Space-Efficient Gradual Typing in Coercion-Passing Style

YUYA TSUDA, Kyoto University, Japan
ATSUSHI IGARASHI, Kyoto University, Japan
TOMOYA TABUCHI, Kyoto University, Japan

Herman et al. (2007, 2010) 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 Garcia, et al. (2009) and Siek et al. (2015). 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 run-time checks" around—just like continuation-passing style (CPS), in which "the rest of computation" has been 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. and give a preliminary experimental result.

Additional Key Words and Phrases: coercion calculus, coercion-passing style, dynamic type checking, gradual typing

ACM Reference Format:
Yuya Tsuda, Atsushi Igarashi, and Tomoya Tabuchi. 2020. Space-Efficient Gradual Typing in Coercion-Passing Style. In *Informal Proceedings of the first ACM SIGPLAN Workshop on Gradual Typing (WGT20).* ACM, New York, NY, USA, 27 pages.

1 INTRODUCTION

1.1 Space-Efficiency Problem in Gradual Typing

Gradual typing [Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006] is one of the linguistic approaches to integrating static and dynamic typing. Allowing programmers to mix statically typed program fragments and dynamically typed fragments in a single program, it advocates the "script to program" evolution. Namely, software 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 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 view points [Herman et al. 2007, 2010; Takikawa et al. 2016]. In particular, Takikawa et al. [2016] showed that even the 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 how run-time checks are inserted and how they

WGT20, January 25, 2020, New Orleans
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.

behave. Since then, several pieces of work have investigated the performance issues [Bauman et al. 2017; Feltey et al. 2018; Kuhlenschmidt et al. 2019; Muehlboeck and Tate 2017; Rastogi et al. 2015; Richards et al. 2017].

Earlier work by Herman et al. [2007, 2010] 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, which will be explained shortly, this is a tail-recursive definition of functions to decide a given integer is even or odd, except that the return type of one of the functions is written ★, which is the dynamic type, which can be any tagged value. This definition expresses a situation where a statically typed function and a dynamically typed function calls each other.[1] The gray part represents inserted run-time checks, written by using Henglein's coercion syntax [Henglein 1994]: `true⟨bool!⟩` means that (untagged) Boolean value `true` will be tagged with bool to make a value of the dynamic type and `(even (x - 1))⟨bool?ᵖ⟩` means that the value returned from recursive call `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; 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 the presence of $\langle \mathsf{bool}! \rangle$ and $\langle \mathsf{bool}?^p \rangle$, the recursive calls are not in tail positions any longer. So, according to the original semantics of coercions [Henglein 1994], evaluation of odd 10 as follows:

$$\begin{array}{rcl}
\mathsf{odd}\ 10 & \longmapsto^* & (\mathsf{even}\ 9)\langle \mathsf{bool}?^p \rangle \\
& \longmapsto^* & (\mathsf{odd}\ 8)\langle \mathsf{bool}! \rangle \langle \mathsf{bool}?^p \rangle \\
& \longmapsto^* & (\mathsf{even}\ 7)\langle \mathsf{bool}?^p \rangle \langle \mathsf{bool}! \rangle \langle \mathsf{bool}?^p \rangle \\
& \longmapsto & \cdots \\
& \longmapsto^* & \mathsf{false}\langle \mathsf{bool}! \rangle \langle \mathsf{bool}?^p \rangle \cdots \langle \mathsf{bool}! \rangle \langle \mathsf{bool}?^p \rangle \\
& \longmapsto^* & \mathsf{false}
\end{array}$$

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. [2007, 2010] also presented a solution to this problem. In the evaluation sequence of odd n above, we could immediately "compress" nested coercion applications $M\langle \mathsf{bool}! \rangle \langle \mathsf{bool}?^p \rangle$ before computation of the target term M ends, because $\langle \mathsf{bool}! \rangle \langle \mathsf{bool}?^p \rangle$—tagging immediately followed by untagging—does virtually nothing. 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 [Henglein 1994]. Since then, a few space-efficient coercion/cast calculi have been proposed [Siek et al. 2009, 2015; Siek and Wadler 2010].

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

$$
\begin{array}{ll}
\mathsf{odd}\,4 & \\
\longmapsto^* & (\mathsf{even}\,3)\langle \mathsf{bool}?^p \rangle \\
\longmapsto & (\mathsf{odd}\,(3-1))\langle \mathsf{bool}! \rangle \langle \mathsf{bool}?^p \rangle \\
\longmapsto & (\mathsf{odd}\,(3-1))\langle \mathsf{bool}! \fatsemi \mathsf{bool}?^p \rangle \\
= & (\mathsf{odd}\,(3-1))\langle \mathsf{id}_{\mathsf{bool}} \rangle \\
\longmapsto & (\mathsf{odd}\,2)\langle \mathsf{id}_{\mathsf{bool}} \rangle \\
\longmapsto & (\mathsf{even}\,(2-1))\langle \mathsf{bool}?^p \rangle \langle \mathsf{id}_{\mathsf{bool}} \rangle \\
\longmapsto & (\mathsf{even}\,(2-1))\langle \mathsf{bool}?^p \fatsemi \mathsf{id}_{\mathsf{bool}} \rangle \\
= & (\mathsf{even}\,(2-1))\langle \mathsf{bool}?^p \rangle \\
\longmapsto & (\mathsf{even}\,1)\langle \mathsf{bool}?^p \rangle \\
\longmapsto & \ldots
\end{array}
\qquad
\begin{array}{ll}
\mathsf{oddk}\,(4, \mathsf{id}_{\mathsf{bool}}) & \\
\longmapsto & \mathsf{evenk}\,(4-1, \mathsf{bool}?^p \mathbin{;;} \mathsf{id}_{\mathsf{bool}}) \\
\longmapsto & \mathsf{evenk}\,(4-1, \mathsf{bool}?^p) \\
\longmapsto & \mathsf{evenk}\,(3, \mathsf{bool}?^p) \\
\longmapsto & \mathsf{oddk}\,(3-1, \mathsf{bool}! \mathbin{;;} \mathsf{bool}?^p) \\
\longmapsto & \mathsf{oddk}\,(3-1, \mathsf{id}_{\mathsf{bool}}) \\
\longmapsto & \mathsf{oddk}\,(2, \mathsf{id}_{\mathsf{bool}}) \\
\longmapsto & \mathsf{evenk}\,(2-1, \mathsf{bool}?^p \mathbin{;;} \mathsf{id}_{\mathsf{bool}}) \\
\longmapsto & \mathsf{evenk}\,(2-1, \mathsf{bool}?^p) \\
\longmapsto & \mathsf{evenk}\,(1, \mathsf{bool}?^p) \\
\longmapsto & \ldots
\end{array}
$$

Fig. 1. Reduction from odd 4 in λS (left) and reduction from $\mathsf{odd}\,(4, \mathsf{id}_{\mathsf{bool}})$ in $\lambda\mathrm{S}_1$ (right).

Among them, Siek et al. [2015] have proposed a space-efficient coercion calculus λS. λS is equipped with a composition function that compresses consecutive coercions. The coercion composition is achieved as a simple recursive function thanks to the restriction of coercions to canonical ones. We show evaluation of odd 4 according to the λS semantics in the left of Figure 1.[2] Here, $s \fatsemi t$ is a meta-level operation that composes two coercions s, t (into the canonical form) and yields another coercion that 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 λS 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.

