\text{bool!} ;; \text{bool?}^p$ composes to id_{bool} by the (meta-level) coercion composition $\text{bool!} \ ; \ \text{bool?}^p$.

Similarly to the $\lambda\mathcal{S}$ semantics described above, coercion composition in the argument takes place before a recursive call, thus the size of coercions stays bounded by the constant order, overcoming the space efficiency problem. A nice property of our solution is that the evaluation is standard call-by-value.

One can view the extra parameter κ as an accumulating parameter and continuation coercions as (delimited) continuations in defunctionalized forms [30]. Unlike simple defunctionalization, however, special composition of two defunctionalized coercions is provided, preventing the sizes of composed coercions from growing.

Contributions

Since the operational semantics of $\lambda\mathcal{S}$ seems nontrivial to implement due to a nonstandard reduction rule, we investigate implementation of the space-efficient semantics via a translation into coercion-passing style. Our contributions in this paper are summarized as follows:

- In the context of the space-efficiency problem of gradual typing, we develop a new calculus $\lambda\mathcal{S}_1$ of space-efficient first-class coercions.
- We formalize a coercion-passing style translation from (a slight variant of) space-efficient coercion calculus $\lambda\mathcal{S}$ [37] to the new calculus $\lambda\mathcal{S}_1$.
- We prove correctness of the coercion-passing style translation via a simulation property.
- We implement the coercion-passing style translation on top of the Grift compiler [24], and conduct some experiments to show that stack overflow is indeed avoided.

Outline

The rest of this paper is organized as follows. We review the space-efficient coercion calculus $\lambda\mathcal{S}$ [37] in Section 2. We introduce a new space-efficient coercion calculus with first-class coercions $\lambda\mathcal{S}_1$ in Section 3, formalize a translation into coercion-passing style as a translation from $\lambda\mathcal{S}$ to $\lambda\mathcal{S}_1$, and prove correctness of the translation in Section 4. We discuss our implementation of coercion-passing translation on top of the Grift compiler [24] and show an experimental result in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7. Proofs of the stated properties can be found in the full version.

2 Space-Efficient Coercion Calculus

In this section, we review the space-efficient coercion calculus $\lambda\mathcal{S}$ [37], which is the source calculus of our translation. Our definition differs from the original in a few respects, as we will explain later. For simplicity, we do not include (mutually) recursive functions and conditional expressions in the formalization but it is straightforward to add them; in fact, our implementation includes them.

Main novelties of $\lambda\mathcal{S}$ over the original coercion calculus $\lambda\mathcal{C}$ [18] are (1) space-efficient coercions, which are canonical forms of coercions, whose composition can be defined by a straightforward recursive function, and (2) operational semantics in which a sequence of coercion applications is collapsed eagerly – even before they are applied to a value [19, 20, 35].

Basic forms of coercions are inherited from $\lambda\mathcal{C}$ [18], which provides (1) identity coercions id_A (where A is a type), which do nothing; (2) injections $G!$, which add a type tag G to a value to make a value of the dynamic type; (3) projections $G?^p$, which test whether a value of the dynamic type is tagged with G , remove the tag if the test succeeds, or raise blame labeled p if it fails; (4) function coercions $c_1 \rightarrow c_2$, which, when they are applied to a function, coerce an argument to the function by c_1 and a value returned from the function by c_2 ; and (5) sequential compositions $c_1; c_2$, which apply c_1 and c_2 in this order. Space-efficient coercions restrict the way basic coercions are combined by sequential composition; they can be roughly expressed by the following regular expression:

$$(G?^p;)^?(\text{id}_\iota + (s_1 \rightarrow s_2)); G!^?)^?$$

(where ι is a base type, s_1 and s_2 stand for space efficient coercions, $(\dots)^?$ stands for an optional element, and $+$ for alternatives). As already mentioned, an advantage of this form is that (meta-level) sequential composition (denoted by $s_1 \mathbin{\text{;}} s_2$) of two space-efficient coercions results in another space-efficient coercion (if the composition is well typed), in other words, space-efficient coercions are closed under $s_1 \mathbin{\text{;}} s_2$. For example, the composition $((G_1?^p;)^?(\text{id}_\iota + (s_1 \rightarrow s_2)); G_2!) \mathbin{\text{;}} (G_3?^{p'}; (\text{id}_\iota + (s_3 \rightarrow s_4)); G_4!)^?$ will be $((G_1?^p;)^?(\text{id}_\iota + ((s_3 \mathbin{\text{;}} s_1) \rightarrow (s_2 \mathbin{\text{;}} s_4))); G_4!)^?$ if $G_2 = G_3$ – that is, tagging with G_2 is immediately followed by inspection whether G_2 is present.³ Notice that the resulting coercion conforms to the regular expression again. (The other case where $G_2 \neq G_3$ means that the projection $G_3?^{p'}$ will fail; we will explain such failures later.)

The operational semantics includes the reduction rule $\mathcal{F}[M\langle s \rangle\langle t \rangle] \longrightarrow \mathcal{F}[M\langle s \mathbin{\text{;}} t \rangle]$ where \mathcal{F} is an evaluation context that does not include nested coercion applications and whose innermost frame is not a coercion application. This rule intuitively means that two consecutive coercions at the outermost position will be composed *even before M is evaluated to a value*. This eager composition avoids a long chain of coercion applications in an evaluation context.

³ Here, we exclude ill-typed coercion compositions such as $(s_1 \rightarrow s_2) \mathbin{\text{;}} \text{id}_\iota$.

Variables	x, y	Constants	a, b	Operators	op	Blame labels	p
Base types			$\iota ::= \text{int} \mid \text{bool} \mid \dots$				
Types			$A, B, C ::= \star \mid \iota \mid A \rightarrow B$				
Ground types			$G, H ::= \iota \mid \star \rightarrow \star$				
Space-efficient coercions			$s, t ::= \text{id}_\star \mid G^{?P}; i \mid i$				
Intermediate coercions			$i ::= g; G! \mid g \mid \perp^{GpH}$				
Ground coercions			$g, h ::= \text{id}_A \text{ (if } A \neq \star) \mid s \rightarrow t \text{ (if } s \neq \text{id or } t \neq \text{id)}$				
Delayed coercions			$d ::= g; G! \mid s \rightarrow t \text{ (if } s \neq \text{id or } t \neq \text{id)}$				
Terms			$L, M, N ::= V \mid op(M, N) \mid MN \mid M\langle s \rangle \mid \text{blame } p$				
Values			$V, W ::= x \mid U \mid U\langle\langle d \rangle\rangle$				
Uncoerced values			$U ::= a \mid \lambda x. M$				
Type environments			$\Gamma ::= \emptyset \mid \Gamma, x : A$				

■ **Figure 2** Syntax of $\lambda\mathcal{S}$.

2.1 Syntax

We show the syntax of $\lambda\mathcal{S}$ in Figure 2. The syntax of $\lambda\mathcal{S}$ extends that of the simply typed lambda calculus (written in gray) with the dynamic type and (space-efficient) coercions.

Types, ranged over by A, B, C , include the dynamic type \star , base types ι , and function types $A \rightarrow B$. Base types ι include `int` (integer type) and `bool` (Boolean type) and so on. *Ground types*, ranged over by G, H , include base types ι and the function type $\star \rightarrow \star$. They are used for type tags put on values of the dynamic type [43]. Here, the ground type for functions is always $\star \rightarrow \star$, reflecting the fact that many dynamically typed languages do not include information on the argument and return types of the function in its type tag.

As we have already discussed, $\lambda\mathcal{S}$ restricts coercions to only canonical ones, namely space-efficient coercions s , whose grammar is defined via ground coercions g and intermediate coercions i . Ground coercions correspond to the middle part of space-efficient coercions; unlike the original $\lambda\mathcal{S}$, ground coercions include identity coercions for any function types – such as $\text{id}_{\iota \rightarrow \iota}$ – and exclude “virtually identity” coercions such as $\text{id}_{\iota} \rightarrow \text{id}_{\iota}$. Although these two coercions are extensionally the same, they reduce in slightly different ways: applying $\text{id}_{\iota \rightarrow \iota}$ to a function immediately returns the function, whereas applying $\text{id}_{\iota} \rightarrow \text{id}_{\iota}$ results in a wrapped function whose argument and return values are monitored by id_{ι} , which does nothing. Adopting id_A for any A simplifies our proof that the coercion-passing translation preserves the semantics. An intermediate coercion adds an optional injection to a ground coercion. Coercions of the form \perp^{GpH} trigger blame (labeled p) if applied to a value. They emerge from coercion composition

$$((G_1^{?P};)^?(\text{id}_A + (s_1 \rightarrow s_2)); G_2!) \circ (G_3^{?P'}; (\text{id}_A + (s_3 \rightarrow s_4)))(; G_4!)^?$$

where $A \neq \star$ and $G_2 \neq G_3$, which means that the projection $G_3^{?P'}$ is bound to fail. The composition results in $(G_1^{?P};)^? \perp^{G_1 P' G_3}$, which means that, unless the optional projection fails – blaming p – it fails with p' . Finally, space-efficient coercions are obtained by adding optional projection to intermediate coercions. id_\star is a special coercion that does not conform to the regular expression above. Strictly speaking, an injection, say `int!`, has to be written $\text{id}_{\text{int}}; \text{int!}$ and a projection, say `int?P`, has to be written $\text{int}^{?P}; \text{id}_{\text{int}}$. We often omit these identity coercions in examples.

Terms, ranged over by L, M, N , include values V , primitive binary operations $op(M, N)$, function applications $M N$, coercion applications $M\langle s \rangle$, and coercion failure $\mathbf{blame} p$. The term $M\langle s \rangle$ coerces the value of M with coercion s at run time. The term $\mathbf{blame} p$ denotes a run-time type error caused by the failure of a coercion (projection) with blame label p .

Values, ranged over by V, W , include variables x , uncoerced values U , and coerced values $U\langle\langle d \rangle\rangle$. Uncoerced values, ranged over by U , include constants a of base types and lambda abstractions $\lambda x. M$. Unlike λC , where values can involve nested coercion applications, there is at most one coercion in a value – nested coercions will be composed. Coerced values $U\langle\langle d \rangle\rangle$ have two forms: injected values $U\langle\langle g; G! \rangle\rangle$ and wrapped functions $U\langle\langle s \rightarrow t \rangle\rangle$. The check of function coercion is delayed until wrapped functions are applied to a value [18, 13, 36]. We include variables as values for technical convenience in defining translations; for operational semantics, though, it is not necessary to do so because we consider evaluation of closed terms.

Unlike many other studies on coercion and blame calculi, we syntactically distinguish coerced values $U\langle\langle d \rangle\rangle$ from $U\langle d \rangle$ (similarly to Wadler and Findler [43]). This distinction plays an important role in our correctness proof; roughly speaking, without the distinction, $U\langle d \rangle\langle t \rangle$ would allow two different interpretations: an application of t to a value $U\langle d \rangle$ or two applications of d and t to a value U , which would result in different translation results. We also note that variables x are considered values, rather than uncoerced values, since they can be bound to coerced values at function calls. In other words, we ensure that values are closed under value substitution.

As usual, applications are left-associative and λ extends as far to the right as possible. We do not commit to a particular choice of precedence between function applications and coercion applications; we will always use parentheses to disambiguate terms like $M N\langle t \rangle$. The term $\lambda x. M$ binds x in M as usual. The definitions of free variables and α -equivalence of terms are standard, and thus we omit them. We identify α -equivalent terms.

The metavariable Γ ranges over *type environments*. A type environment is a sequence of pairs of a variable and its type.

2.2 Type System

We give the type system of λS , which consists of three judgments for *type consistency* $A \sim B$, *well-formed coercions* $c : A \rightsquigarrow B$, and *typing* $\Gamma \vdash_S M : A$. We use c to denote any kind of coercions. The inference rules (except for $A \sim B$) are shown in Figure 3. (We omit the subscript S on \vdash in rules, as some of them are reused for λS_1 .)

The type consistency relation $A \sim B$ is the least reflexive and symmetric and compatible relation that contains $A \sim \star$. As this is standard [36], we omit inference rules here. (We have them in the full version.)

The relation $c : A \rightsquigarrow B$ means that coercion c , which ranges over all kinds of coercions, converts a value from type A to type B . We often call A and B the source and target types of c , respectively. The rule (CT-ID) is for identity coercion id_A . The rule (CT-INJ) is for injection $G!$, which converts type G to type \star . The rule (CT-PROJ) is for projection $G?^p$, which converts type \star to type G . The rule (CT-FUN) is for function coercion $c_1 \rightarrow c_2$. If its argument coercion c_1 converts type A' to type A and its return-value coercion c_2 converts type B to type B' , then function coercion $c_1 \rightarrow c_2$ converts type $A \rightarrow B$ to type $A' \rightarrow B'$. In other words, function coercions are contravariant in their argument coercions and covariant in return-value coercions. The rule (CT-FAIL) is for failure coercion \perp^{GpH} . Here, the source

$$\begin{array}{c}
 \text{Well-formed coercions} \quad \boxed{c : A \rightsquigarrow B} \\
 \\
 \frac{}{G! : G \rightsquigarrow \star} \text{CT-INJ} \quad \frac{}{G?^p : \star \rightsquigarrow G} \text{CT-PROJ} \quad \frac{c_1 : A' \rightsquigarrow A \quad c_2 : B \rightsquigarrow B'}{c_1 \rightarrow c_2 : A \rightarrow B \rightsquigarrow A' \rightarrow B'} \text{CT-FUN} \\
 \\
 \frac{}{\text{id}_A : A \rightsquigarrow A} \text{CT-ID} \quad \frac{c_1 : A \rightsquigarrow B \quad c_2 : B \rightsquigarrow C}{(c_1; c_2) : A \rightsquigarrow C} \text{CT-SEQ} \quad \frac{A \neq \star \quad A \sim G \quad G \neq H}{\perp^{GpH} : A \rightsquigarrow B} \text{CT-FAIL} \\
 \\
 \text{Term typing} \quad \boxed{\Gamma \vdash_S M : A} \\
 \\
 \frac{}{\Gamma \vdash a : \text{ty}(a)} \text{T-CONST} \quad \frac{\text{ty}(op) = \iota_1 \rightarrow \iota_2 \rightarrow \iota \quad \Gamma \vdash M : \iota_1 \quad \Gamma \vdash N : \iota_2}{\Gamma \vdash op(M, N) : \iota} \text{T-OP} \\
 \\
 \frac{(\lambda x. A) \in \Gamma}{\Gamma \vdash x : A} \text{T-VAR} \quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x. M : A \rightarrow B} \text{T-ABS} \quad \frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B} \text{T-APP} \\
 \\
 \frac{\Gamma \vdash M : A \quad s : A \rightsquigarrow B}{\Gamma \vdash M\langle s \rangle : B} \text{T-CRC} \quad \frac{\emptyset \vdash U : A \quad d : A \rightsquigarrow B}{\emptyset \vdash U\langle\langle d \rangle\rangle : B} \text{T-CRCV} \quad \frac{}{\emptyset \vdash \text{blame } p : A} \text{T-BLAME}
 \end{array}$$

■ **Figure 3** Typing rules of $\lambda\mathcal{S}$.

type is not necessarily G but can be any nondynamic type A consistent with G because the source type of a failure coercion may change during coercion composition. For example, the following judgments are derivable:

$$\begin{array}{l}
 (\text{id}_{\text{int}}; \text{int}!) \rightarrow (\text{int}?^p; \text{id}_{\text{int}}) : \star \rightarrow \star \quad \rightsquigarrow \text{int} \rightarrow \text{int} \\
 \perp^{\star \rightarrow \star p \text{int}} : \text{int} \rightarrow \text{bool} \quad \rightsquigarrow \text{int}
 \end{array}$$

Proposition 1 below, which is about the source and target types of intermediate coercions and ground coercions, is useful to understand the syntactic structure of space-efficient coercions. In particular, it states that neither the source nor target type of ground coercions g is the type \star .

► **Proposition 1** (Source and Target Types).

1. If $i : A \rightsquigarrow B$ then $A \neq \star$.
2. If $g : A \rightsquigarrow B$, then $A \neq \star$ and $B \neq \star$ and $A \sim G$ and $G \sim B$ for some unique G .

The judgment $\Gamma \vdash_S M : A$ means that the $\lambda\mathcal{S}$ -term M is given type A under type environment Γ . When clear from the context, we sometimes write \vdash for \vdash_S with the subscript S omitted. We adopt similar conventions for other relations (such as \mapsto_S) introduced later.

The rules (T-CONST), (T-OP), (T-VAR), (T-ABS), and (T-APP) are standard. Here, $\text{ty}(a)$ maps constant a to a base type ι , and $\text{ty}(op)$ maps binary operator op to a (first-order) function type $\iota_1 \rightarrow \iota_2 \rightarrow \iota$. The rule (T-CRC) states that if M is given type A and space-efficient coercion s converts type A to B , then coercion application $M\langle s \rangle$ is given type B . The rule (T-CRCV) is similar to (T-CRC), but for coerced values $U\langle\langle d \rangle\rangle$. The rule (T-BLAME) allows $\text{blame } p$ to have an arbitrary type A . Here, type environments are always empty \emptyset in (T-CRCV) and (T-BLAME). It is valid because the terms $U\langle\langle d \rangle\rangle$ and $\text{blame } p$ arise only during evaluation, which runs a closed term. In other words, these terms are not written by programmers in the surface language, and also they do not appear as the result of coercion insertion.

Coercion composition

$$\boxed{s \circledast t = s'}$$

$\text{id}_\star \circledast t = t$	CC-IDDYNL	$(G^{?^p}; i) \circledast t = G^{?^p}; (i \circledast t)$	CC-PROJL
$(g; G!) \circledast \text{id}_\star = g; G!$	CC-INJID	$(g; G!) \circledast (G^{?^p}; i) = g \circledast i$	CC-COLLAPSE
$\perp^{G^pH} \circledast s = \perp^{G^pH}$	CC-FAILL	$(g; G!) \circledast (H^{?^p}; i) = \perp^{G^pH}$ (if $G \neq H$)	CC-CONFLICT
$g \circledast \perp^{G^pH} = \perp^{G^pH}$	CC-FAILR	$g \circledast (h; H!) = (g \circledast h); H!$	CC-INJR
$\text{id}_A \circledast g = g$ (if $A \neq \star$)	CC-IDL	$g \circledast \text{id}_A = g$ (if $A \neq \star, g \neq \text{id}_A$)	CC-IDR
$(s \rightarrow t) \circledast (s' \rightarrow t') = \begin{cases} \text{id}_{A \rightarrow B} & \text{if } s' \circledast s = \text{id}_A \text{ and } t \circledast t' = \text{id}_B \\ (s' \circledast s) \rightarrow (t \circledast t') & \text{otherwise} \end{cases}$			CC-FUN

■ **Figure 4** Coercion composition rules of $\lambda\mathcal{S}$.

Evaluation contexts

$$\mathcal{E} ::= \mathcal{F} \mid \mathcal{F}[\square \langle s \rangle] \quad \mathcal{F} ::= \square \mid \mathcal{E}[\text{op}(\square, M)] \mid \mathcal{E}[\text{op}(V, \square)] \mid \mathcal{E}[\square M] \mid \mathcal{E}[V \square]$$

Reduction

$$\boxed{M \xrightarrow{e} N} \quad \boxed{M \xrightarrow{c} N}$$

$\text{op}(a, b) \xrightarrow{e} \delta(\text{op}, a, b)$	R-OP	$U \langle \text{id}_A \rangle \xrightarrow{c} U$	R-ID
$(\lambda x. M) V \xrightarrow{e} M[x := V]$	R-BETA	$U \langle \perp^{G^pH} \rangle \xrightarrow{c} \text{blame } p$	R-FAIL
$(U \langle \langle s \rightarrow t \rangle \rangle) V \xrightarrow{e} (U (V \langle s \rangle)) \langle t \rangle$	R-WRAP	$U \langle d \rangle \xrightarrow{c} U \langle \langle d \rangle \rangle$	R-CRC
		$M \langle s \rangle \langle t \rangle \xrightarrow{c} M \langle s \circledast t \rangle$	R-MERGE C
		$U \langle \langle d \rangle \rangle \langle t \rangle \xrightarrow{c} U \langle d \circledast t \rangle$	R-MERGE V

Evaluation

$$\boxed{M \mapsto_{S_1} N} \quad \boxed{M \xrightarrow{c}_{S_1} N}$$

$$\frac{M \xrightarrow{e} N}{\mathcal{E}[M] \mapsto_{S_1} \mathcal{E}[N]} \text{E-CTXE} \quad \frac{M \xrightarrow{c} N}{\mathcal{F}[M] \mapsto_{S_1} \mathcal{F}[N]} \text{E-CTXC} \quad \frac{\mathcal{E} \neq \square}{\mathcal{E}[\text{blame } p] \mapsto_{S_1} \text{blame } p} \text{E-ABORT}$$

■ **Figure 5** Reduction/evaluation rules of $\lambda\mathcal{S}$.

2.3 Operational Semantics

2.3.1 Coercion Composition

The coercion composition $s \circledast t$ is a recursive function that takes two space-efficient coercions and computes another space-efficient coercion corresponding to their sequential composition. We show the coercion composition rules in Figure 4. The function is defined in such a way that the form of the first coercion determines which rule to apply.

The rules (CC-IDDYNL) and (CC-PROJL) are applied if the first coercion is not an intermediate coercion. The rules (CC-INJID), (CC-COLLAPSE), (CC-CONFLICT), and (CC-FAILL) are applied if the first one is a (nonground) intermediate coercion, in which case another intermediate coercion is yielded. The rules (CC-COLLAPSE) and (CC-CONFLICT) deal with cases where an injection and a projection meet and perform tag checks. If type tags do not match, a failure coercion arises.

Failure coercions are necessary for eager coercion composition to preserve the behavior of $\lambda\mathcal{C}$. The term $M \langle G! \rangle \langle H^{?^p} \rangle$ (if $G \neq H$) in $\lambda\mathcal{C}$ evaluates to $\text{blame } p$ – *only after M evaluates to a value*. By contrast, the two coercions $G!$ and $H^{?^p}$ in the term $M \langle \text{id}_G; G! \rangle \langle H^{?^p}; \text{id}_H \rangle$

8:10 Space-Efficient Gradual Typing in Coercion-Passing Style

are eagerly composed in $\lambda\mathcal{S}$. Raising blame p immediately would not match the semantics of $\lambda\mathcal{C}$ because M may evaluate to another blame or even diverge, in which case p is not blamed. Thus, \perp^{GpH} must raise blame p only after M evaluates to a value.

The rules (CC-FAILR) and (CC-INJR) are applied if a ground coercion and an intermediate coercion are composed to another intermediate coercion. The rules (CC-FAILL) and (CC-FAILR) represent the propagation of a failure to the context, somewhat similarly to exceptions. The rule (CC-INJR) represents associativity of sequential compositions but \circ is propagated to the inside.

The rules (CC-IDL), (CC-IDR), and (CC-FUN) are applied if two ground coercions are composed to another ground coercion. They are straightforward except that $\text{id}_A \rightarrow \text{id}_B$ has to be normalized to $\text{id}_{A \rightarrow B}$ (CC-FUN).

We present a few examples of coercion composition below:

$$\begin{aligned} (\text{id}_{\text{bool}}; \text{bool}!) \circ (\text{bool}^?^p; \text{id}_{\text{bool}}) &= \text{id}_{\text{bool}} \circ \text{id}_{\text{bool}} = \text{id}_{\text{bool}} \\ (\text{id}_{\star \rightarrow \star}; (\star \rightarrow \star)!) \circ (\text{int}^?^p; \text{id}_{\text{int}}) &= \perp^{\star \rightarrow \star p \text{int}} \\ ((\iota^?^p; \text{id}_\iota) \rightarrow (\text{id}_{\iota'}; \iota'!)) \circ ((\text{id}_\iota; \iota!) \rightarrow \text{id}_\star) &= ((\text{id}_\iota; \iota!) \circ (\iota^?^p; \text{id}_\iota)) \rightarrow ((\text{id}_{\iota'}; \iota'!) \circ \text{id}_\star) \\ &= \text{id}_\iota \rightarrow (\text{id}_{\iota'}; \iota'!) \end{aligned}$$

These examples involve situations where an injection meets a projection by (CC-COLLAPSE) or (CC-CONFLICT). The third example is by (CC-FUN).

$$\begin{aligned} (\iota^?^p; \text{id}_\iota) \circ (\text{id}_\iota; \iota!) &= \iota^?^p; (\text{id}_\iota \circ (\text{id}_\iota; \iota!)) = \iota^?^p; ((\text{id}_\iota \circ \text{id}_\iota); \iota!) = \iota^?^p; (\text{id}_\iota; \iota!) \\ (\text{id}_\iota; \iota!) \circ (\iota^?^p; \text{id}_\iota) &= \text{id}_\iota \circ (\text{id}_\iota; \iota!) = (\text{id}_\iota \circ \text{id}_\iota); \iota! = \text{id}_\iota; \iota! \end{aligned}$$

As the fourth example shows, a projection followed by an injection does not collapse since the projection might fail. Such a coercion is simplified when it is preceded by another injection (the fifth example).