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 and every function call is at a tail position. Actually, the idea of coercion-passing style has already been listed as one of the possible implementation techniques by Herman et al. [2007, 2010] but it has not been 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.

```
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 to return values as in $\texttt{false}\langle\kappa\rangle$. We often call these coercions *continuation coercions*. Coercion applications such as $\texttt{true}\langle\texttt{bool!}\rangle$ and $(\texttt{oddk (x - 1)})\langle\texttt{bool!}\rangle$ at tail positions in the original program are translated to coercion compositions such as $\texttt{true}\langle\texttt{bool! ;; }\kappa\rangle$ and `oddk (x - 1,`

[2] Strictly speaking, bool! and $\mathsf{bool}?^p$ are abbreviations of $\mathsf{id}_{\mathsf{bool}}; \mathsf{bool}!$ and $\mathsf{bool}?^p; \mathsf{id}_{\mathsf{bool}}$, respectively, in λS.

bool! ;; κ), 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 oddk $(4, \mathrm{id}_{\mathsf{bool}})$ (where $\mathrm{id}_{\mathsf{bool}}$ is an identity coercion, which does nothing) proceeds as in the left of Figure 1. Since tagging followed by untagging (with the same tag) actually does nothing, bool! ;; bool?^{p} composes to $\mathrm{id}_{\mathsf{bool}}$ by the (meta-level) coercion composition bool! $\fatsemi$ bool?^{p}.

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

Contributions. Our contributions in this paper are summarized as follows.

- In the context of the space-efficiency problem of gradual typing, we develop a new space-efficient coercion calculus $\lambda\mathrm{S}_1$ with first-class coercions.
- We formalize a coercion-passing translation from (a slight variant of) space-efficient coercion calculus λS [Siek et al. 2015] to the new calculus $\lambda\mathrm{S}_1$.
- We prove correctness of the coercion-passing translation.
- We implement the coercion-passing translation on top of the Grift compiler [Kuhlenschmidt et al. 2019], and conduct a preliminary experiment.

Outline. The rest of this paper is organized as follows. We review the space-efficient coercion calculus λS [Siek et al. 2015] in Section 2. We introduce a new space-efficient coercion calculus with first-class coercions $\lambda\mathrm{S}_1$ in Section 3, formalize a translation into coercion-passing style as a translation from λS to $\lambda\mathrm{S}_1$, and prove correctness of the translation in Section 4. We discuss our implementation of coercion-passing translation on top of the Grift compiler [Kuhlenschmidt et al. 2019] and show a preliminary 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: https://arxiv.org/abs/1908.02414.

2 SPACE-EFFICIENT COERCION CALCULUS

In this section, we review the space-efficient coercion calculus λS [Siek et al. 2015], 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 λS over the original coercion calculus λC [Henglein 1994] 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 are collapsed eagerly—even before they are applied to a value [Herman et al. 2007, 2010; Siek et al. 2009].

Basic forms of coercions are inherited from λC [Henglein 1994], 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 $c_1 \to c_2$, 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

Variables x, y Constants a, b Operators op Blame labels p

$$
\begin{array}{rrcl}
\text{Base types} & \iota & ::= & \mathsf{int} \mid \mathsf{bool} \mid \ldots \\
\text{Types} & A, B, C & ::= & \star \mid \iota \mid A \to B \\
\text{Ground types} & G, H & ::= & \iota \mid \star \to \star \\
\text{Space-efficient coercions} & s, t & ::= & \mathsf{id}_\star \mid G?^p; i \mid i \\
\text{Intermediate coercions} & i & ::= & g; G! \mid g \mid \bot^{GpH} \\
\text{Ground coercions} & g, h & ::= & \mathsf{id}_A \ (\text{if } A \neq \star) \mid s \to t \ (\text{if } s \neq \mathsf{id} \text{ or } t \neq \mathsf{id}) \\
\text{Delayed coercions} & d & ::= & g; G! \mid s \to t \ (\text{if } s \neq \mathsf{id} \text{ or } t \neq \mathsf{id}) \\
\text{Terms} & L, M, N & ::= & V \mid op(M, N) \mid M\,N \mid M\langle s \rangle \mid \mathsf{blame}\ p \\
\text{Values} & V, W & ::= & x \mid U \mid U\langle\!\langle d \rangle\!\rangle \\
\text{Uncoerced values} & U & ::= & a \mid \lambda x.\, M \\
\text{Type environments} & \Gamma & ::= & \emptyset \mid \Gamma, x : A
\end{array}
$$

Fig. 2. Syntax of λS.

following regular expression:

$$(G?^p;)^?(\mathsf{id}_\iota + (s_1 \to s_2))(; G'!)^?$$

(where ι is a base type, s_1 and s_2 stand for space efficient coercions, $(\cdots)^?$ stands for an optional element, and + for alternatives). As already mentioned, an advantage of this form is that (meta-level) sequential composition (denoted by $s_1 \fatsemi s_2$) of two space-efficient coercions results in another space-efficient coercion (if the composition is well typed). For example, the composition

$$((G_1?^p;)^?(\mathsf{id}_\iota + (s_1 \to s_2)); G_2!) \fatsemi (G_3?^{p'}; (\mathsf{id}_\iota + (s_3 \to s_4))(; G_4!)^?)$$

will be

$$((G_1?^p;)^?(\mathsf{id}_\iota + ((s_3 \fatsemi s_1) \to (s_2 \fatsemi s_4)))(; G_4!)^?)$$

if $G_2 = G_3$—that is, tagging with G_2 is immediately followed by inspection whether G_2 is present. Notice that the resulting coercion conforms to the regular expression again. (The other case where $G_2 \neq G_3$ means the projection $G_3?^{p'}$ will fail; we will explain such failures later.)

The operational semantics includes the following reduction rule

$$\mathcal{F}[M\langle s \rangle\langle t \rangle] \longrightarrow \mathcal{F}[M\langle s \fatsemi t \rangle]$$

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.

2.1 Syntax

We show the syntax of λS in Figure 2. The syntax of λS 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 $A \to B$. 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 $\star \to \star$. They are used for type tags put on values of the dynamic type [Wadler and Findler 2009]. Here, the ground type for functions is always

$\star \to \star$, 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, λS 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 λS, ground coercions include identity coercions for any function types—such as $\mathsf{id}_{\iota\to\iota}$—and exclude "virtually identity" coercions such as $\mathsf{id}_\iota \to \mathsf{id}_\iota$. Although these two coercions are extensionally the same, they reduce in slightly different ways: applying $\mathsf{id}_{\iota\to\iota}$ to a function immediately returns the function, whereas applying $\mathsf{id}_\iota \to \mathsf{id}_\iota$ 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 $\bot^{GpH}$ trigger blame (labeled p) if applied to a value. They emerge from coercion composition

$$((G_1?^p;)^?(\mathsf{id}_A + (s_1 \to s_2)); G_2!) \fatsemi (G_3?^{p'}; (\mathsf{id}_A + (s_3 \to s_4))(; G_4!)^?)$$

where $A \neq \star$ and $G_2 \neq G_3$, which means the projection $G_3?^{p'}$ is bound to a failure. The composition results in $(G_1?^p;)^? \bot^{G_1 p' G_3}$, 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. $\mathsf{id}_\star$ is a special coercion that does not conform to the regular expression above. Strictly speaking, an injection, say $\mathsf{int}!$, has to be written $\mathsf{id}_{\mathsf{int}}; \mathsf{int}!$ and a projection, say $\mathsf{int}?^p$, has to be written $\mathsf{int}?^p; \mathsf{id}_{\mathsf{int}}$. 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 blame p. The term $M\langle s\rangle$ coerces the value of M with coercion s at run time. The term 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 $U\langle\!\langle s \to t\rangle\!\rangle$. The check of function coercion is delayed until wrapped functions are applied to a value [Findler and Felleisen 2002; Henglein 1994; Siek and Taha 2006].

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 [2009]). 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, not 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. So, 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.

Well-formed coercions $\boxed{c : A \rightsquigarrow B}$

$$\frac{}{G! : G \rightsquigarrow \star}\ \text{CT-Inj} \qquad \frac{}{G?^p : \star \rightsquigarrow G}\ \text{CT-Proj} \qquad \frac{A \neq \star \quad A \sim G \quad G \neq H}{\bot^{GpH} : A \rightsquigarrow B}\ \text{CT-Fail}$$

$$\frac{}{\mathsf{id}_A : A \rightsquigarrow A}\ \text{CT-Id} \qquad \frac{c_1 : A' \rightsquigarrow A \quad c_2 : B \rightsquigarrow B'}{c_1 \to c_2 : A \to B \rightsquigarrow A' \to B'}\ \text{CT-Fun} \qquad \frac{c_1 : A \rightsquigarrow B \quad c_2 : B \rightsquigarrow C}{(c_1; c_2) : A \rightsquigarrow C}\ \text{CT-Seq}$$

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

$$\frac{}{\Gamma \vdash a : ty(a)}\ \text{T-Const} \qquad \frac{ty(op) = \iota_1 \to \iota_2 \to \iota \quad \Gamma \vdash M : \iota_1 \quad \Gamma \vdash N : \iota_2}{\Gamma \vdash op(M, N) : \iota}\ \text{T-Op}$$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ \text{T-Var} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.\, M : A \to B}\ \text{T-Abs} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B}\ \text{T-App}$$

$$\frac{\Gamma \vdash M : A \quad s : A \rightsquigarrow B}{\Gamma \vdash M\langle s \rangle : B}\ \text{T-Crc} \qquad \frac{\emptyset \vdash U : A \quad d : A \rightsquigarrow B}{\emptyset \vdash U\langle\!\langle d \rangle\!\rangle : B}\ \text{T-CrcV} \qquad \frac{}{\emptyset \vdash \mathsf{blame}\, p : A}\ \text{T-Blame}$$

Fig. 3. Typing rules of λS.

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* $\Gamma \vdash_S M : A$. 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 [Siek and Taha 2006], 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 $c_1 \to c_2$. 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 $c_1 \to c_2$ converts type $A \to B$ to type $A' \to B'$. 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 $\bot^{GpH}$. Here, the source 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:

$$\begin{array}{lll} (\mathsf{id}_{\mathsf{int}}; \mathsf{int}!) \to (\mathsf{int}?^p; \mathsf{id}_{\mathsf{int}}) & : \star \to \star & \rightsquigarrow \mathsf{int} \to \mathsf{int} \\ \bot^{\star \to \star p \mathsf{int}} & : \mathsf{int} \to \mathsf{bool} & \rightsquigarrow \mathsf{int} \end{array}$$

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 there exists a unique* G *such that* $A \sim G$ *and* $G \sim B$.

The judgment $\Gamma \vdash_S M : A$ means that the λS-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 $\longmapsto_S$) introduced later.

The rules (T-Const), (T-Op), (T-Var), (T-Abs), and (T-App) are standard. Here, $ty(a)$ maps constant a to a base type ι, and $ty(op)$ maps binary operator op to a (first-order) function type $\iota_1 \to \iota_2 \to \iota$. 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 $M\langle s \rangle$ has 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 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 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.

2.3 Operational Semantics

2.3.1 Coercion Composition. The coercion composition $s \fatsemi t$ 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 decides which rule to apply.

The rules (CC-IdDynL) and (CC-ProjL) are applied when the first one is not an intermediate coercion. The rules (CC-InjId), (CC-Collapse), (CC-Conflict), and (CC-FailL) are applied when the first one is a (nonground) intermediate coercion, in which case another intermediate coercion is yielded. Here, (CC-Collapse) and (CC-Conflict) perform tag checks if an injection and a projection meet. If type tags do not match, a failure coercion arises.

Failure coercions are necessary for eager coercion composition not to change the behavior of ordinary coercion calculus λC. The term $M\langle G! \rangle\langle H?^p \rangle$ (if $G \neq H$) in λC evaluates to blame p—*after M evaluates to a value.* By contrast, the two coercions $G!$ and $H?^p$ in the term $M\langle \mathsf{id}_G; G! \rangle\langle H?^p; \mathsf{id}_H \rangle$ are eagerly composed in λS. Raising blame p immediately would not match the semantics of λC if M evaluates to another blame. $\bot^{GpH}$ must raise blame p only after M evaluates to a value.

The rules (CC-FailR) and (CC-InjR) are applied when 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 $\fatsemi$ is propagated to the inside.

The rules (CC-IdL), (CC-IdR), and (CC-Fun) are applied when two ground coercions are composed to another ground coercion. They are straightforward except that $\mathsf{id}_A \to \mathsf{id}_B$ has to be normalized to $\mathsf{id}_{A \to B}$ (CC-Fun).

We present a few examples of coercion composition below:

$$
\begin{aligned}
(\mathsf{id}_{\mathsf{bool}}; \mathsf{bool}!) \fatsemi (\mathsf{bool}?^p; \mathsf{id}_{\mathsf{bool}}) &= \mathsf{id}_{\mathsf{bool}} \fatsemi \mathsf{id}_{\mathsf{bool}} = \mathsf{id}_{\mathsf{bool}} \\
(\mathsf{id}_{\star \to \star}; (\star \to \star)!) \fatsemi (\mathsf{int}?^p; \mathsf{id}_{\mathsf{int}}) &= \bot^{\star \to \star p \mathsf{int}} \\
((\iota?^p; \mathsf{id}_\iota) \to (\mathsf{id}_{\iota'}; \iota'!)) \fatsemi ((\mathsf{id}_\iota; \iota!) \to \mathsf{id}_\star) &= ((\mathsf{id}_\iota; \iota!) \fatsemi (\iota?^p; \mathsf{id}_\iota)) \to ((\mathsf{id}_{\iota'}; \iota'!) \fatsemi \mathsf{id}_\star) \\
&= \mathsf{id}_\iota \to (\mathsf{id}_{\iota'}; \iota'!)
\end{aligned}
$$

These examples involve situations where an injection meets a projection by (CC-Collapse) or (CC-Conflict). The third example is by (CC-Fun).

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

Evaluation contexts $\mathcal{E} ::= \mathcal{F} \mid \mathcal{F}[\square\langle s\rangle]$

$\mathcal{F} ::= \square \mid \mathcal{E}[op(\square, M)] \mid \mathcal{E}[op(V, \square)] \mid \mathcal{E}[\square\, M] \mid \mathcal{E}[V\, \square]$

Coercion composition $\boxed{s \,⨟\, t = s'}$

$$\mathsf{id}_\star ⨟ t = t \qquad \text{CC-IdDynL}$$
$$(G?^p; i) ⨟ t = G?^p; (i ⨟ t) \qquad \text{CC-ProjL}$$

$$(g; G!) ⨟ \mathsf{id}_\star = g; G! \qquad \text{CC-InjId}$$
$$(g; G!) ⨟ (G?^p; i) = g ⨟ i \qquad \text{CC-Collapse}$$
$$(g; G!) ⨟ (H?^p; i) = \bot^{GpH} \quad \text{if } G \neq H \qquad \text{CC-Conflict}$$
$$\bot^{GpH} ⨟ s = \bot^{GpH} \qquad \text{CC-FailL}$$

$$g ⨟ \bot^{GpH} = \bot^{GpH} \qquad \text{CC-FailR}$$
$$g ⨟ (h; H!) = (g ⨟ h); H! \qquad \text{CC-InjR}$$

$$\mathsf{id}_A ⨟ g = g \quad \text{if } A \neq \star \qquad \text{CC-IdL}$$
$$g ⨟ \mathsf{id}_A = g \quad \text{if } A \neq \star \text{ and } g \neq \mathsf{id}_A \qquad \text{CC-IdR}$$

$$(s \to t) ⨟ (s' \to t') = \begin{cases} \mathsf{id}_{A\to B} & \text{if } s' ⨟ s = \mathsf{id}_A \text{ and } t ⨟ t' = \mathsf{id}_B \\ (s' ⨟ s) \to (t ⨟ t') & \text{otherwise} \end{cases} \qquad \text{CC-Fun}$$

Reduction $\boxed{M \xrightarrow{\mathsf{e}}_{\mathsf{S}} N}$ $\boxed{M \xrightarrow{\mathsf{c}}_{\mathsf{S}} N}$

$$op(a, b) \xrightarrow{\mathsf{e}} \delta\,(op, a, b) \qquad \text{R-Op}$$
$$(\lambda x.\, M)\, V \xrightarrow{\mathsf{e}} M[x := V] \qquad \text{R-Beta}$$
$$(U\langle\!\langle s \to t\rangle\!\rangle)\, V \xrightarrow{\mathsf{e}} (U\,(V\langle s\rangle))\langle t\rangle \qquad \text{R-Wrap}$$
$$U\langle \mathsf{id}_A\rangle \xrightarrow{\mathsf{c}} U \qquad \text{R-Id}$$
$$U\langle \bot^{GpH}\rangle \xrightarrow{\mathsf{c}} \mathsf{blame}\, p \qquad \text{R-Fail}$$
$$U\langle d\rangle \xrightarrow{\mathsf{c}} U\langle\!\langle d\rangle\!\rangle \qquad \text{R-Crc}$$
$$M\langle s\rangle\langle t\rangle \xrightarrow{\mathsf{c}} M\langle s ⨟ t\rangle \qquad \text{R-MergeC}$$
$$U\langle\!\langle d\rangle\!\rangle\langle t\rangle \xrightarrow{\mathsf{c}} U\langle d ⨟ t\rangle \qquad \text{R-MergeV}$$

Evaluation $\boxed{M \xmapsto{\mathsf{e}}_{\mathsf{S}_1} N}$ $\boxed{M \xmapsto{\mathsf{c}}_{\mathsf{S}_1} N}$

$$\frac{M \xrightarrow{\mathsf{e}} N}{\mathcal{E}[M] \xmapsto{\mathsf{e}} \mathcal{E}[N]}\ \text{E-CtxE} \qquad \frac{M \xrightarrow{\mathsf{c}} N}{\mathcal{F}[M] \xmapsto{\mathsf{c}} \mathcal{F}[N]}\ \text{E-CtxC} \qquad \frac{\mathcal{E} \neq \square}{\mathcal{E}[\mathsf{blame}\, p] \xmapsto{\mathsf{e}} \mathsf{blame}\, p}\ \text{E-Abort}$$

Fig. 4. Reduction/evaluation rules of λS.

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* $(s \fatsemi t) : A \rightsquigarrow C$.

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

We show the reduction rules and the evaluation rules of λS in Figure 4. 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 $\longrightarrow_{\mathsf{S}}$ for $\xrightarrow{\mathsf{e}}_{\mathsf{S}} \cup \xrightarrow{\mathsf{c}}_{\mathsf{S}}$, and $\longmapsto_{\mathsf{S}}$ for $\overset{\mathsf{e}}{\longmapsto}_{\mathsf{S}} \cup \overset{\mathsf{c}}{\longmapsto}_{\mathsf{S}}$. We sometimes call $\overset{\mathsf{e}}{\longmapsto}_{\mathsf{S}}$ and $\overset{\mathsf{c}}{\longmapsto}_{\mathsf{S}}$ e-reduction and c-reduction, 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 $ty(op) = \iota_1 \to \iota_2 \to \iota$ 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, which we omit.

The rule (R-Wrap) applies to applications of wrapped function $U\langle\!\langle s \to t \rangle\!\rangle$ 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 $\bot^{GpH}$ to uncoerced value U, which reduces to 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-MergeC) and (R-MergeV) 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-MergeV), we can assume that there is at most one coercion in a value. The outermost nested coercion applications are merged by (R-MergeC).

Now, we explain *evaluation contexts*, ranged over by $\mathcal{E}$, shown in the top of Figure 4. Following Siek et al. [2015], we define them in the so-called "inside-out" style [Danvy and Nielsen 2001; Felleisen et al. 1988]. Evaluation contexts represent that function calls in λS 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.[3] 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:

$$\mathcal{F}_1 = \Box \qquad \mathcal{E}_1 = \mathcal{F}_1[\Box\langle s \rangle] = \Box\langle s \rangle$$
$$\mathcal{F}_2 = \mathcal{E}_1[V\,\Box] = (V\,\Box)\langle s \rangle \qquad \mathcal{E}_2 = \mathcal{F}_2[\Box\langle t \rangle] = (V\,(\Box\langle t \rangle))\langle s \rangle$$
$$\mathcal{F}_3 = \mathcal{E}_2[\Box\,M] = (V\,((\Box\,M)\langle t \rangle))\langle s \rangle$$

We then come back to evaluation rules: The rules (E-CtxE) and (E-CtxC) enable us to evaluate the subterm in an evaluation context. Here, (E-CtxC) requires that computation of coercion applications is only performed under contexts $\mathcal{F}$—otherwise, the innermost frame may be a coercion application,

[3] $\mathcal{F}[\Box\langle s \rangle]$ (instead of $\mathcal{F}[\Box\langle f \rangle]$) in the definition of $\mathcal{E}$ fixes a problem in Siek et al. [2015] that an identity coercion applied to a nonvalue gets stuck (personal communication).

in which case (R-MergeC) has to be applied first. For example, $U\langle d\rangle\langle t\rangle$ reduces to $U\langle d \fatsemi t\rangle$ rather than $U\langle\!\langle d\rangle\!\rangle\langle t\rangle$. The rule (E-Abort) halts evaluation of a program if it raises blame.

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

$$
\begin{array}{rcll}
(U\langle \mathsf{int}! \to \mathsf{int}?^p\rangle)\,3 & \longmapsto & ((U\langle\!\langle \mathsf{int}! \to \mathsf{int}?^p\rangle\!\rangle)\,3)\langle \mathsf{int}!\rangle & \text{by (R-Crc)} \\
& \longmapsto & (U\,(3\langle \mathsf{int}!\rangle))\langle \mathsf{int}?^p\rangle\langle \mathsf{int}!\rangle & \text{by (R-Wrap)} \\
& \longmapsto & (U\,(3\langle \mathsf{int}!\rangle))\langle \mathsf{int}?^p; \mathsf{id}; \mathsf{int}!\rangle & \text{by (R-MergeC)} \\
& \longmapsto & (U\,(3\langle\!\langle \mathsf{int}!\rangle\!\rangle))\langle \mathsf{int}?^p; \mathsf{id}; \mathsf{int}!\rangle & \text{by (R-Crc)} \\
& \longmapsto & (3\langle\!\langle \mathsf{int}!\rangle\!\rangle\langle \mathsf{int}?^p\rangle + 2)\langle \mathsf{int}!\rangle\langle \mathsf{int}?^p; \mathsf{id}; \mathsf{int}!\rangle & \text{by (R-Beta)} \\