The following lemma states that composition is defined for two well-formed coercions with matching target and source types.

► **Lemma 2.** *If $s : A \rightsquigarrow B$ and $t : B \rightsquigarrow C$, then $(s \circ t) : A \rightsquigarrow C$.*

2.3.2 Evaluation

We give a small-step operational semantics to $\lambda\mathcal{S}$ consisting of two relations on closed terms: the reduction relation $M \longrightarrow_{\mathcal{S}} N$ for basic computation, and the evaluation relation $M \mapsto_{\mathcal{S}} N$ for computing subterms and raising errors.

We show the reduction rules and the evaluation rules of $\lambda\mathcal{S}$ in Figure 5. The reduction/evaluation rules are labeled either e or c. The label e is for essential computation, and the label c is for coercion applications. As we see later, this distinction is important in our correctness proof. We write $\longrightarrow_{\mathcal{S}}$ for $\xrightarrow{e}_{\mathcal{S}} \cup \xrightarrow{c}_{\mathcal{S}}$, and $\mapsto_{\mathcal{S}}$ for $\xrightarrow{e}_{\mathcal{S}} \cup \xrightarrow{c}_{\mathcal{S}}$. We sometimes call $\xrightarrow{e}_{\mathcal{S}}$ and $\xrightarrow{c}_{\mathcal{S}}$ e-evaluation and c-evaluation, respectively.

The rule (R-OP) applies to primitive operations. Here, δ is a (partial) function that takes an operator op and two constants a_1, a_2 , and returns the resulting constant of the primitive operation. We assume that if $ty(op) = \iota_1 \rightarrow \iota_2 \rightarrow \iota$ and $ty(a_1) = \iota_1$ and $ty(a_2) = \iota_2$, then $\delta(op, a_1, a_2) = a$ and $ty(a) = \iota$ for some constant a .

The rule (R-BETA) performs the standard call-by-value β -reduction. We write $M[x := V]$ for capture-avoiding substitution of V for free occurrences of x in M . The definition of substitution is standard and thus omitted.

The rule (R-WRAP) applies to applications of wrapped function $U\langle\langle s \rightarrow t \rangle\rangle$ to value V . In this case, we first apply coercion s on the argument to V , and get $V\langle s \rangle$. We next apply function U to $V\langle s \rangle$, and get $U(V\langle s \rangle)$. We then apply coercion t on the returned value, hence $(U(V\langle s \rangle))\langle t \rangle$.

The rule (R-ID) represents that identity coercion id_A returns the input value U as it is. The rule (R-FAIL) applies to applications of failure coercion \perp^{GpH} to uncoerced value U , which reduces to $\text{blame } p$. The rule (R-CRC) applies to applications $U\langle d \rangle$ of delayed coercion d to uncoerced value U , which reduces to a coerced value $U\langle\langle d \rangle\rangle$.

The rules (R-MERGE_C) and (R-MERGE_V) apply to two consecutive coercion applications, and the two coercions are merged by the composition operation. These rules are key to space efficiency. Thanks to (R-MERGE_V), we can assume that there is at most one coercion in a value. Since $d \circ t$ may or may not be a delayed coercion, the right-hand side has to be $U\langle d \circ t \rangle$, rather than $U\langle\langle d \circ t \rangle\rangle$. The outermost nested coercion applications are merged by (R-MERGE_C).

Now, we explain *evaluation contexts*, ranged over by \mathcal{E} , shown in the top of Figure 5. Following Siek et al. [37], we define them in the so-called “inside-out” style [11, 9]. Evaluation contexts represent that function calls in $\lambda\mathcal{S}$ are call-by-value and that primitive operations and function applications are evaluated from left to right. The grammar is mutually recursive with \mathcal{F} , which stands for evaluation contexts whose innermost frames are not a coercion application, whereas \mathcal{E} may contain a coercion application as the innermost frame.⁴ Careful inspection will reveal that both \mathcal{E} and \mathcal{F} contain no consecutive coercion applications. As usual, we write $\mathcal{E}[M]$ for the term obtained by replacing the hole in \mathcal{E} with M , similarly for $\mathcal{F}[M]$. (We omit their definitions.)

We present a few examples of evaluation contexts below:

$$\begin{aligned} \mathcal{F}_1 &= \square & \mathcal{E}_1 &= \mathcal{F}_1[\square\langle s \rangle] = \square\langle s \rangle \\ \mathcal{F}_2 &= \mathcal{E}_1[V\square] = (V\square)\langle s \rangle & \mathcal{E}_2 &= \mathcal{F}_2[\square\langle t \rangle] = (V(\square\langle t \rangle))\langle s \rangle \\ \mathcal{F}_3 &= \mathcal{E}_2[\square M] = (V((\square M)\langle t \rangle))\langle s \rangle \end{aligned}$$

We then come back to evaluation rules: The rules (E-CTX_E) and (E-CTX_C) enable us to evaluate the subterm in an evaluation context. Here, (E-CTX_C) requires that computation of coercion applications is only performed under contexts \mathcal{F} – otherwise, the innermost frame may be a coercion application, in which case (R-MERGE_C) has to be applied first. For example, $U\langle d \rangle\langle t \rangle$ reduces to $U\langle d \circ t \rangle$ rather than $U\langle\langle d \rangle\rangle\langle t \rangle$. The rule (E-ABORT) halts the evaluation of a program if it raises blame.

► **Example 3.** Let U be $\lambda x. (x\langle\text{int}^?^p\rangle + 2)\langle\text{int}!\rangle$. Term $((U\langle\text{int}! \rightarrow \text{int}^?^p\rangle) 3)\langle\text{int}!\rangle$ evaluates to $5\langle\langle\text{int}!\rangle\rangle$ as follows:

$$\begin{aligned} & ((U\langle\text{int}! \rightarrow \text{int}^?^p\rangle) 3)\langle\text{int}!\rangle \\ & \mapsto^* (U(3\langle\text{int}!\rangle))\langle\text{int}^?^p\rangle\langle\text{int}!\rangle && \text{by (R-CRC), (R-WRAP)} \\ & \mapsto (U(3\langle\text{int}!\rangle))\langle\text{int}^?^p; \text{id}; \text{int}!\rangle && \text{by (R-MERGE}_C\text{)} \\ & \mapsto^* (3\langle\langle\text{int}!\rangle\rangle\langle\text{int}^?^p\rangle + 2)\langle\text{int}!\rangle\langle\text{int}^?^p; \text{id}; \text{int}!\rangle && \text{by (R-CRC), (R-BETA)} \\ & \mapsto^* (3\langle\text{id}\rangle + 2)\langle\text{int}!\rangle && \text{by (R-MERGE}_C\text{), (R-MERGE}_V\text{)} \\ & \mapsto^* 5\langle\langle\text{int}!\rangle\rangle && \text{by (R-ID), (R-OP), (R-CRC)}. \end{aligned}$$

⁴ $\mathcal{F}[\square\langle s \rangle]$ (instead of $\mathcal{F}[\square\langle f \rangle]$) in the definition of \mathcal{E} fixes a problem in Siek et al. [37] that an identity coercion applied to a nonvalue gets stuck (personal communication).

2.4 Properties

We state a few important properties of $\lambda\mathcal{S}$, including determinacy of the evaluation relation and type safety via progress and preservation [46]. We write $\mapsto_{\mathcal{S}}^*$ for the reflexive and transitive closure of $\mapsto_{\mathcal{S}}$, and $\mapsto_{\mathcal{S}}^+$ for the transitive closure of $\mapsto_{\mathcal{S}}$. We say that $\lambda\mathcal{S}$ -term M *diverges*, denoted by $M \uparrow_{\mathcal{S}}$, if there exists an infinite evaluation sequence from M .

Proofs of the stated properties are in the full version.

- ▶ **Lemma 4** (Determinacy). *If $M \mapsto_{\mathcal{S}} N$ and $M \mapsto_{\mathcal{S}} N'$, then $N = N'$.*
- ▶ **Theorem 5** (Progress). *If $\emptyset \vdash_{\mathcal{S}} M : A$, then one of the following holds: (1) $M \mapsto_{\mathcal{S}} M'$ for some M' ; (2) $M = V$ for some V ; or (3) $M = \text{blame } p$ for some p .*
- ▶ **Theorem 6** (Preservation). *If $\emptyset \vdash_{\mathcal{S}} M : A$ and $M \mapsto_{\mathcal{S}} N$, then $\emptyset \vdash_{\mathcal{S}} N : A$.*
- ▶ **Corollary 7** (Type Safety). *If $\emptyset \vdash_{\mathcal{S}} M : A$, then one of the following holds: (1) $M \mapsto_{\mathcal{S}}^* V$ and $\emptyset \vdash_{\mathcal{S}} V : A$ for some V ; (2) $M \mapsto_{\mathcal{S}}^* \text{blame } p$ for some p ; or (3) $M \uparrow_{\mathcal{S}}$.*

3 Space-Efficient First-Class Coercion Calculus

In this section, we introduce $\lambda\mathcal{S}_1$, a new space-efficient coercion calculus with first-class coercions; $\lambda\mathcal{S}_1$ serves as the target calculus of the translation into coercion-passing style. The design of $\lambda\mathcal{S}_1$ is tailored to coercion-passing style and, as a result, first-class coercions are not as general as one might expect: for example, coercions for coercions are restricted to identity coercions (e.g., $\text{id}_{l \rightsquigarrow l}$).

Since coercions are first-class in $\lambda\mathcal{S}_1$, the use of (space-efficient) coercions s is not limited to coercion applications $M\langle s \rangle$; they can be passed to a function as an argument, for example. We equip $\lambda\mathcal{S}$ with the infix (object-level) operator $M ;; N$ to compute the composition of two coercions: if M and N evaluate to coercions s and t , respectively, then $M ;; N$ reduces to their composition $s \S t$, which is another space-efficient coercion. The type of (first-class) coercions from A to B is written $A \rightsquigarrow B$.⁵

In $\lambda\mathcal{S}_1$, every function abstraction takes two arguments, one of which is a parameter for a continuation coercion to be applied to the value returned from this abstraction. For example, $\lambda x. 1$ in $\lambda\mathcal{S}$ corresponds to $\lambda(x, \kappa). 1\langle \kappa \rangle$ in $\lambda\mathcal{S}_1$ — here, κ is a coercion parameter. Correspondingly, a function application takes the form $M(N, L)$, which calls function M with an argument pair (N, L) , in which L is a coercion argument, which is applied to the value returned from M . For example, $(f\ 3)\langle s \rangle$ in $\lambda\mathcal{S}$ corresponds to $f(3, s)$ in $\lambda\mathcal{S}_1$; $(f\ 3)$ (without a coercion application) corresponds to $f(3, \text{id})$.

The type of a function abstraction in $\lambda\mathcal{S}_1$ is written $A \Rightarrow B$, which means that the type of the first argument is the type A and the source type of the second coercion argument is B . An abstraction is polymorphic over the target type of the coercion argument; so, if a function of type $A \Rightarrow B$ is applied to a pair of A and $B \rightsquigarrow C$, then the type of the application will be C . Polymorphism is useful – and in fact required – for coercion-passing translation to work because coercions with different target types may be passed to calls to the same function in $\lambda\mathcal{S}$. Intuitively, $A \Rightarrow B$ means $\forall X. (A \times (B \rightsquigarrow X)) \rightarrow X$ but we do not introduce \forall -types explicitly because our use of \forall is limited to the target-type polymorphism. However, we do have to introduce type variables for typing function abstractions.

⁵ In $\lambda\mathcal{S}$, \rightsquigarrow is the symbol used in the three-place judgment form $c : A \rightsquigarrow B$, whereas \rightsquigarrow is also a type constructor in $\lambda\mathcal{S}_1$.

Variables	x, y, κ	Type variables	X, Y
Types	$A, B, C ::= \star \mid \iota \mid A \rightsquigarrow B \mid A \Rightarrow B \mid X$		
Ground types	$G, H ::= \iota \mid \star \Rightarrow \star$		
Space-efficient coercions	$s, t ::= \text{id}_\star \mid G^{?^p}; i \mid i$		
Intermediate coercions	$i ::= g; G! \mid g \mid \perp^{GpH}$		
Ground coercions	$g, h ::= \text{id}_A \text{ (if } A \neq \star) \mid s \Rightarrow t \text{ (if } s \neq \text{id or } t \neq \text{id)}$		
Delayed coercions	$d ::= g; G! \mid s \Rightarrow t \text{ (if } s \neq \text{id or } t \neq \text{id)}$		
Terms	$L, M, N ::= V \mid \text{op}(M, N) \mid L(M, N) \mid \text{let } x = M \text{ in } N$ $\mid M ;; N \mid M \langle N \rangle \mid \text{blame } p$		
Values	$V, W, K ::= x \mid U \mid U \langle d \rangle$		
Uncoerced values	$U ::= a \mid \lambda(x, \kappa). M \mid s$		
Type environments	$\Gamma ::= \emptyset \mid \Gamma, x : A$		

■ **Figure 6** Syntax of $\lambda\mathcal{S}_1$.

Following the change to function types, function coercions in $\lambda\mathcal{S}_1$ take the form $s \Rightarrow t$. Roughly speaking, its meaning is the same: it coerces an input to a function by s and coerces an output by t . However, due to the coercion passing semantics, there is slight change in how t is used at a function call. Consider $f \langle \langle s \Rightarrow t \rangle \rangle$, i.e., coercion-passing function f wrapped by coercion $s \Rightarrow t$. If the wrapped function is applied to (V, t') , V is coerced by s before passing to f as in $\lambda\mathcal{S}$; instead of coercing the return value by t , however, t is prepended to t' and passed to f (together with the coerced V) so that the return value is coerced by t and then t' . In the reduction rule, prepending t to t' is represented by composition $t ;; t'$.

3.1 Syntax

We show the syntax of $\lambda\mathcal{S}_1$ in Figure 6. We reuse the same metavariables from $\lambda\mathcal{S}$. We also use κ for variables, and K for values.

We replace $A \rightarrow B$ with $A \Rightarrow B$ and add $A \rightsquigarrow B$ and type variables to types. The syntax for ground types and space-efficient, intermediate, ground, and delayed coercions is the same except that \rightarrow is replaced with \Rightarrow , similarly to types. As we have mentioned, we replace abstractions and applications with two-argument versions. We also add let-expressions (although they could be introduced as derived forms) and coercion composition $M ;; N$. The syntax for coercion applications is now $M \langle N \rangle$, where N is a general term (of type $A \rightsquigarrow B$). Uncoerced values now include space-efficient coercions.

The term $\lambda(x, \kappa). M$ binds x and κ in M , and the term $\text{let } x = M \text{ in } N$ binds x in N . The definitions of free variables and α -equivalence of terms are standard, and thus we omit them. We identify α -equivalent terms.

The definition of type environments, ranged over by Γ , is the same as $\lambda\mathcal{S}$.

3.2 Type System

Figure 7 shows the main typing rules of $\lambda\mathcal{S}_1$, which are a straightforward adaption from $\lambda\mathcal{S}$.

The relation $c : A \rightsquigarrow B$ is mostly the same as that of $\lambda\mathcal{S}$. We replace the rule (CT-FUN) as shown. As in $\lambda\mathcal{S}$, function coercions are contravariant in their argument coercions and covariant in their return-value coercions.

Well-formed coercions (replacement)

$$\boxed{c : A \rightsquigarrow B}$$

$$\frac{c_1 : A' \rightsquigarrow A \quad c_2 : B \rightsquigarrow B'}{c_1 \Rightarrow c_2 : A \Rightarrow B \rightsquigarrow A' \Rightarrow B'} \text{CT-FUN}$$

Term typing (excerpt)

$$\boxed{\Gamma \vdash_{\mathcal{S}_1} M : A}$$

$$\frac{s : A \rightsquigarrow B}{\Gamma \vdash s : A \rightsquigarrow B} \text{T-CRCN} \quad \frac{\Gamma \vdash M : A \rightsquigarrow B \quad \Gamma \vdash N : B \rightsquigarrow C}{\Gamma \vdash M ;; N : A \rightsquigarrow C} \text{T-CMP}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : A \rightsquigarrow B}{\Gamma \vdash M \langle N \rangle : B} \text{T-CRC} \quad \frac{\emptyset \vdash U : A \quad \emptyset \vdash d : A \rightsquigarrow B}{\emptyset \vdash U \langle\langle d \rangle\rangle : B} \text{T-CRCV}$$

$$\frac{\Gamma, x : A, \kappa : B \rightsquigarrow X \vdash M : X \quad (X \text{ does not appear in } \Gamma, A, B)}{\Gamma \vdash \lambda(x, \kappa). M : A \Rightarrow B} \text{T-ABS}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma, x : A \vdash N : B}{\Gamma \vdash \text{let } x = M \text{ in } N : B} \text{T-LET} \quad \frac{\Gamma \vdash L : A \Rightarrow B \quad \Gamma \vdash M : A \quad \Gamma \vdash N : B \rightsquigarrow C}{\Gamma \vdash L(M, N) : C} \text{T-APP}$$

■ **Figure 7** Typing rules of $\lambda\mathcal{S}_1$.

The judgment $\Gamma \vdash_{\mathcal{S}_1} M : A$ means that term M of $\lambda\mathcal{S}_1$ has type A under type environment Γ . The rules (T-CONST), (T-OP), (T-VAR), and (T-BLAME) are the same as $\lambda\mathcal{S}$, and so we omit them. The rule (T-LET) is standard.

The rules (T-ABS) and (T-APP) look involved but the intuition that $A \Rightarrow B$ corresponds to $\forall X. (A \times (B \rightsquigarrow X)) \rightarrow X$ should help to understand them. The rule (T-ABS) assigns type $A \Rightarrow B$ to an abstraction $\lambda(x, \kappa). M$ if the body is well typed under the assumption that x is of type A and κ is of type $B \rightsquigarrow X$ for fresh X . The type variable X must not appear in Γ, A, B so that the target type can be polymorphic at call sites. The rule (T-APP) for applications is already explained.

The rule (T-CRCN) assigns type $A \rightsquigarrow B$ to space-efficient coercion s if it converts a value from type A to type B . The rules (T-CRC) and (T-CRCV) are similar to the corresponding rules of $\lambda\mathcal{S}$, but adjusted to first-class coercions.

3.3 Operational Semantics

The composition function $s \circledast t$ is mostly the same as that of $\lambda\mathcal{S}$. We only replace (CC-FUN) as shown in Figure 8.

Similarly to $\lambda\mathcal{S}$, we give a small-step operational semantics to $\lambda\mathcal{S}_1$ consisting of two relations on closed terms: the reduction relation $M \longrightarrow_{\mathcal{S}_1} N$ and the evaluation relation $M \mapsto_{\mathcal{S}_1} N$. We show the reduction/evaluation rules of $\lambda\mathcal{S}_1$ in Figure 8. As in $\lambda\mathcal{S}$, they are labeled either e or c. We write $\longrightarrow_{\mathcal{S}_1}$ for $\xrightarrow{e}_{\mathcal{S}_1} \cup \xrightarrow{c}_{\mathcal{S}_1}$, and $\mapsto_{\mathcal{S}_1}$ for $\xrightarrow{e}_{\mathcal{S}_1} \cup \xrightarrow{c}_{\mathcal{S}_1}$.

The rules (R-OP) and (R-BETA) are standard. Note that (R-BETA) is adjusted for pair arguments. We write $M[x := V, \kappa := K]$ for capture-avoiding simultaneous substitution of V and K for x and κ , respectively, in M .

The rule (R-WRAP) applies to applications of wrapped function $U \langle\langle s \Rightarrow t \rangle\rangle$ to value V . Since coercion s is for function arguments, it is applied to V , as in $\lambda\mathcal{S}$. Additionally, we compose coercion t on the return value with continuation coercion W . Thus, $V \langle s \rangle$ and $t ;; W$ are passed to function U . Note that we use a let expression to evaluate the second argument $t ;; W$ before $V \langle s \rangle$. It is a necessary adjustment for the semantics of $\lambda\mathcal{S}$ and $\lambda\mathcal{S}_1$ to match.

Coercion composition (replacement)

$$s \mathbin{\&} t = s'$$

$$(s \Rightarrow t) \mathbin{\&} (s' \Rightarrow t') = \begin{cases} \text{id}_{A \Rightarrow B} & \text{if } s' \mathbin{\&} s = \text{id}_A \text{ and } t \mathbin{\&} t' = \text{id}_B \\ (s' \mathbin{\&} s) \Rightarrow (t \mathbin{\&} t') & \text{otherwise} \end{cases} \quad \text{CC-FUN}$$

Evaluation contexts

$$\mathcal{E} ::= \square \mid \mathcal{E}[\square(M, N)] \mid \mathcal{E}[V(\square, N)] \mid \mathcal{E}[V(W, \square)] \mid \mathcal{E}[op(\square, M)] \mid \mathcal{E}[op(V, \square)] \\ \mid \mathcal{E}[\text{let } x = \square \text{ in } M] \mid \mathcal{E}[\square ;; M] \mid \mathcal{E}[V ;; \square] \mid \mathcal{E}[\square \langle M \rangle] \mid \mathcal{E}[V \langle \square \rangle]$$

Reduction

$$M \xrightarrow{e}_{\mathcal{S}_1} N \quad M \xrightarrow{c}_{\mathcal{S}_1} N$$

$$\begin{array}{ll} op(a, b) \xrightarrow{e} \delta(op, a, b) & \text{R-OP} \\ (\lambda(x, \kappa). M)(V, W) \xrightarrow{e} M[x := V, \kappa := W] & \text{R-BETA} \\ (U \langle \langle s \Rightarrow t \rangle \rangle)(V, W) \xrightarrow{e} \text{let } \kappa = t ;; W \text{ in } U(V \langle s \rangle, \kappa) & \text{R-WRAP} \end{array}$$

$$\begin{array}{ll} \text{let } x = V \text{ in } M \xrightarrow{c} M[x := V] & \text{R-LET} & s ;; t \xrightarrow{c} s \mathbin{\&} t & \text{R-CMP} \\ U \langle \text{id}_A \rangle \xrightarrow{c} U & \text{R-ID} & U \langle \perp^{GpH} \rangle \xrightarrow{c} \text{blame } p & \text{R-FAIL} \\ U \langle d \rangle \xrightarrow{c} U \langle \langle d \rangle \rangle & \text{R-CRC} & U \langle \langle d \rangle \rangle \langle t \rangle \xrightarrow{c} U \langle d ;; t \rangle & \text{R-MERGEV} \end{array}$$

Evaluation

$$M \vdash^e_{\mathcal{S}_1} N \quad M \vdash^c_{\mathcal{S}_1} N$$

$$\frac{M \xrightarrow{x} N \quad \mathcal{X} \in \{\mathbf{e}, \mathbf{c}\}}{\mathcal{E}[M] \vdash^x \mathcal{E}[N]} \text{E-CTX} \quad \frac{\mathcal{E} \neq \square}{\mathcal{E}[\text{blame } p] \vdash^e \text{blame } p} \text{E-ABORT}$$

■ **Figure 8** Reduction/evaluation rules of $\lambda\mathcal{S}_1$.

The rule (R-LET) is standard; it is labeled as **c** because we use let-expressions only for coercion compositions. The rule (R-CMP) applies to coercion compositions $s ;; t$, which is evaluated by meta-level coercion composition function $s \mathbin{\&} t$. The rules (R-ID), (R-FAIL), (R-CRC), and (R-MERGEV) are the same as $\lambda\mathcal{S}$.

Evaluation contexts, ranged over by \mathcal{E} , are defined also in Figure 8. In contrast to $\lambda\mathcal{S}$, evaluation contexts are standard in $\lambda\mathcal{S}_1$. The definition represents that function calls in $\lambda\mathcal{S}_1$ are call-by-value, and primitive operations, function applications, coercion compositions, and coercion applications are all evaluated from left to right.

We then come back to evaluation rules: The evaluation rules (E-CTX) and (E-ABORT) are the same as $\lambda\mathcal{S}$. (However, evaluation contexts in (E-CTX) are more straightforward in $\lambda\mathcal{S}_1$.)

Finally, we should emphasize that we no longer need (R-MERGEV) in $\lambda\mathcal{S}_1$. So, $\lambda\mathcal{S}_1$ is an ordinary call-by-value language and its semantics should be easy to implement.