& \longmapsto & (3\langle\!\langle \mathsf{int}!\rangle\!\rangle\langle \mathsf{int}?^p\rangle + 2)\langle \mathsf{int}!\rangle & \text{by (R-MergeC)} \\
& \longmapsto & (3\langle \mathsf{id}\rangle + 2)\langle \mathsf{int}!\rangle & \text{by (R-MergeV)} \\
& \longmapsto & (3 + 2)\langle \mathsf{int}!\rangle & \text{by (R-Id)} \\
& \longmapsto & 5\langle \mathsf{int}!\rangle & \text{by (R-Op)} \\
& \longmapsto & 5\langle\!\langle \mathsf{int}!\rangle\!\rangle & \text{by (R-Crc)}
\end{array}
$$

2.4 Properties

We state a few important properties of λS, including determinacy of the evaluation relation and type safety via preservation and progress [Wright and Felleisen 1994]. We write $\longmapsto_{\mathrm{S}}^*$ for the reflexive and transitive closure of $\longmapsto_{\mathrm{S}}$, and $\longmapsto_{\mathrm{S}}^+$ for the transitive closure of $\longmapsto_{\mathrm{S}}$. We say that λS-term M *diverges*, denoted by $M \Uparrow_{\mathrm{S}}$, if there exists an infinite evaluation sequence from M.

Proofs of the stated properties are in the full version.

Lemma 4 (Determinacy). *If $M \longmapsto_{\mathrm{S}} N$ and $M \longmapsto_{\mathrm{S}} N'$, then $N = N'$.*

Theorem 5 (Progress). *If $\emptyset \vdash_{\mathrm{S}} M : A$, then one of the following holds.*

(1) *$M \longmapsto_{\mathrm{S}} M'$ for some M'.*
(2) *$M = V$ for some V.*
(3) *$M = \mathsf{blame}\,p$ for some p.*

Theorem 6 (Preservation). *If $\emptyset \vdash_{\mathrm{S}} M : A$ and $M \longmapsto_{\mathrm{S}} N$, then $\emptyset \vdash_{\mathrm{S}} N : A$.*

Corollary 7 (Type Safety). *If $\emptyset \vdash_{\mathrm{S}} M : A$, then one of the following holds.*

(1) *$M \longmapsto_{\mathrm{S}}^* V$ and $\emptyset \vdash_{\mathrm{S}} V : A$ for some V.*
(2) *$M \longmapsto_{\mathrm{S}}^* \mathsf{blame}\,p$ for some p.*
(3) *$M \Uparrow_{\mathrm{S}}$.*

3 SPACE-EFFICIENT FIRST-CLASS COERCION CALCULUS

In this section, we introduce $\lambda\mathrm{S}_1$, a new space-efficient coercion calculus with first-class coercions; $\lambda\mathrm{S}_1$ serves as the target calculus of the translation into coercion-passing style. The design of $\lambda\mathrm{S}_1$ 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., $\mathsf{id}_{\iota \rightsquigarrow \iota}$).

Since coercions are first-class in $\lambda\mathrm{S}_1$, 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 λS with the infix (object-level) operator $M \mathbin{;;} N$ to compute the composition of two coercions: if M and N evaluate to coercions s and t, respectively, then $M \mathbin{;;} N$ reduces to their composition $s \fatsemi t$, which is another space-efficient coercion. The type of (first-class) coercions from A to B is written $A \rightsquigarrow B$.[4]

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

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	$::=$	$\mathsf{id}_\star \mid G?^p; i \mid i$
Intermediate coercions	i	$::=$	$g; G! \mid g \mid \bot^{GpH}$
Ground coercions	g, h	$::=$	id_A (if $A \neq \star$) $\mid s \Rightarrow t$ (if $s \neq \mathsf{id}$ or $t \neq \mathsf{id}$)
Delayed coercions	d	$::=$	$g; G! \mid s \Rightarrow t$ (if $s \neq \mathsf{id}$ or $t \neq \mathsf{id}$)
Terms	L, M, N	$::=$	$V \mid op(M, N) \mid L\,(M, N) \mid \mathsf{let}\ x = M\ \mathsf{in}\ N$
			$\mid M\ ;;\ N \mid M\langle N \rangle \mid \mathsf{blame}\ p$
Values	V, W, K	$::=$	$x \mid U \mid U\langle\!\langle d \rangle\!\rangle$
Uncoerced values	U	$::=$	$a \mid \lambda(x, \kappa).\, M \mid s$
Type environments	Γ	$::=$	$\emptyset \mid \Gamma, x : A$

Fig. 5. Syntax of $\lambda\mathsf{S}_1$.

In $\lambda\mathsf{S}_1$, 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 $\lambda\mathsf{S}$ corresponds to $\lambda(x, \kappa).\, 1\langle\kappa\rangle$ in $\lambda\mathsf{S}_1$—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 $\lambda\mathsf{S}$ corresponds to $f\,(3, s)$ in $\lambda\mathsf{S}_1$; $(f\ 3)$ (without a coercion application) corresponds to $f\,(3, \mathsf{id})$.

The type of a function abstraction in $\lambda\mathsf{S}_1$ 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 $\lambda\mathsf{S}$. Intuitively, $A \Rightarrow B$ means $\forall X. A \times (B \rightsquigarrow X) \to X$ 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.

Following the change to function types, function coercions in $\lambda\mathsf{S}_1$ 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 $\lambda\mathsf{S}$; 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 $\lambda\mathsf{S}_1$ in Figure 5. We reuse the same metavariables from $\lambda\mathsf{S}$. We also use κ for variables, and K for values.

Well-formed coercions (replace) $\boxed{c : A \rightsquigarrow B}$

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

Term typing (excerpt) $\boxed{\Gamma \vdash_{S_1} M : A}$

$$\frac{s : A \rightsquigarrow B}{\Gamma \vdash s : A \rightsquigarrow B}\ \text{T-Crcn} \qquad \frac{\Gamma \vdash M : A \rightsquigarrow B \qquad \Gamma \vdash N : B \rightsquigarrow C}{\Gamma \vdash M \mathbin{;;} N : A \rightsquigarrow C}\ \text{T-Cmp}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : A \rightsquigarrow B}{\Gamma \vdash M\langle N \rangle : B}\ \text{T-Crc} \qquad \frac{\emptyset \vdash U : A \qquad \emptyset \vdash d : A \rightsquigarrow B}{\emptyset \vdash U\langle\!\langle d \rangle\!\rangle : B}\ \text{T-CrcV}$$

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

$$\frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash N : B}{\Gamma \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N : B}\ \text{T-Let} \qquad \frac{\Gamma \vdash L : A \Rightarrow B \qquad \Gamma \vdash M : A \qquad \Gamma \vdash N : B \rightsquigarrow C}{\Gamma \vdash L\,(M, N) : C}\ \text{T-App}$$

Fig. 6. Typing rules of λS_1.

We replace $A \to B$ 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 $\to$ 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 $M \mathbin{;;} N$. 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 $\mathsf{let}\ x = M\ \mathsf{in}\ N$ 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 λS.

3.2 Type System

The type system of λS_1 is straightforward adaption of that of λS. Main rules are shown in Figure 6.

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

The judgment $\Gamma \vdash_{S_1} M : A$ means that term M of λS_1 has type A under type environment Γ. The rules (T-Const), (T-Op), (T-Var), and (T-Blame) are the same as λS, 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 $\forall X.\, A \times (B \rightsquigarrow X) \to X$ 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 should 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 λS, but adjusted to first-class coercions.

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

Coercion composition (replace) $\boxed{s \fatsemi t = s'}$

$$(s \Rightarrow t) \fatsemi (s' \Rightarrow t') = \begin{cases} \mathsf{id}_{A \Rightarrow B} & \text{if } s' \fatsemi s = \mathsf{id}_A \text{ and } t \fatsemi t' = \mathsf{id}_B \\ (s' \fatsemi s) \Rightarrow (t \fatsemi t') & \text{otherwise} \end{cases} \qquad \text{CC-Fun}$$

Reduction $\boxed{M \xrightarrow{\mathsf{e}}_{\mathsf{S}_1} N}$ $\boxed{M \xrightarrow{\mathsf{c}}_{\mathsf{S}_1} N}$

$$op(a, b) \xrightarrow{\mathsf{e}} \delta\,(op, a, b) \qquad \text{R-Op}$$

$$(\lambda(x, \kappa).\, M)\,(V, W) \xrightarrow{\mathsf{e}} M[x := V, \kappa := W] \qquad \text{R-Beta}$$

$$(U\langle\!\langle s \Rightarrow t \rangle\!\rangle)\,(V, W) \xrightarrow{\mathsf{e}} \mathsf{let}\ \kappa = t\ ;;\ W\ \mathsf{in}\ U\,(V\langle s\rangle, \kappa) \qquad \text{R-Wrap}$$

$$\mathsf{let}\ x = V\ \mathsf{in}\ M \xrightarrow{\mathsf{c}} M[x := V] \qquad \text{R-Let}$$

$$s\ ;;\ t \xrightarrow{\mathsf{c}} s \fatsemi t \qquad \text{R-Cmp}$$

$$U\langle \mathsf{id}_A \rangle \xrightarrow{\mathsf{c}} U \qquad \text{R-Id}$$

$$U\langle \bot^{GpH} \rangle \xrightarrow{\mathsf{c}} \mathsf{blame}\ p \qquad \text{R-Fail}$$

$$U\langle d \rangle \xrightarrow{\mathsf{c}} U\langle\!\langle d \rangle\!\rangle \qquad \text{R-Crc}$$

$$U\langle\!\langle d \rangle\!\rangle\langle t \rangle \xrightarrow{\mathsf{c}} U\langle d\ ;;\ t \rangle \qquad \text{R-MergeV}$$

Evaluation $\boxed{M \overset{\mathsf{e}}{\longmapsto}_{\mathsf{S}_1} N}$ $\boxed{M \overset{\mathsf{c}}{\longmapsto}_{\mathsf{S}_1} N}$

$$\frac{M \xrightarrow{\mathcal{X}} N \qquad \mathcal{X} \in \{\mathsf{e}, \mathsf{c}\}}{\mathcal{E}[M] \overset{\mathcal{X}}{\longmapsto} \mathcal{E}[N]}\ \text{E-Ctx} \qquad \frac{\mathcal{E} \neq \Box}{\mathcal{E}[\mathsf{blame}\ p] \overset{\mathsf{e}}{\longmapsto} \mathsf{blame}\ p}\ \text{E-Abort}$$

Fig. 7. Reduction/evaluation rules of $\lambda\mathsf{S}_1$.

3.3 Operational Semantics

The composition function $s \fatsemi t$ is mostly the same as that of λS. We only replace (CC-Fun) as shown in Figure 7. As in λS, function coercions are contravariant in their argument coercions and covariant in their return-value coercions.

Similarly to λS, we give a small-step operational semantics to $\lambda\mathsf{S}_1$ consisting of two relations on closed terms: the reduction relation $M \longrightarrow_{\mathsf{S}_1} N$ and the evaluation relation $M \longmapsto_{\mathsf{S}_1} N$. We show the reduction/evaluation rules of $\lambda\mathsf{S}_1$ in Figure 7. As in λS, they are labeled either e or c. We write $\longrightarrow_{\mathsf{S}_1}$ for $\xrightarrow{\mathsf{e}}_{\mathsf{S}_1} \cup \xrightarrow{\mathsf{c}}_{\mathsf{S}_1}$, and $\longmapsto_{\mathsf{S}_1}$ for $\overset{\mathsf{e}}{\longmapsto}_{\mathsf{S}_1} \cup \overset{\mathsf{c}}{\longmapsto}_{\mathsf{S}_1}$.

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 λS. 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 λS and $\lambda\mathsf{S}_1$ to match.

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 $s \mathbin{;;} t$, which is evaluated by meta-level coercion composition function $s \fatsemi t$. The rules (R-Id), (R-Fail), (R-Crc), and (R-MergeV) are the same as λS.

Now, we explain evaluation contexts, ranged over by $\mathcal{E}$, shown in the top of Figure 7. In contrast to λS, evaluation contexts are standard in $\lambda\mathrm{S}_1$. The definition of evaluation contexts $\mathcal{E}$ represents that function calls in $\lambda\mathrm{S}_1$ 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 λS. (However, evaluation contexts in (E-Ctx) are more straightforward in $\lambda\mathrm{S}_1$.)

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

Example 8. Let U be $\lambda(x, \kappa).\, \mathsf{let}\, \kappa' = \mathsf{int}\,! \mathbin{;;} \kappa\, \mathsf{in}\, (x\langle \mathsf{int}?^p \rangle + 2)\langle \kappa' \rangle$, which corresponds to the λS-term $\lambda x.\, (x\langle \mathsf{int}?^p \rangle + 2)\langle \mathsf{int}\,! \rangle$ 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 $(U\langle \mathsf{int}\,! \Rightarrow \mathsf{int}?^p \rangle)\,(3, \mathsf{int}\,!)$ evaluates to $5\langle\!\langle \mathsf{int}\,! \rangle\!\rangle$ as follows:

$$
\begin{array}{lll}
& (U\langle \mathsf{int}\,! \Rightarrow \mathsf{int}?^p \rangle)\,(3, \mathsf{int}\,!) & \\
\longmapsto & (U\langle\!\langle \mathsf{int}\,! \Rightarrow \mathsf{int}?^p \rangle\!\rangle)\,(3, \mathsf{int}\,!) & \text{by (R-Crc)} \\
\longmapsto & \mathsf{let}\, \kappa'' = \mathsf{int}?^p \mathbin{;;} \mathsf{int}\,!\, \mathsf{in}\, U\,(3\langle \mathsf{int}\,! \rangle, \kappa'') & \text{by (R-Wrap)} \\
\longmapsto & \mathsf{let}\, \kappa'' = \mathsf{int}?^p; \mathsf{id}; \mathsf{int}\,!\, \mathsf{in}\, U\,(3\langle \mathsf{int}\,! \rangle, \kappa'') & \text{by (R-Cmp)} \\
\longmapsto & U\,(3\langle \mathsf{int}\,! \rangle, (\mathsf{int}?^p; \mathsf{id}; \mathsf{int}\,!)) & \text{by (R-Let)} \\
\longmapsto & U\,(3\langle\!\langle \mathsf{int}\,! \rangle\!\rangle, (\mathsf{int}?^p; \mathsf{id}; \mathsf{int}\,!)) & \text{by (R-Crc)} \\
\longmapsto & \mathsf{let}\, \kappa' = \mathsf{int}\,! \mathbin{;;} (\mathsf{int}?^p; \mathsf{id}; \mathsf{int}\,!)\, \mathsf{in}\, (3\langle\!\langle \mathsf{int}\,! \rangle\!\rangle\langle \mathsf{int}?^p \rangle + 2)\langle \kappa' \rangle & \text{by (R-Beta)} \\
\longmapsto & \mathsf{let}\, \kappa' = \mathsf{int}\,!\, \mathsf{in}\, (3\langle\!\langle \mathsf{int}\,! \rangle\!\rangle\langle \mathsf{int}?^p \rangle + 2)\langle \kappa' \rangle & \text{by (R-Cmp)} \\
\longmapsto & (3\langle\!\langle \mathsf{int}\,! \rangle\!\rangle\langle \mathsf{int}?^p \rangle + 2)\langle \mathsf{int}\,! \rangle & \text{by (R-Let)} \\
\longmapsto & (3\langle \mathsf{id} \rangle + 2)\langle \mathsf{int}\,! \rangle & \text{by (R-MergeV)} \\
\longmapsto & (3 + 2)\langle \mathsf{int}\,! \rangle & \text{by (R-Id)} \\
\longmapsto & 5\langle \mathsf{int}\,! \rangle & \text{by (R-Op)} \\
\longmapsto & 5\langle\!\langle \mathsf{int}\,! \rangle\!\rangle & \text{by (R-Crc)}
\end{array}
$$

It is easy to see that the steps by (R-MergeC) in Example 3 are simulated by (R-Cmp) followed by (R-Let).

3.4 Properties

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

Lemma 9 (Determinacy). *If $M \longmapsto_{\mathrm{S}_1} N$ and $M \longmapsto_{\mathrm{S}_1} N'$, then $N = N'$.*

Theorem 10 (Progress). *If $\emptyset \vdash_{\mathrm{S}_1} M : A$, then one of the following holds.*

(1) *$M \longmapsto_{\mathrm{S}_1} M'$ for some M'.*
(2) *$M = V$ for some V.*
(3) *$M = \mathsf{blame}\, p$ for some p.*

Theorem 11 (Preservation). *If $\emptyset \vdash_{\mathrm{S}_1} M : A$ and $M \longmapsto_{\mathrm{S}_1} N$, then $\emptyset \vdash_{\mathrm{S}_1} N : A$.*

Corollary 12 (Type Safety). *If $\emptyset \vdash_{\mathrm{S}_1} M : A$, then one of the following holds.*

(1) *$M \longmapsto^*_{\mathrm{S}_1} V$ and $\emptyset \vdash_{\mathrm{S}_1} V : A$ for some V.*
(2) *$M \longmapsto^*_{\mathrm{S}_1} \mathsf{blame}\, p$ for some p.*
(3) *$M \Uparrow_{\mathrm{S}_1}$.*

Type translation $\boxed{\Psi(A) = A'}$

$$\Psi(\star) = \star \qquad \Psi(\iota) = \iota \qquad \Psi(A \to B) = \Psi(A) \Rightarrow \Psi(B)$$

Coercion translation $\boxed{\Psi(s) = s'}$

$$\begin{aligned}
\Psi(\mathrm{id}_A) &= \mathrm{id}_{\Psi(A)} \\
\Psi(g; G!) &= \Psi(g); \Psi(G)! \\
\Psi(G?^p; i) &= \Psi(G)?^p; \Psi(i) \\
\Psi(s \to t) &= \Psi(s) \Rightarrow \Psi(t) \\
\Psi(\bot^{GpH}) &= \bot^{GpH}
\end{aligned}$$

Value translation $\boxed{\Psi(V) = V'}$

$$\begin{aligned}
\Psi(x) &= x \\
\Psi(a) &= a \\
\Psi(\lambda x.\, M) &= \lambda(x, \kappa).\, (\mathscr{K}[\![M]\!]\kappa) \\
\Psi(U\langle\!\langle d \rangle\!\rangle) &= \Psi(U)\langle\!\langle \Psi(d) \rangle\!\rangle
\end{aligned}$$

Term translation $\boxed{\mathscr{C}[\![M]\!] = M'}$ $\boxed{\mathscr{K}[\![M]\!]K = M'}$

$$\begin{aligned}
\mathscr{C}[\![V]\!] &= \Psi(V) \\
\mathscr{C}[\![M^A]\!] &= \mathscr{K}[\![M]\!]\mathrm{id}_{\Psi(A)} && \text{if } M \text{ is not a value}
\end{aligned}$$

$$\begin{aligned}
\mathscr{K}[\![V]\!]K &= \Psi(V)\langle K \rangle && && \textsc{Tr-Val} \\
\mathscr{K}[\![op(M, N)]\!]K &= op(\mathscr{C}[\![M]\!], \mathscr{C}[\![N]\!])\langle K \rangle && && \textsc{Tr-Op} \\
\mathscr{K}[\![M\, N]\!]K &= (\mathscr{C}[\![M]\!])\,(\mathscr{C}[\![N]\!], K) && && \textsc{Tr-App} \\
\mathscr{K}[\![M\langle s \rangle]\!]\mathrm{id} &= \mathscr{K}[\![M]\!]\Psi(s) && && \textsc{Tr-CrcId} \\
\mathscr{K}[\![M\langle s \rangle]\!]K &= \mathrm{let}\ \kappa = \Psi(s)\ ;;\ K\ \mathrm{in}\ (\mathscr{K}[\![M]\!]\kappa) && \text{if } K \neq \mathrm{id} && \textsc{Tr-Crc} \\
\mathscr{K}[\![\mathrm{blame}\, p]\!]K &= \mathrm{blame}\, p && && \textsc{Tr-Blame}
\end{aligned}$$

Fig. 8. Translation into coercion-passing style (from λS to $\lambda\mathrm{S}_1$).

4 TRANSLATION INTO COERCION-PASSING STYLE

In this section, we formalize a translation into coercion-passing style as a translation from λS to $\lambda\mathrm{S}_1$ 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 [Plotkin 1975].

4.1 Definition of Translation

We give the translation into coercion-passing style by the translation rules presented in Figure 8. In order to distinguish metavariables of λS and $\lambda\mathrm{S}_1$, we often use blue for the source calculus λS. 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 λS.

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

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

Value translation $\Psi(V)$ is rather straightforward: every function $\lambda x.\, M$ is translated to a $\lambda\mathrm{S}_1$-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 $\mathscr{K}[\![M]\!]\kappa$.

We now describe the translation for terms. We write $\mathscr{K}[\![M]\!]K$ for the translation of λS-term M with continuation coercion K. We first explain the basic transformation scheme given by the

following simpler rules:

$$\mathscr{K}'[\![V]\!]K = \Psi(V)\langle K\rangle \qquad \textsc{Tr}'\text{-Val}$$
$$\mathscr{K}'[\![op(M^{\iota_1}, N^{\iota_2})]\!]K = op(\mathscr{K}'[\![M]\!]\mathrm{id}_{\iota_1}, \mathscr{K}'[\![N]\!]\mathrm{id}_{\iota_2})\langle K\rangle \qquad \textsc{Tr}'\text{-Op}$$
$$\mathscr{K}'[\![M^{A\to B}\, N^{A}]\!]K = (\mathscr{K}'[\![M]\!]\mathrm{id}_{\Psi(A\to B)})\,(\mathscr{K}'[\![N]\!]\mathrm{id}_{\Psi(A)}, K) \qquad \textsc{Tr}'\text{-App}$$
$$\mathscr{K}'[\![M\langle s\rangle]\!]K = \mathsf{let}\ \kappa = \Psi(s)\ ;;\ K\ \mathsf{in}\ (\mathscr{K}'[\![M]\!]\kappa) \qquad \textsc{Tr}'\text{-Crc}$$
$$\mathscr{K}'[\![\mathsf{blame}\ p]\!]K = \mathsf{blame}\ p \qquad \textsc{Tr}'\text{-Blame}$$

(We put a prime on $\mathscr{K}$ to avoid confusion.)

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 $op(M, N)$. We translate the arguments M and N with identity continuation coercions by $\mathscr{K}'[\![M]\!]\mathrm{id}$ and $\mathscr{K}'[\![N]\!]\mathrm{id}$ 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 λS, not by the translation. In other words, the correctness of the translation relies on the fact that λS 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 $\mathscr{K}'[\![M]\!]\mathrm{id}$.

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 $\mathscr{K}'[\![M]\!]\kappa$. In general, it is not necessarily the case that $\mathscr{K}'[\![M]\!]K$ evaluates K first, so if we set $\mathscr{K}'[\![M\langle s\rangle]\!]K = (\mathscr{K}'[\![M]\!](\Psi(s)\ ;;\ K))$, then the order of computation would change by the translation and correctness of translation would be harder.

Lastly, the rule (Tr′-Blame) defines the translation of blame p with continuation K as blame p.

The translation $\mathscr{K}'$ seems acceptable but, just as naive CPS transformation leaves administrative redexes, it leaves many applications of id, which we call *administrative coercions*. We expect M and $\mathscr{K}'[\![M]\!]K$ behave "similarly" but administrative redexes make it hard to show such semantic correspondence. So, we will optimize the translation so that administrative coercions are eliminated, similarly to CPS transformations that eliminates administrative redexes [Appel 1992; Danvy and Filinski 1992; Danvy and Nielsen 2003; Plotkin 1975; Sabry and Felleisen 1993; Sabry and Wadler 1997; Wand 1991].

The bottom of Figure 8 shows the optimized translation rules. The idea to eliminate administrative coercions is close to the colon translation by Plotkin [1975]: we avoid translating values with administrative coercions. So, we introduce an auxiliary translation function $\mathscr{C}[\![M]\!]$, which returns $\Psi(V)$—without a coercion application—if M is a value V and returns $\mathscr{K}[\![M]\!]\mathrm{id}$ otherwise. Translation rules for primitive operations and function applications are adapted so that they use $\mathscr{C}[\![M]\!]$ to translate subexpressions. We also split the rule for coercion applications according to whether K is

id or not. The rule (Tr-CrcId), which applies if K is id, optimizes the trivial composition $\Psi(s) \fatsemi \mathsf{id}$ away.

We present a few examples of the translation below:

$$\Psi(5) = 5$$
$$\Psi(\lambda x.\, x + 1) = \lambda(x, \kappa).\, (x + 1)\langle\kappa\rangle$$