► **Example 8.** Let U be $\lambda(x, \kappa). \text{let } \kappa' = \text{int}! ;; \kappa \text{ in } (x \langle \text{int}^{?p} \rangle + 2) \langle \kappa' \rangle$, which corresponds to the $\lambda\mathcal{S}$ -term $\lambda x. (x \langle \text{int}^{?p} \rangle + 2) \langle \text{int}! \rangle$ in Example 3. In fact, we will obtain this term as a result of our coercion-passing translation defined in the next section. The term $(U \langle \text{int}! \Rightarrow \text{int}^{?p} \rangle)(3, \text{int}!)$ evaluates to $5 \langle \langle \text{int}! \rangle \rangle$ as follows:

$$\begin{aligned}
& (U\langle \text{int!} \Rightarrow \text{int}^?^p \rangle) (3, \text{int!}) \\
& \longmapsto^* \text{let } \kappa'' = \text{int}^?^p ;; \text{int! in } U (3\langle \text{int!} \rangle, \kappa'') && \text{by (R-CRC), (R-WRAP)} \\
& \longmapsto \text{let } \kappa'' = \text{int}^?^p; \text{id; int! in } U (3\langle \text{int!} \rangle, \kappa'') && \text{by (R-CMP)} \\
& \longmapsto^* U (3\langle\langle \text{int!} \rangle\rangle, (\text{int}^?^p; \text{id; int!})) && \text{by (R-LET), (R-CRC)} \\
& \longmapsto \text{let } \kappa' = \text{int!} ;; (\text{int}^?^p; \text{id; int!}) \text{ in } (3\langle\langle \text{int!} \rangle\rangle\langle \text{int}^?^p \rangle + 2)\langle \kappa' \rangle && \text{by (R-BETA)} \\
& \longmapsto^* (3\langle\langle \text{int!} \rangle\rangle\langle \text{int}^?^p \rangle + 2)\langle \text{int!} \rangle && \text{by (R-CMP), (R-LET)} \\
& \longmapsto^* 5\langle\langle \text{int!} \rangle\rangle && \text{by (R-MERGEV), (R-ID), (R-OP), (R-CRC)}
\end{aligned}$$

It is easy to see that the steps by (R-MERGEV) in Example 3 are simulated by (R-CMP) followed by (R-LET).

3.4 Properties

We state a few properties of $\lambda\mathcal{S}_1$ below. Their proofs are in the full version.

- **Lemma 9** (Determinacy). *If $M \mapsto_{\mathcal{S}_1} N$ and $M \mapsto_{\mathcal{S}_1} N'$, then $N = N'$.*
- **Theorem 10** (Progress). *If $\emptyset \vdash_{\mathcal{S}_1} M : A$, then one of the following holds: (1) $M \mapsto_{\mathcal{S}_1} M'$ for some M' ; (2) $M = V$ for some V ; or (3) $M = \text{blame } p$ for some p .*
- **Theorem 11** (Preservation). *If $\emptyset \vdash_{\mathcal{S}_1} M : A$ and $M \mapsto_{\mathcal{S}_1} N$, then $\emptyset \vdash_{\mathcal{S}_1} N : A$.*
- **Corollary 12** (Type Safety). *If $\emptyset \vdash_{\mathcal{S}_1} M : A$, then one of the following holds: (1) $M \mapsto_{\mathcal{S}_1}^* V$ and $\emptyset \vdash_{\mathcal{S}_1} V : A$ for some V ; (2) $M \mapsto_{\mathcal{S}_1}^* \text{blame } p$ for some p ; or (3) $M \uparrow_{\mathcal{S}_1}$.*

4 Translation into Coercion-Passing Style

In this section, we formalize a translation into coercion-passing style as a translation from $\lambda\mathcal{S}$ to $\lambda\mathcal{S}_1$ and state its correctness. As its name suggests, this translation is similar to transformations into continuation-passing style (CPS transformations) for the call-by-value λ -calculus [28].

4.1 Definition of Translation

We give the translation into coercion-passing style by the translation rules presented in Figure 9. In order to distinguish metavariables of $\lambda\mathcal{S}$ and $\lambda\mathcal{S}_1$, we often use [blue](#) for the source calculus $\lambda\mathcal{S}$. When we need static type information in translation rules, we write M^A to indicate that term M has type A . Thus, strictly speaking, the translation is defined for type derivations in $\lambda\mathcal{S}$.

Translations for types $\Psi(A)$ and coercions $\Psi(s)$ are very straightforward, thanks to the special type/coercion constructor \Rightarrow : they just recursively replace \rightarrow with \Rightarrow .

Value translation $\Psi(V)$ and term translation $\mathcal{K}[[M]]K$ are defined in a mutually recursive manner. In $\mathcal{K}[[M]]K$, M is a $\lambda\mathcal{S}$ -term whereas K is a $\lambda\mathcal{S}_1$ -term, which is either a variable or a $\lambda\mathcal{S}_1$ -coercion. $\mathcal{K}[[M]]K$ returns a $\lambda\mathcal{S}_1$ -term – in coercion-passing style – that applies K to the value of M .

Value translation $\Psi(V)$ is straightforward: every function $\lambda x. M$ is translated to a $\lambda\mathcal{S}_1$ -abstraction that takes as the second argument κ a coercion which is to be applied to the return value. So, the body is translated by term translation $\mathcal{K}[[M]]\kappa$.

Type translation

$$\Psi(A) = A'$$

$$\Psi(\star) = \star \quad \Psi(\iota) = \iota \quad \Psi(A \rightarrow B) = \Psi(A) \Rightarrow \Psi(B)$$

Coercion translation

$$\Psi(s) = s'$$

Value translation

$$\Psi(V) = V'$$

$$\Psi(\text{id}_A) = \text{id}_{\Psi(A)}$$

$$\Psi(g; G!) = \Psi(g); \Psi(G)!$$

$$\Psi(G^{?^p}; i) = \Psi(G)^{?^p}; \Psi(i)$$

$$\Psi(s \rightarrow t) = \Psi(s) \Rightarrow \Psi(t)$$

$$\Psi(\perp^{GpH}) = \perp^{GpH}$$

$$\Psi(x) = x$$

$$\Psi(a) = a$$

$$\Psi(\lambda x. M) = \lambda(x, \kappa). (\mathcal{K} \llbracket M \rrbracket \kappa)$$

$$\Psi(U \langle\langle d \rangle\rangle) = \Psi(U) \langle\langle \Psi(d) \rangle\rangle$$

Term translation

$$\mathcal{C} \llbracket M \rrbracket = M'$$

$$\mathcal{K} \llbracket M \rrbracket K = M'$$

$$\mathcal{C} \llbracket V \rrbracket = \Psi(V)$$

TRC-VAL

$$\mathcal{C} \llbracket M \langle s \rangle \rrbracket = \mathcal{K} \llbracket M \rrbracket \Psi(s)$$

TRC-CRC

$$\mathcal{C} \llbracket M^A \rrbracket = \mathcal{K} \llbracket M \rrbracket \text{id}_{\Psi(A)}$$

otherwise

TRC-ELSE

$$\mathcal{K} \llbracket V \rrbracket K = \Psi(V) \langle K \rangle$$

TR-VAL

$$\mathcal{K} \llbracket \text{op}(M, N) \rrbracket K = \text{op}(\mathcal{C} \llbracket M \rrbracket, \mathcal{C} \llbracket N \rrbracket) \langle K \rangle$$

TR-OP

$$\mathcal{K} \llbracket M N \rrbracket K = (\mathcal{C} \llbracket M \rrbracket) (\mathcal{C} \llbracket N \rrbracket, K)$$

TR-APP

$$\mathcal{K} \llbracket M \langle s \rangle \rrbracket K = \text{let } \kappa = \Psi(s) ;; K \text{ in } (\mathcal{K} \llbracket M \rrbracket \kappa)$$

TR-CRC

$$\mathcal{K} \llbracket \text{blame } p \rrbracket K = \text{blame } p$$

TR-BLAME

■ **Figure 9** Translation into coercion-passing style (from $\lambda\mathcal{S}$ to $\lambda\mathcal{S}_1$).

We now describe the translation for terms. We write $\mathcal{K} \llbracket M \rrbracket K$ for the translation of $\lambda\mathcal{S}$ -term M with continuation coercion K . We first explain the basic transformation scheme given by the recursive function \mathcal{K}' defined by the following simpler rules:

$$\mathcal{K}' \llbracket V \rrbracket K = \Psi(V) \langle K \rangle$$

TR'-VAL

$$\mathcal{K}' \llbracket \text{op}(M^{\iota_1}, N^{\iota_2}) \rrbracket K = \text{op}(\mathcal{K}' \llbracket M \rrbracket \text{id}_{\iota_1}, \mathcal{K}' \llbracket N \rrbracket \text{id}_{\iota_2}) \langle K \rangle$$

TR'-OP

$$\mathcal{K}' \llbracket M^{A \rightarrow B} N^A \rrbracket K = (\mathcal{K}' \llbracket M \rrbracket \text{id}_{\Psi(A \rightarrow B)}) (\mathcal{K}' \llbracket N \rrbracket \text{id}_{\Psi(A)}, K)$$

TR'-APP

$$\mathcal{K}' \llbracket M \langle s \rangle \rrbracket K = \text{let } \kappa = \Psi(s) ;; K \text{ in } (\mathcal{K}' \llbracket M \rrbracket \kappa)$$

TR'-CRC

$$\mathcal{K}' \llbracket \text{blame } p \rrbracket K = \text{blame } p$$

TR'-BLAME

(We put a prime on \mathcal{K} to distinguish with the final version.)

The rule (TR'-VAL) applies to values V , where we apply coercion K to the result of value translation $\Psi(V)$.

The rule (TR'-OP) applies to primitive operations $\text{op}(M, N)$. We translate the arguments M and N with identity continuation coercions by $\mathcal{K}' \llbracket M \rrbracket \text{id}$ and $\mathcal{K}' \llbracket N \rrbracket \text{id}$ and pass them to the primitive operation. The given continuation coercion K is applied to the result. Translating subexpressions with id is one of the main differences from CPS transformation. While continuations in continuation-passing style capture the whole rest of computation, continuation coercions in coercion-passing style capture only the coercion applied right after the current computation. Since neither M nor N is surrounded by a coercion, they are translated with identity coercions of appropriate types. (Cases where a subexpression itself is a coercion application will be discussed shortly.) Careful readers may notice at this point that left-to-right evaluation of arguments is enforced by the semantics (or the definition of

evaluation contexts) of $\lambda\mathcal{S}$, not by the translation. In other words, the correctness of the translation relies on the fact that $\lambda\mathcal{S}$ evaluation is left-to-right and call-by-value. This is another point that is different from CPS transformation, which dismisses the distinction of call-by-name and call-by-value.

The rule (TR'-APP) applies to function applications $M N$. We translate function M and argument N with identity continuation coercions just like the case for primitive operations. We then pass the continuation coercion K as the second argument to function $\mathcal{K}'\llbracket M \rrbracket \text{id}$.

The rule (TR'-CRC) applies to coercion applications $M\langle s \rangle$. We can think of the sequential composition of $\Psi(s)$ and K as the continuation coercion for M . Thus, we first compute the composition $\Psi(s) ;; K$, bind its result to κ , and translate M with continuation κ . The let-expression is necessary to compose $\Psi(s)$ and K before evaluating $\mathcal{K}'\llbracket M \rrbracket \kappa$. In general, it is not necessarily the case that $\mathcal{K}'\llbracket M \rrbracket K$ evaluates K first, so if we set $\mathcal{K}'\llbracket M\langle s \rangle \rrbracket K = (\mathcal{K}'\llbracket M \rrbracket (\Psi(s) ;; K))$, then the order of computation would change by the translation and correctness of translation would be harder to show.

Lastly, the rule (TR'-BLAME) means that continuation K is discarded for blame p .

The translation \mathcal{K}' seems acceptable but, just as naïve CPS transformation leaves administrative redexes, it leaves many applications of id , which we call *administrative coercions*. We expect M and $\mathcal{K}'\llbracket M \rrbracket K$ to “behave similarly” but administrative redexes make it hard to show such semantic correspondence. Therefore, we will optimize the translation so that administrative coercions are eliminated, similarly to CPS transformations that eliminate administrative redexes [28, 3, 45, 32, 10, 8, 33].