$$\mathscr{K}[\![(\lambda x.\, x)\, 5]\!]\mathsf{int}\,! = (\lambda(x, \kappa).\, x\langle\kappa\rangle)\,(5, \mathsf{int}\,!)$$
$$\mathscr{K}[\![((\lambda x.\, x)\, 5)\langle\mathsf{int}\,!\rangle]\!]\mathsf{int}?^p = \mathsf{let}\,\kappa = \mathsf{int}\,!\,;;\,\mathsf{int}?^p\,\mathsf{in}\,(\lambda(x, \kappa).\, x\langle\kappa\rangle)\,(5, \kappa)$$

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

Example 13. Let U be a λS-term $\lambda x.\, (x\langle\mathsf{int}?^p\rangle + 2)\langle\mathsf{int}\,!\rangle$.

$$\begin{array}{rcl}
\Psi(U) & = & \lambda(x, \kappa).\, (\mathscr{K}[\![(x\langle\mathsf{int}?^p\rangle + 2)\langle\mathsf{int}\,!\rangle]\!]\kappa) \\
& = & \lambda(x, \kappa).\, \mathsf{let}\,\kappa' = \mathsf{int}\,!\,;;\,\kappa\,\mathsf{in}\,(\mathscr{K}[\![(x\langle\mathsf{int}?^p\rangle + 2)]\!]\kappa') \\
& = & \lambda(x, \kappa).\, \mathsf{let}\,\kappa' = \mathsf{int}\,!\,;;\,\kappa\,\mathsf{in}\,(\mathscr{C}[\![x\langle\mathsf{int}?^p\rangle]\!] + \mathscr{C}[\![2]\!])\langle\kappa'\rangle \\
& = & \lambda(x, \kappa).\, \mathsf{let}\,\kappa' = \mathsf{int}\,!\,;;\,\kappa\,\mathsf{in}\,((\mathscr{K}[\![x\langle\mathsf{int}?^p\rangle]\!]\mathsf{id}) + 2)\langle\kappa'\rangle \\
& = & \lambda(x, \kappa).\, \mathsf{let}\,\kappa' = \mathsf{int}\,!\,;;\,\kappa\,\mathsf{in}\,((\mathscr{K}[\![x]\!]\mathsf{int}?^p) + 2)\langle\kappa'\rangle \\
& = & \lambda(x, \kappa).\, \mathsf{let}\,\kappa' = \mathsf{int}\,!\,;;\,\kappa\,\mathsf{in}\,(x\langle\mathsf{int}?^p\rangle + 2)\langle\kappa'\rangle
\end{array}$$

$$\begin{array}{rcl}
\mathscr{K}[\![((U\langle\mathsf{int}\,! \to \mathsf{int}?^p\rangle)\, 3)]\!]\mathsf{id} & = & (\mathscr{K}[\![(U\langle\mathsf{int}\,! \to \mathsf{int}?^p\rangle)]\!]\mathsf{id})\,(\mathscr{K}[\![3]\!]\mathsf{id}, \mathsf{id}) \\
& = & (\mathscr{K}[\![U]\!](\mathsf{int}\,! \to \mathsf{int}?^p))\,(3, \mathsf{id}) \\
& = & (\Psi(U)\langle\mathsf{int}\,! \Rightarrow \mathsf{int}?^p\rangle)\,(3, \mathsf{id})
\end{array}$$

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 the following:

$$(x : A) \in \Gamma \text{ if and only if } (x : \Psi(A)) \in \Psi(\Gamma).$$

Theorem 14 (Translation Preserves Typing). *If* $\Gamma \vdash_{\mathsf{S}} M : A$, *then* $\Psi(\Gamma) \vdash_{\mathsf{S}_1} (\mathscr{K}[\![M]\!]\mathsf{id}_{\Psi(A)}) : \Psi(A)$.

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 Semantcs). *If* $\emptyset \vdash_{\mathsf{S}} M : \iota$, *then*

(1) $M \longmapsto_{\mathsf{S}}^* a$ *iff* $\mathscr{K}[\![M]\!]\mathsf{id}_\iota \longmapsto_{\mathsf{S}_1}^* a$;
(2) $M \longmapsto_{\mathsf{S}}^* \mathsf{blame}\, p$ *iff* $\mathscr{K}[\![M]\!]\mathsf{id}_\iota \longmapsto_{\mathsf{S}_1}^* \mathsf{blame}\, p$; *and*
(3) $M \Uparrow_{\mathsf{S}}$ *iff* $\mathscr{K}[\![M]\!]\mathsf{id}_\iota \Uparrow_{\mathsf{S}_1}$.

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 $\emptyset \vdash_{\mathsf{S}} M : \iota$ and $\mathscr{K}[\![M]\!]\mathsf{id}_\iota \Uparrow_{\mathsf{S}_1}$, 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 $\longmapsto_{\mathsf{S}_1}$ is deterministic).

Theorem 16 (Translation Soundness). *Suppose* $\Gamma \vdash_{\mathsf{S}} M : A$.

(1) *If* $M \longmapsto_{\mathsf{S}}^* V$, *then* $\mathscr{K}[\![M]\!]\mathsf{id} \longmapsto_{\mathsf{S}_1}^* \Psi(V)$.
(2) *If* $M \longmapsto_{\mathsf{S}}^* \mathsf{blame}\, p$, *then* $\mathscr{K}[\![M]\!]\mathsf{id} \longmapsto_{\mathsf{S}_1}^* \mathsf{blame}\, p$.

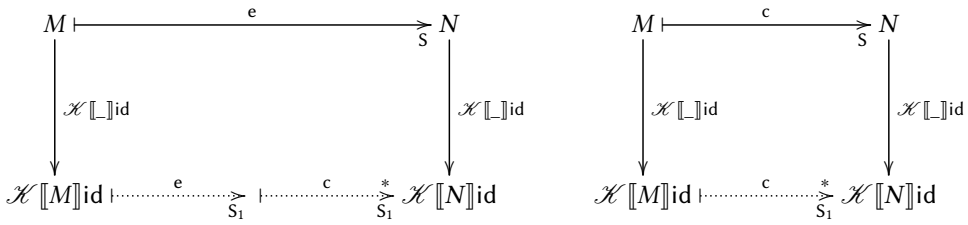
(3) *If* $M \Uparrow_{\mathrm{S}}$, *then* $\mathscr{K}[\![M]\!]\mathsf{id} \Uparrow_{\mathrm{S}_1}$.

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

Lemma 17 (Simulation).

(1) *If* $M \overset{\mathrm{e}}{\longmapsto}_{\mathrm{S}} N$, *then* $\mathscr{K}[\![M]\!]\mathsf{id} \overset{\mathrm{e}}{\longmapsto}_{\mathrm{S}_1} \overset{\mathrm{c}}{\longmapsto}^*_{\mathrm{S}_1} \mathscr{K}[\![N]\!]\mathsf{id}$.

(2) *If* $M \overset{\mathrm{c}}{\longmapsto}_{\mathrm{S}} N$, *then* $\mathscr{K}[\![M]\!]\mathsf{id} \overset{\mathrm{c}}{\longmapsto}^*_{\mathrm{S}_1} \mathscr{K}[\![N]\!]\mathsf{id}$.



Whereas a single-step e-reduction in $\lambda\mathrm{S}$ is translated to one or more steps in $\lambda\mathrm{S}_1$ starting from an e-reduction step, a single-step c-reduction in $\lambda\mathrm{S}$ can be translated to *zero* steps in $\lambda\mathrm{S}_1$. An example is $0\langle\mathsf{int}!\rangle\langle\mathsf{id}\rangle \overset{\mathrm{c}}{\longmapsto}_{\mathrm{S}} 0\langle\mathsf{int}!\rangle$; the two terms both translate to $0\langle\mathsf{int}!\rangle$ by removing id. Still an infinite reduction sequence in $\lambda\mathrm{S}$ is preserved by translation because c-reduction is terminating (proved in the full version) and there is an infinite number of e-reductions.

As is the case for simulation proofs for CPS translation [Appel 1992; Danvy and Filinski 1992; Danvy and Nielsen 2003; Plotkin 1975; Sabry and Felleisen 1993; Sabry and Wadler 1997; Wand 1991], this simulation property[5] 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 $\mathscr{C}[\![M]\!]$. For example, $(\lambda x.\, x)\, 5 \overset{\mathrm{e}}{\longmapsto} 5$ holds in $\lambda\mathrm{S}$ but the translation $\mathscr{K}'[\![(\lambda x.\, x)\, 5]\!]K$ without removing administrative redexes would yield $(\lambda(x, \kappa).\, x\langle\kappa\rangle)\langle\mathsf{id}\rangle\, (5\langle\mathsf{id}\rangle, K)$, which performs c-reduction before calling the function. More formally, we prove the following lemma, which means the redex in the source is also the redex in the target.

Lemma 18.

(1) *For any* $\mathcal{F}$, *there exist* $\mathcal{E}'$ *such that for any* M, $\mathscr{K}[\![\mathcal{F}[M]]\!]\mathsf{id} = \mathcal{E}'[\mathscr{C}[\![M]\!]]$.

(2) *For any* $\mathcal{F}$ *and* s, *there exists* $\mathcal{E}'$ *such that for any* M, $\mathscr{K}[\![\mathcal{F}[M\langle s\rangle]]\!]\mathsf{id} = \mathcal{E}'[\mathscr{K}[\![M]\!]\Psi(s)]$.

To prove this lemma, the rule (Tr-CrcID) also plays an important role: for example, if we removed (Tr-CrcID) and used (Tr-Crc) for all coercion applications, $\mathscr{K}[\![(1+1)\langle\mathsf{int}!\rangle]\!]\mathsf{id}$ would translate to $\mathsf{let}\,\kappa = \mathsf{int}!\;;;\,\mathsf{id}\,\mathsf{in}\,(1+1)\langle\kappa\rangle$, which performs c-reduction before adding 1 and 1, which is the first thing the original term $(1+1)\langle\mathsf{int}!\rangle$ will do.

Second, optimizing too many (identity) coercions can be break simulation. Consider $M \overset{\mathrm{def}}{=} (((\lambda x.\, M_1)\langle\!\langle \mathsf{id}_\iota \to \iota! \rangle\!\rangle)\, a)\langle \iota?^p \rangle$ and $N \overset{\mathrm{def}}{=} ((\lambda x.\, M_1)\, (a\langle\mathsf{id}_\iota\rangle))\langle\iota!\rangle\langle\iota?^p\rangle$, for which $M \longmapsto_{\mathrm{S}} N$ holds by (R-Wrap). Then,

$$\begin{array}{lll} \mathscr{K}[\![M]\!]\mathsf{id} & = & ((\mathscr{K}[\![\lambda(x,\kappa).\, M_1]\!]\kappa)\langle\!\langle \mathsf{id}_\iota \Rightarrow \iota! \rangle\!\rangle)\, (a, \iota?^p) \\ & \longmapsto_{\mathrm{S}_1} & \mathsf{let}\,\kappa' = \iota!\;;;\,\iota?^p\,\mathsf{in}\,(\mathscr{K}[\![\lambda(x,\kappa).\, M_1]\!]\kappa)\,(a\langle\mathsf{id}_\iota\rangle, \kappa') \\ & = & \mathscr{K}[\![N]\!]\mathsf{id}. \end{array}$$

[5] If we had been interested only in 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 to show that divergence is preserved by translation.

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

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

(notice that $\langle \mathsf{id}_\iota \rangle$ on a is removed). Although $\mathscr{K}''[\![M]\!]\mathsf{id}$ and $\mathscr{K}''[\![N]\!]\mathsf{id}$ reduced to the same term, we did not quite have $\mathscr{K}''[\![M]\!]\mathsf{id} \longmapsto^+ \mathscr{K}''[\![N]\!]\mathsf{id}$ as we had expected.

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

Lemma 19 (Substitution). *If* $\kappa \notin FV(M) \cup FV(V)$, *then* $(\mathscr{K}[\![M]\!]\kappa)[x := \Psi(V), \kappa := K] \overset{c}{\longmapsto}{}^*_{S_1} \mathscr{K}[\![M[x := V]]\!]K$.

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, 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 this property.

5 IMPLEMENTATION AND PRELIMINARY EVALUATION

We have implemented the coercion-passing translation described in Section 4 and the semantics of λS_1 for Grift [Kuhlenschmidt et al. 2019], 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, and mutable references. The compiler supports different run-time check schemes, those based on type-based casts [Siek and Taha 2006] and space-efficient coercions [Siek et al. 2015]. Note that, although space-efficient coercions are supported, only nested coercions on *values* are composed; in other words, (R-MergeC) is 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 [Siek et al. 2015]. 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:

$$\mathscr{K}[\![\mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2]\!]K = \mathsf{if}\ (\mathscr{K}[\![M]\!]\mathsf{id})\ \mathsf{then}\ (\mathscr{K}[\![N_1]\!]K)\ \mathsf{else}\ (\mathscr{K}[\![N_2]\!]K).$$

Intermediate languages used in Grift already supports first-class coercions, making it straightforward to implement our approach. We modify another compiler phase that generates operations on coercions such as $M\;;;\,N$ and (R-Wrap). The current implementation, which generates C code and uses `clang`[6] 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[7] hampers tail-call optimizations, so our implementation does not deal with recursive types. We leave their implementation for future work.

We have conducted a preliminary experiment to measure the overhead of the coercion-passing translation. The benchmark programs we have used are the tail-recursive even–odd functions in direct style:

[6] https://clang.llvm.org/

[7] 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.

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

and its CPS-transformed version:

```
(letrec ([evenk (lambda ([n : A1] [k : (A3 -> A3)]) : A3
                (if (= n 0) (k #t)
                    (oddk (- n 1) k)))]
         [oddk (lambda ([n : A2] [k : (A4 -> A4)]) : A4
                (if (= n 0) (k #f)
                    (evenk (- n 1) k)))])
  (oddk n (lambda ([v : Bool]) : Bool v)))
```

We run the former with the original and modified compilers and the latter with the original for all combinations of A_1 and A_2, which are either `Int` or `Dyn`, and A_3 and A_3, which are either `Bool` or `Dyn`. They result in 48 different configurations. We call the direct-style program compiled by the original compiler Base, the direct-style program compiled by the modified compiler CrcPS, the CPS program compiled by the original compiler CPS.

First, we have confirmed that, as n increases, 12 of 16 configurations of Base cause stack overflow.[8] In the four configurations that survived, both A_3 and A_4 are set to `Bool`. CrcPS never causes stack overflow for any configuration. For CPS, 8 of 16 configurations cause stack overflow. In all the crashed configurations A_3 and A_4 are different.

To our surprise, Base causes stack overflow even when $A_3 = A_4 =$ `Dyn` and CPS crashes for some configurations. 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 static type, say `Bool`, and the other is `Dyn`, the whole `if`-expression is given the static type and the compiler inserts a cast from `Dyn` to the branch of type `Dyn`. In the direct-style program where both A_3 and A_4 are `Dyn`, the two then-branches are always Boolean constants and the recursive calls in the two else-branches involve casts from `Dyn` to `Bool`, hence the insertion of projections bool?^{p}. 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⟩) #f
              (odd (- n⟨int?^p2⟩ 1))⟨bool?^p3⟩)⟨bool!⟩)]
         [odd (lambda ([n : Dyn) : Dyn
          (if (= 0 n⟨int?^p4⟩) #t
              (even (- n⟨int?^p5⟩ 1))⟨bool?^p6⟩)⟨bool!⟩))])
  (odd n))
```