The bottom of Figure 9 shows the optimized translation rules. The idea to eliminate administrative coercions is close to the colon translation by Plotkin [28]: we avoid translating values with administrative coercions. So, we introduce an auxiliary translation function $\mathcal{C}\llbracket M \rrbracket$, which, if M is a value V , returns $\Psi(V)$ – without a coercion application – and, if M is a coercion application $N\langle s \rangle$, returns $\mathcal{K}\llbracket N \rrbracket \Psi(s)$ – with the trivial composition $\Psi(s) ; \text{id}$ optimized away – and returns $\mathcal{K}\llbracket M \rrbracket \text{id}$ otherwise. Translation rules for primitive operations and function applications are adapted so that they use $\mathcal{C}\llbracket M \rrbracket$ to translate subexpressions.

In other words, $\mathcal{C}\llbracket M \rrbracket$ helps us precisely distinguish between id introduced by the translation and id that was present in the original term. Whenever we introduce id as an initial coercion for the translation, we first apply $\mathcal{C}\llbracket M \rrbracket$ and then apply $\mathcal{K}\llbracket M \rrbracket \text{id}$ only if necessary. We note that $\mathcal{K}\llbracket M \rrbracket \text{id} \mapsto_{\mathcal{S}_1} \mathcal{C}\llbracket M \rrbracket$ holds. We present a few examples of the translation below:

$$\begin{aligned} \Psi(\lambda x. x + 1) &= \lambda(x, \kappa). (x + 1)\langle \kappa \rangle \\ \mathcal{K}\llbracket (\lambda x. x) 5 \rrbracket \text{id} &= (\lambda(x, \kappa). x\langle \kappa \rangle) (5, \text{id}) \\ \mathcal{K}\llbracket ((\lambda x. x) 5)\langle \text{id} \rangle \rrbracket \text{id} &= \text{let } \kappa = \text{id} ! ;; \text{id}^{?p} \text{ in } (\lambda(x, \kappa). x\langle \kappa \rangle) (5, \kappa) \end{aligned}$$

The following example shows the translation of the $\lambda\mathcal{S}$ -term in Example 3 will be the $\lambda\mathcal{S}_1$ -term in Example 8.

► **Example 13.** Let U be a $\lambda\mathcal{S}$ -term $\lambda x. (x\langle \text{id}^{?p} \rangle + 2)\langle \text{id} \rangle$.

$$\begin{aligned} \Psi(U) &= \lambda(x, \kappa). (\mathcal{K}\llbracket (x\langle \text{id}^{?p} \rangle + 2)\langle \text{id} \rangle \rrbracket \kappa) \\ &= \lambda(x, \kappa). \text{let } \kappa' = \text{id} ! ;; \kappa \text{ in } (\mathcal{K}\llbracket (x\langle \text{id}^{?p} \rangle + 2) \rrbracket \kappa') \\ &= \lambda(x, \kappa). \text{let } \kappa' = \text{id} ! ;; \kappa \text{ in } (x\langle \text{id}^{?p} \rangle + 2)\langle \kappa' \rangle \\ \mathcal{K}\llbracket ((U\langle \text{id} \rangle \rightarrow \text{id}^{?p}) 3) \rrbracket \text{id} &= (\mathcal{K}\llbracket (U\langle \text{id} \rangle \rightarrow \text{id}^{?p}) \rrbracket \text{id}) (\mathcal{K}\llbracket 3 \rrbracket \text{id}, \text{id}) \\ &= (\mathcal{K}\llbracket U \rrbracket (\text{id} \rightarrow \text{id}^{?p})) (3, \text{id}) \\ &= (\Psi(U)\langle \text{id} \rangle \Rightarrow \text{id}^{?p}) (3, \text{id}) \end{aligned}$$

4.2 Correctness of Translation

Having defined the translation, we now state its correctness properties with auxiliary lemmas. (Their proofs are in the full version.)

To begin with, the translation preserves typing. Here, we write $\Psi(\Gamma)$ for the type environment satisfying: $(x : A) \in \Gamma$ if and only if $(x : \Psi(A)) \in \Psi(\Gamma)$.

► **Theorem 14** (Translation Preserves Typing).

1. If $\Gamma \vdash_S M : A$ and $s : A \rightsquigarrow B$, then $\Psi(\Gamma) \vdash_{S_1} (\mathcal{N} \llbracket M \rrbracket \Psi(s)) : \Psi(B)$.
2. If $\Gamma \vdash_S V : A$, then $\Psi(\Gamma) \vdash_{S_1} \Psi(V) : \Psi(A)$.

As for the preservation of semantics, we will prove the following theorem that states the semantics is preserved by the translation:

► **Theorem 15** (Translation Preserves Semantics). *If $\emptyset \vdash_S M : \iota$, then (1) $M \mapsto_S^* a$ iff $\mathcal{C} \llbracket M \rrbracket \mapsto_{S_1}^* a$; (2) $M \mapsto_S^* \text{blame } p$ iff $\mathcal{C} \llbracket M \rrbracket \mapsto_{S_1}^* \text{blame } p$; and (3) $M \uparrow_S$ iff $\mathcal{C} \llbracket M \rrbracket \uparrow_{S_1}$.*

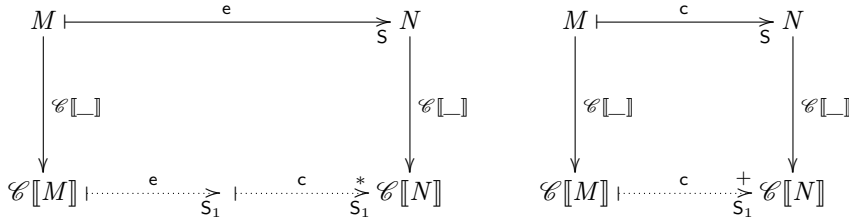
To prove this theorem, it suffices to show the left-to-right direction (Theorem 16 below) for each item because the other direction follows from Theorem 16 together with other properties: for example, if $\emptyset \vdash_S M : \iota$ and $\mathcal{C} \llbracket M \rrbracket \uparrow_{S_1}$, then M can neither get stuck (by type soundness of λS) nor terminate (as it contradicts the left-to-right direction and the fact that \mapsto_S is deterministic).

► **Theorem 16** (Translation Soundness). *Suppose $\Gamma \vdash_S M : A$. (1) If $M \mapsto_S^* V$, then $\mathcal{C} \llbracket M \rrbracket \mapsto_{S_1}^* \Psi(V)$; (2) if $M \mapsto_S^* \text{blame } p$, then $\mathcal{C} \llbracket M \rrbracket \mapsto_{S_1}^* \text{blame } p$; and (3) if $M \uparrow_S$, then $\mathcal{C} \llbracket M \rrbracket \uparrow_{S_1}$.*

A standard proof strategy would be to show that single-step evaluation in the source language is simulated by multi-step evaluation in the target language. In fact, we prove the following lemma:

► **Lemma 17** (Simulation).

1. If $M \mapsto_S^e N$, then $\mathcal{C} \llbracket M \rrbracket \mapsto_{S_1}^e \mathcal{C} \llbracket N \rrbracket$.
2. If $M \mapsto_S^c N$, then $\mathcal{C} \llbracket M \rrbracket \mapsto_{S_1}^c \mathcal{C} \llbracket N \rrbracket$.



The straightforward simulation property below follows from Lemma 17.

► **Lemma 18.** *If $M \mapsto_S N$, then $\mathcal{C} \llbracket M \rrbracket \mapsto_{S_1}^+ \mathcal{C} \llbracket N \rrbracket$.*

As is the case for simulation proofs for CPS translation [28, 3, 45, 32, 10, 8, 33], the simulation property⁶ is quite subtle. We discuss this subtlety below.

First, it is important that the translation removes administrative identity coercions by distinguishing values and nonvalues in $\mathcal{C} \llbracket M \rrbracket$. For example, $(\lambda x. x) 5 \mapsto_S 5$ holds in λS , but the translation $\mathcal{N} \llbracket (\lambda x. x) 5 \rrbracket K$ without removing administrative redexes would yield

⁶ If we had been interested only in the property that translation preserves term equivalence, we could have simplified the technical development by, say, removing the distinction between $U \langle s \rangle$ and $U \langle\langle s \rangle\rangle$. However, simulation is crucial for showing that divergence is preserved by the translation.

$((\lambda(x, \kappa). x \langle \kappa \rangle) \langle \text{id} \rangle) (5 \langle \text{id} \rangle, K)$, which performs c-evaluation before calling the function. We avoid such a situation. More formally, we prove the following lemma, which means the redex in the source is also the redex in the target.

► **Lemma 19.**

1. For any \mathcal{F} , there exists \mathcal{E}' such that for any M , $\mathcal{C}[\mathcal{F}[M]] = \mathcal{E}'[\mathcal{C}[M]]$.
2. For any \mathcal{F} and s , there exists \mathcal{E}' such that for any M , $\mathcal{C}[\mathcal{F}[M \langle s \rangle]] = \mathcal{E}'[\mathcal{C}[M] \Psi(s)]$.

To prove this lemma, the rule (TRC-CRC) also plays an important role: for example, if we removed (TRC-CRC), $\mathcal{C}[(1 + 1) \langle \text{int}! \rangle] \text{id}$ would translate to $\text{let } \kappa = \text{int}! ;; \text{id in } (1 + 1) \langle \kappa \rangle$, which performs c-evaluation before adding 1 and 1, which is the first thing the original term $(1 + 1) \langle \text{int}! \rangle$ will do.

Second, optimizing too many (identity) coercions can break simulation. We should only remove administrative identity coercions, and keep identity coercions that were present in the original term. Consider $M \stackrel{\text{def}}{=} ((\lambda x. M_1) \langle \text{id}_\iota \rightarrow \iota! \rangle) a \langle \iota?^p \rangle$ and $N \stackrel{\text{def}}{=} ((\lambda x. M_1) (a \langle \text{id}_\iota \rangle)) \langle \iota! \rangle \langle \iota?^p \rangle$, for which $M \mapsto_{\mathcal{S}} N$ holds by (R-WRAP). Then,

$$\begin{aligned} \mathcal{C}[M] &= \mathcal{C}[\mathcal{C}[M] \text{id}] = ((\mathcal{C}[\lambda(x, \kappa). M_1] \kappa) \langle \text{id}_\iota \Rightarrow \iota! \rangle) (a, \iota?^p) \\ &\mapsto_{\mathcal{S}_1} \text{let } \kappa' = \iota! ;; \iota?^p \text{ in } (\mathcal{C}[\lambda(x, \kappa). M_1] \kappa) (a \langle \text{id}_\iota \rangle, \kappa') = \mathcal{C}[N]. \end{aligned}$$

At one point, we defined the translation (let's call it \mathcal{C}'') so that applications of identity coercions would be removed as much as possible, namely,

$$\mathcal{C}''[N] \text{id} = \text{let } \kappa' = \iota! ;; \iota?^p \text{ in } (\mathcal{C}''[\lambda(x, \kappa). M_1] \kappa) (a, \kappa')$$

(notice that $\langle \text{id}_\iota \rangle$ on a is removed). Although $\mathcal{C}''[M] \text{id}$ and $\mathcal{C}''[N] \text{id}$ reduced to the same term, we did not quite have $\mathcal{C}''[M] \text{id} \mapsto^+ \mathcal{C}''[N] \text{id}$ as we had desired.

Third, the distinction between $U \langle s \rangle$ and $U \langle \langle s \rangle \rangle$ is crucial for ensuring that substitution commutes with the translation:

► **Lemma 20 (Substitution).** *If $\kappa \notin FV(M) \cup FV(V)$, then $(\mathcal{C}[M] \kappa)[x := \Psi(V), \kappa := K] = \mathcal{C}[M[x := V]] K$.*

Roughly speaking, if we identified a value $U \langle \langle s \rangle \rangle$ and an application $U \langle s \rangle$ of s to an uncoerced value U , then the term $U \langle s \rangle \langle t \rangle$ would allow two interpretations: an application of t to a value $U \langle s \rangle$ and applications of s and t to U and committing to either interpretation would break Lemma 20.

5 Implementation and Evaluation

5.1 Implementation

We have implemented the coercion-passing translation described in Section 4 and the semantics of $\lambda\mathcal{S}_1$ for Grift [24]⁷, an experimental compiler for gradually typed languages. GTLC+, the language that the Grift compiler implements, supports integers, floating-point numbers, Booleans, higher-order functions, local binding by **let**, (mutually) recursive definitions by **letrec**, conditional expressions, iterations, sequencing, mutable references, and vectors (mutable arrays).

⁷ The semantics of coercions in Grift is so-called D [35], which is slightly different from that of $\lambda\mathcal{S}_1$, which is UD. Since the main difference is in the coercion composition, our technique can be applied to Grift.

The Grift compiler compiles a GTLC+ program into the C language where coercions are represented as values of a `struct` type, and operations such as coercion application and coercion composition are C functions. The compiler supports different run-time check schemes, those based on type-based casts [36] and space-efficient coercions [37]. Note that, although meta-level composition $s_1 \circ s_2$ is implemented, only nested coercions on *values* are composed; in other words, (R-MERGE) was not implemented. Thus, implicit run-time checks may break tail calls and seemingly tail-recursive functions may cause stack overflow.

We modify the compiler phases for run-time checking based on the space-efficient coercions. After typechecking a user program, the compiler inserts type-based casts to the program and converts type-based casts to space-efficient coercions, following the translation from blame calculus λB to λS [37]. Our implementation performs the coercion-passing translation after the translation into λS . It is straightforward to extend the translation scheme to language features that are not present in λS . For example, here is translation for conditional expressions:

$$\mathcal{K}[\text{if } M \text{ then } N_1 \text{ else } N_2]K = \text{if } \mathcal{C}[\![M]\!] \text{ then } (\mathcal{K}[\![N_1]\!]K) \text{ else } (\mathcal{K}[\![N_2]\!]K).$$

Since coercions are represented as `structs`, we did not have to do anything special to make coercions first-class. We modify another compiler phase that generates operations on coercions such as $M \circ N$ and (R-WRAP). The current implementation, which generates C code and uses `clang`⁸ for compilation to machine code, relies on the C compiler to perform tail-call optimizations. We have found the original compiler’s handling of recursive types hampers tail-call optimizations,⁹ so our implementation does not deal with recursive types. We leave their implementation for future work.

5.2 Even and Odd Functions

We first inspected the tail-recursive even–odd functions in GTLC+:

```
(letrec ([even (lambda ([n : A1]) : A3
  (if (= 0 n) #t (odd (- n 1))))]
 [odd (lambda ([n : A2]) : A4
  (if (= 0 n) #f (even (- n 1))))])
 (odd n))
```

where A_1 and A_2 are either `Int` or `Dyn`, and A_3 and A_4 are either `Bool` or `Dyn`. We run this program with the original and modified compilers for all combinations of A_1, A_2, A_3 , and A_4 . We call the program compiled by the original compiler `Base`, the program compiled by the modified compiler `CrcPS`.

We have confirmed that, as n increases, 12 of 16 configurations of `Base` cause stack overflow.¹⁰ In the four configurations that survived, both A_3 and A_4 are set to `Bool`. `CrcPS` never causes stack overflow for any configuration.

Although we expected that `Base` would crash if A_3 and A_4 are different, it is our surprise that `Base` causes stack overflow even when $A_3 = A_4 = \text{Dyn}$. We have found that it is due to the typing rule of Grift for conditional expressions. In Grift, if one of the branches is given a

⁸ <https://clang.llvm.org/>

⁹ The C function to compose coercions takes a pointer to a *stack-allocated* object as an argument and writes into the object when recursive coercions are composed. Although those stack-allocated objects never escape and tail-call optimization is safe, the C compiler is not powerful enough to see it.

¹⁰ The size of the run-time stack is 8 MB.

static type, say `Bool`, and the other is `Dyn`, the whole `if`-expression is given the static type and the compiler put a cast from `Dyn` on the branch of type `Dyn`. If both A_3 and A_4 are `Dyn`, the recursive calls in the two else-branches will involve casts `bool?p` from `Dyn` to `Bool` because the two then-branches are Boolean constants and the `if`-expressions are given type `Bool`. However, since the return types are declared to be `Dyn`, the whole `if`-expressions are cast back to `Dyn`, inserting injections `bool!`. Thus, every recursive call involves a projection immediately followed by an injection, as shown below, eventually causing stack overflow.

```
(letrec ([even (lambda ([n : Dyn]) : Dyn
  (if (= 0 n<int?p1) #t
      (odd (- n<int?p2 1))<bool?p3><bool!>)]
 [odd (lambda ([n : Dyn]) : Dyn
  (if (= 0 n<int?p4) #f
      (even (- n<int?p5 1))<bool?p6><bool!>))])
  (odd n))
```

5.3 Evaluation

We have conducted some experiments to measure the overhead of the coercion-passing style translation. The benchmark programs we have used are taken from Kuhlenschmidt et al. [24]¹¹; we excluded the sieve program because of the use of recursive types. We also include the even/odd program only for reference, which is relatively small compared to other programs.

We compare the running time of a benchmark program between Base and CrcPS. To take many partially typed configurations for each benchmark program into account, we focus on the so-called *fine-grained* approach, where everywhere a type is required is given either the dynamic type `Dyn` or an appropriate static type.¹² In the fine-grained approach, the number of configurations is 2^n where n is the number of type annotations. When this number is very large, we consider uniformly sampled configurations. We use the sampling algorithm¹³ from [24].

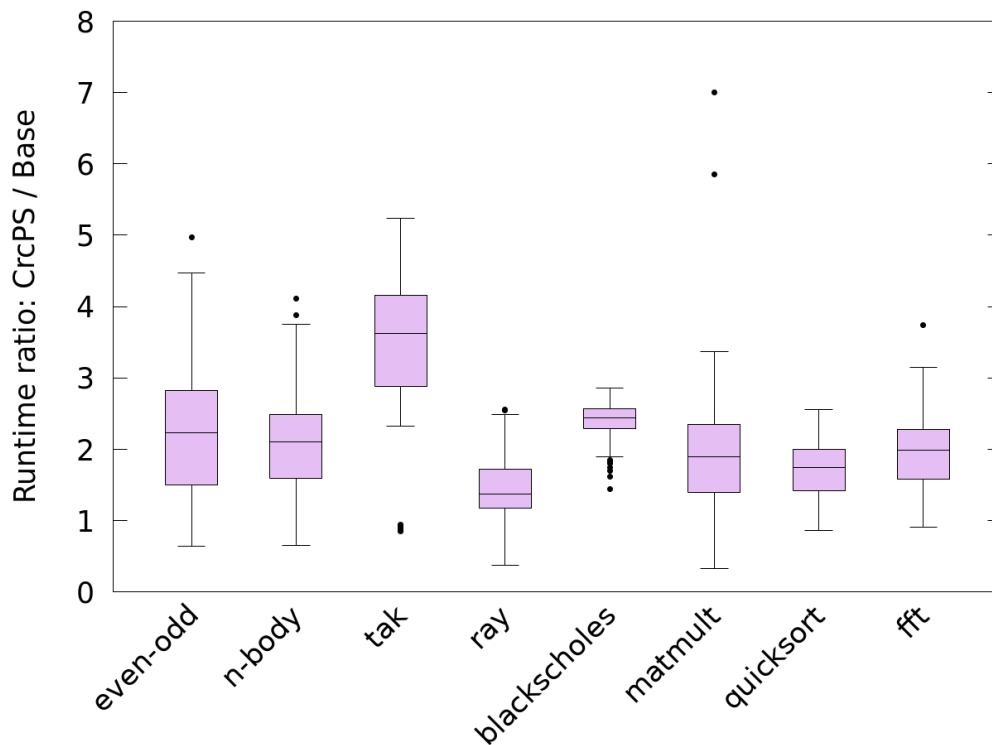
We describe the (sampled) number of partially typed configurations and main language features used for each benchmark program below. (Each benchmark program has one additional type annotation for the return type of the 0-ary main function.) For more detailed description of benchmark programs, we refer readers to Kuhlenschmidt et al. [24].

name	# of configurations	description
even-odd	all $32 = 2^5$	mutually tail-recursive functions
n-body	300 out of 2^{136}	vectors
tak	all $256 = 2^8$	recursive function
ray	300 out of 2^{280}	tuples and iterations
blackscholes	300 out of 2^{128}	vectors and iterations
matmult	300 out of 2^{33}	vectors and iterations
quicksort	300 out of 2^{44}	vectors
fft	300 out of 2^{67}	vectors

¹¹ <https://github.com/Gradual-Typing/benchmarks>

¹² The other approach is called *coarse-grained*, where functions in each module are all statically or all dynamically typed.

¹³ <https://github.com/Gradual-Typing/Dynamizer>



■ **Figure 10** A box plot for the running time ratios of CrcPS to Base across (sampled) partially typed configurations of the benchmark programs. (As is standard, the lower/upper end of a box indicates the first/third quartile, respectively, and the middle line in a box indicates the median. The length of each whisker is below 1.5 times of interquartile range, and outliers are plotted individually.)

Our benchmark method is as follows: For each partially typed configuration of a benchmark program, we measure its running time by taking the average of 5 runs for Base and CrcPS, and compute the ratio of CrcPS to Base. We use a machine with a 8-core 3.6 GHz Intel Core i7-7700 and 16 GB memory, and run the benchmark programs within a Docker container (Docker version 19.03.5) which runs Arch Linux. The generated C code is compiled by clang version 9.0.0 with `-O3` so that tail-call optimization is applied. The size of the run-time stack is set as unlimited.

Figure 10 shows the result in box plots. (Detailed plots for each benchmark are shown in the full version.) It shows that, except for `tak` (and `even-odd`), practical programs in CrcPS run up to three times as slow as Base, for most configurations. It is natural because coercion-passing style translation adds an extra coercion argument to each function. In fact, `tak` and `even-odd`, which have a lot of function calls, have large overhead compared with other programs. In `even-odd`, CrcPS performs many coercion composition operations (and one coercion application) while Base performs many coercion applications (without any coercion composition).¹⁴ Thus, the difference between Base and CrcPS for `even-odd` is partially due to the difference of the cost of coercion application and coercion composition.

¹⁴ An application of a projection coercion to an injected value is always computed by coercion composition in CrcPS, while the implementation of Base is slightly optimized for first-order types.

The benchmark programs other than `tak` and even-odd mainly concern vectors and iterations over them. Vector operations are treated in the translation as primitive operations, which we consider do not have much overhead by the translation. In fact, our translation implementation optimizes the rule (TR-OP) when its continuation is `id`: $\mathcal{K}[\![op(M, N)]\!]id = op(\mathcal{C}[\![M]\!], \mathcal{C}[\![N]\!])$ without an application of an identity coercion.

There are several configurations in which CrcPS is faster than Base but we have not figured out why this is the case.

6 Related Work

6.1 Space-Efficient Coercion/Cast Calculi

As we have already mentioned, it is fairly well known that coercions [18] and casts [43] hamper tail-call optimization and make the space complexity of the execution of a program worse than the execution under an unchecked semantics. We discuss below a few pieces of work [19, 20, 35, 38, 14, 37] addressing the problem.

To the best of our knowledge, Herman et al. [19, 20] were the first to observe the space-efficiency problem of inserted dynamic checks. They developed a variant of Henglein’s coercion calculus with semantics such that a sequence of coercion applications is eagerly composed to reduce the size of coercions. However, they identified two coercions $(c_1; c_2); c_3$ and $c_1; (c_2; c_3)$ (note that $c_1; c_2$ is not a meta-level operator but only a *formal* composition constructor); thus, an algorithm for computing coercion composition was not very clear. They did not take blame tracking [13] into account, either.

Later, Siek et al. [35] extended Herman et al. [19, 20] with a few different blame tracking strategies. The issue of identifying $(c_1; c_2); c_3$ and $c_1; (c_2; c_3)$ remained. According to their terminology, our work, which follows previous work [37], adopts the UD semantics, which allows only $\star \rightarrow \star$ as a tag to functional values, as opposed to the D semantics, which allows any function types to be used as a tag.

Siek and Wadler [38] introduced threesomes to a blame calculus as another solution to the space-efficiency problem. Threesome casts have a third type (called a mediating type) in addition to the source and target types; a threesome cast is considered a downcast from the source type to the mediating, followed by an upcast from the mediating type to the target. Threesome casts allow a simple recursive algorithm to compose two threesome casts but blame tracking is rather complicated.

Garcia [14] gave a translation from coercion calculi to threesome calculi and show that the two solutions to the space-efficiency problem are equivalent in some sense. He introduced supercoercions and a recursive algorithm to compute composition of supercoercions but they were complex, too.

Siek et al. [37] proposed yet another space-efficient coercion calculus λS , in which they succeeded in developing a simple recursive algorithm for coercion composition by restricting coercions to be in certain canonical forms – what they call space-efficient coercions. They also gave a translation from blame calculus λB to λS (via Henglein’s coercion calculus λC) and showed that the translation is fully abstract. As we have discussed already, our λS has introduced syntax that distinguishes an application $U\langle s \rangle$ of a coercion to (uncoerced) values from $U\langle\langle d \rangle\rangle$ for a value wrapped by a delayed coercion. Such distinction, which can be seen in some blame calculi [43], is not just an aesthetic choice but crucial for proving correctness of the translation.

All the above-mentioned calculi adopt a nonstandard reduction rule to compose coercions or casts even before the subject evaluates to a value, together with a nonstandard form of evaluation contexts, and as a result it has not been clear how to implement them

efficiently. Herman et al. [19, 20] sketched a few possible implementation strategies, including coercion passing, but details were not discussed. Siek and Garcia [34] showed an interpreter which performs coercion composition at tail calls. Although not showing correctness of the interpreter, their interpreter would give a hint to direct low-level implementation of space-efficient coercions. Our work addresses the problem of the nonstandard semantics in a different way – by translating a program into coercion-passing style. The difference, however, may not be so large as it may appear at first: in Siek and Garcia [34], a state of the abstract machine includes an evaluation context, which contains the information on a coercion to be applied to a return value and such a coercion roughly corresponds to our continuation coercions. More detailed analysis of the relationship between the two implementation schemes is left for future work.

Kuhlenschmidt et al. [24] built an experimental compiler Grift for gradual typing with structural types. It supports run-time checking with the space-efficient coercions of λS but does not support composition of coercions at tail positions. We have implemented our coercion-passing translation for the Grift compiler.

Greenberg [15] has studied the same space-efficiency problem in the context of manifest contract calculi [23, 16, 17] and proposed a few semantics for composing casts that involve contract checking. Feltey et al. [12] recently implemented Greenberg’s eidetic contracts on top of Typed Racket [41] but, similarly to Kuhlenschmidt et al. [24], composition is limited to a sequence of contracts applied to values.

There is other recent work for making gradual typing efficient [4, 27, 31, 29] but as far as we know, none of them addresses the problem caused by run-time checking applied to tail positions. Additionally, Castagna et al. [5] implemented a virtual machine for space-efficient gradual typing in presence of set-theoretic types, but without blame tracking. They address the problem caused by casts applied to tail positions by an approach similar to the one in the interpreter by Siek and Garcia [34]. They implemented their virtual machine and evaluated their implementation by benchmarks such as the even–odd functions.

6.2 Continuation-Passing Style

Our coercion-passing style translation is inspired by continuation-passing style translation, first formalized by Plotkin [28]. However, coercions represent only a part of the rest of computation and are, in this sense, closer to delimited continuations [7]. Roughly speaking, translating a subexpression with `id` corresponds to the `reset` operation [7] to delimit continuations. Unlike (delimited) continuations, which are usually expressed by first-class functions, coercions have compact representations and compactness can be preserved by composition.

Wallach and Felten [44] proposed security-passing style to implement Java stack inspection [25]. The idea is indeed similar to ours: each function is augmented by an additional argument to pass information on run-time security checking.

In CPS, it is crucial to eliminate administrative redexes to achieve a simulation property [28, 3, 45, 32, 10, 8, 33], which says that a reduction in the source is simulated by a sequence of (one-directional) reductions in the translation. Simulation is usually achieved by applying different translations to an application $M N$, depending on whether M and N are values or not. In addition to such value/nonvalue distinction, our coercion-passing style translation also relies on whether subterms are coercion applications or not.

Continuation-passing style eliminates the difference between call-by-name and call-by-value but our coercion-passing style translation works only under the call-by-value semantics of the target language because coercions have to be eagerly composed. It would be interesting to investigate call-by-name for either the source or the target language, or both.

6.3 First-Class Coercions

The idea of first-class coercions is also found in Cretin and Rémy [6]. Their language F_L is equipped with abstraction over coercions. However, their coercions are not for gradual typing but for parametric polymorphism and subtyping polymorphism.

7 Conclusion

We have developed a new coercion calculus λS_1 with first-class coercions as a target language of coercion-passing style translation from λS , an existing space-efficient coercion calculus. We have proved the translation preserves both typing and semantics. To achieve a simulation property, it is important to reduce administrative coercions, just as in CPS transformations. Our coercion-passing style translation solves the difficulty in implementing the semantics of λS in a faithful manner and, with the help of first-class coercions, makes it possible to implement in a compiler for a call-by-value language. We have modified an existing compiler for a gradually typed language and conducted some experiments. We have confirmed that our implementation successfully overcomes stack overflow caused by coercions at tail positions, which Kuhlenschmidt et al. [24] did not support. Our experiment has shown that for practical programs (without heavy use of function calls), the coercion-passing style translation causes slowdown up to 3 times for most partially typed configurations.

Aside from completing the implementation by adding recursive types, which the original Grift compiler supports, more efficient implementation is an obvious direction of future work. Our coercion-passing style translation introduces several identity coercions and optimizing operations on coercions will be necessary.

From a theoretical point of view, it would be interesting to extend the technique to gradual typing in the presence of parametric polymorphism [1, 2, 21, 47, 42], for which a polymorphic coercion calculus has to be studied first – Luo [26] and Kießling and Luo [22], who study coercive subtyping in polymorphic settings, may be relevant. The present design of λS_1 is geared towards coercion-passing style. For example, in λS_1 , trivial (namely identity) coercions for coercion types $A \rightsquigarrow B$ are allowed; passing coercions to dynamically typed code is prohibited; variables cannot appear as an argument to coercion constructors, like $x \Rightarrow s$. It may be interesting to study more general first-class coercions without such restrictions.

References

- 1 Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 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*, pages 201–214, 2011. doi: 10.1145/1926385.1926409.
- 2 Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. Theorems for free for free: parametricity, with and without types. *PACMPL*, 1(ICFP):39:1–39:28, 2017. doi: 10.1145/3110283.
- 3 Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
- 4 Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy G. Siek, and Sam Tobin-Hochstadt. Sound gradual typing: only mostly dead. *PACMPL*, 1(OOPSLA):54:1–54:24, 2017. doi: 10.1145/3133878.
- 5 Giuseppe Castagna, Guillaume Duboc, Victor Lanvin, and Jeremy G. Siek. A space-efficient call-by-value virtual machine for gradual set-theoretic types. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages, IFL 2019, Singapore, September 25-27, 2019*, 2019.

- 6 Julien Cretin and Didier Rémy. 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*, pages 361–372, 2012. doi:10.1145/2103656.2103699.
- 7 Olivier Danvy and Andrzej Filinski. Abstracting control. In *LISP and Functional Programming*, pages 151–160, 1990. doi:10.1145/91556.91622.
- 8 Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992. doi:10.1017/S0960129500001535.
- 9 Olivier Danvy and Lasse R. Nielsen. Syntactic theories in practice. *Electr. Notes Theor. Comput. Sci.*, 59(4):358–374, 2001. doi:10.1016/S1571-0661(04)00297-X.
- 10 Olivier Danvy and Lasse R. Nielsen. A first-order one-pass CPS transformation. *Theor. Comput. Sci.*, 308(1-3):239–257, 2003. doi:10.1016/S0304-3975(02)00733-8.
- 11 Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full jumps. In *LISP and Functional Programming*, pages 52–62, 1988. doi:10.1145/62678.62684.
- 12 Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. Collapsible contracts: fixing a pathology of gradual typing. *PACMPL*, 2(OOPSLA):133:1–133:27, 2018. doi:10.1145/3276503.
- 13 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002.*, pages 48–59, 2002. doi:10.1145/581478.581484.
- 14 Ronald Garcia. Calculating threesomes, with blame. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 417–428, 2013. doi:10.1145/2500365.2500603.
- 15 Michael Greenberg. Space-efficient manifest contracts. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 181–194, 2015. doi:10.1145/2676726.2676967.
- 16 Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 353–364, 2010. doi:10.1145/1706299.1706341.
- 17 Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. *J. Funct. Program.*, 22(3):225–274, 2012. doi:10.1017/S0956796812000135.
- 18 Fritz Henglein. Dynamic typing: Syntax and proof theory. *Sci. Comput. Program.*, 22(3):197–230, 1994. doi:10.1016/0167-6423(94)00004-2.
- 19 David Herman, Aaron Tomb, and Cormac Flanagan. 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.*, pages 1–18, 2007.
- 20 David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, 23(2):167–189, 2010. doi:10.1007/s10990-011-9066-z.
- 21 Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. On polymorphic gradual typing. *PACMPL*, 1(ICFP):40:1–40:29, 2017. doi:10.1145/3110284.
- 22 Robert Kießling and Zhaohui Luo. Coercions in Hindley–Milner systems. In *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, pages 259–275, 2003. doi:10.1007/978-3-540-24849-1_17.
- 23 Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2):6:1–6:34, 2010. doi:10.1145/1667048.1667051.
- 24 Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. Toward efficient gradual typing for structural types via coercions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 517–532, 2019. doi:10.1145/3314221.3314627.

- 25 Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.
- 26 Zhaohui Luo. Coercions in a polymorphic type system. *Mathematical Structures in Computer Science*, 18(4):729–751, 2008. doi:10.1017/S0960129508006804.
- 27 Fabian Muehlboeck and Ross Tate. Sound gradual typing is nominally alive and well. *PACMPL*, 1(OOPSLA):56:1–56:30, 2017. doi:10.1145/3133880.
- 28 Gordon D. Plotkin. Call-by-name, call-by-value and the λ -calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975. doi:10.1016/0304-3975(75)90017-1.
- 29 Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. 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*, pages 167–180, 2015. doi:10.1145/2676726.2676971.
- 30 John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, December 1998. This paper originally appeared in the Proceedings of the ACM National Conference, volume 2, August 1972, ACM, New York, pages 717–740.
- 31 Gregor Richards, Ellen Arteca, and Alexi Turcotte. The VM already knew that: leveraging compile-time knowledge to optimize gradual typing. *PACMPL*, 1(OOPSLA):55:1–55:27, 2017. doi:10.1145/3133879.
- 32 Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3-4):289–360, 1993.
- 33 Amr Sabry and Philip Wadler. A reflection on call-by-value. *ACM Trans. Program. Lang. Syst.*, 19(6):916–941, 1997. doi:10.1145/267959.269968.
- 34 Jeremy G. Siek and Ronald Garcia. Interpretations of the gradually-typed lambda calculus. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming, Scheme 2012, Copenhagen, Denmark, September 9-15, 2012*, pages 68–80, 2012. doi:10.1145/2661103.2661112.
- 35 Jeremy G. Siek, Ronald Garcia, and Walid Taha. Exploring the design space of higher-order casts. 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*, pages 17–31, 2009. doi:10.1007/978-3-642-00590-9_2.
- 36 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.
- 37 Jeremy G. Siek, Peter Thiemann, and Philip Wadler. Blame and coercion: together again for the first time. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 425–435, 2015. doi:10.1145/2737924.2737968.
- 38 Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 365–376, 2010. doi:10.1145/1706299.1706342.
- 39 Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 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*, pages 456–468, 2016. doi:10.1145/2837614.2837630.
- 40 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *Proc. of Dynamic Languages Symposium*, pages 964–974, 2006. doi:10.1145/1176617.1176755.
- 41 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 395–406, 2008. doi:10.1145/1328438.1328486.

- 42 Matías Toro, Elizabeth Labrada, and Éric Tanter. Gradual parametricity, revisited. *PACMPL*, 3(POPL):17:1–17:30, 2019. doi:10.1145/3290330.
- 43 Philip Wadler and Robert Bruce Findler. 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*, pages 1–16, 2009. doi:10.1007/978-3-642-00590-9_1.
- 44 Dan S. Wallach and Edward W. Felten. Understanding java stack inspection. In *Security and Privacy - 1998 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 3-6, 1998, Proceedings*, pages 52–63, 1998. doi:10.1109/SECPRI.1998.674823.
- 45 Mitchell Wand. Correctness of procedure representations in higher-order assembly language. In *Mathematical Foundations of Programming Semantics, 7th International Conference, Pittsburgh, PA, USA, March 25-28, 1991, Proceedings*, pages 294–311, 1991. doi:10.1007/3-540-55511-0_15.
- 46 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.
- 47 Ningning Xie, Xuan Bi, and Bruno C. d. S. Oliveira. Consistent subtyping for all. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, pages 3–30, 2018. doi:10.1007/978-3-319-89884-1_1.