Similarly, a projection and an injection are inserted implicitly to the CPS program with $A_3 \neq A_4$, causing stack overflow.

[8] The size of the run-time stack is 32MB.

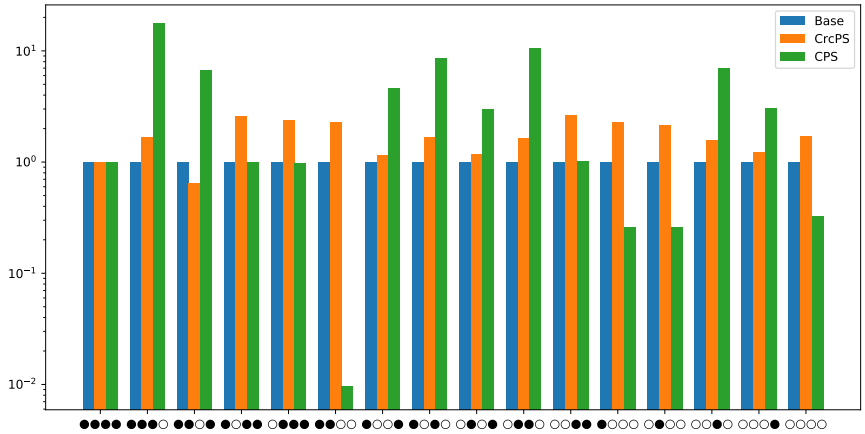


Fig. 9. A comparison of the normalized average running time with respect to Base. (Larger bars are slower.) For each group, the four black/white circles represent whether each A_i is static (represented by a black circle) or dynamic (by a white circle).

Now, we show running time of different configurations, measured by taking the average of 1,000 runs. We run these programs on a machine with 2.6 GHz Intel Core i5 and 8 GB memory. The generated C code is compiled by clang version 10.0.1 with `-O2` so that tail-call optimization is performed. The size of the run-time stack is set to 32MB, as we have already mentioned.

Figure 9 shows the overhead (normalized average running time) for each configuration. The overhead is computed by dividing the average running time of CrcPS or CPS by that of Base. (We set the argument $n = 10^6$, which is below the threshold to cause a stack overflow.)

Overall, the normalized average running time of CrcPS is between 0.64–2.61. The reason why CrcPS is faster than Base in some configurations is not very clear. The performance of CPS (ranging between 0.26–17.7 except for one extreme case ●●○○, where $A_1 = A_2 = \texttt{Int}$ and $A_3 = A_4 = \texttt{Dyn}$) seems less stable than CrcPS. It turns out that, in the extreme configuration, Base (and CrcPS) suffer from implicit coercions for if-expressions, as discussed above, but CPS does not have such coercions.

Figure 10 shows performance lattices [Takikawa et al. 2016] for each program. Each performance lattice shows the average running time, normalized with respect to the dynamically typed version ○○○○. Base and CrcPS are similar: the performance gets better as more static types are used. Even partially typed programs show better performance compared to the dynamically typed version ○○○○ at the bottom. This is probably because even and odd in direct style do not use higher-order casts, to which much slowdown is often attributed. In fact, the performance of CPS, which uses higher-order functions and casts, depends highly on whether A_3 and A_4 are the same.

6 RELATED WORK

The idea of first-class coercions is also found in Cretin and Rémy [2012]. Their language F_ι is equipped with abstraction over coercions. However, their coercions are not for gradual typing but for parametric polymorphism and subtyping polymorphism.

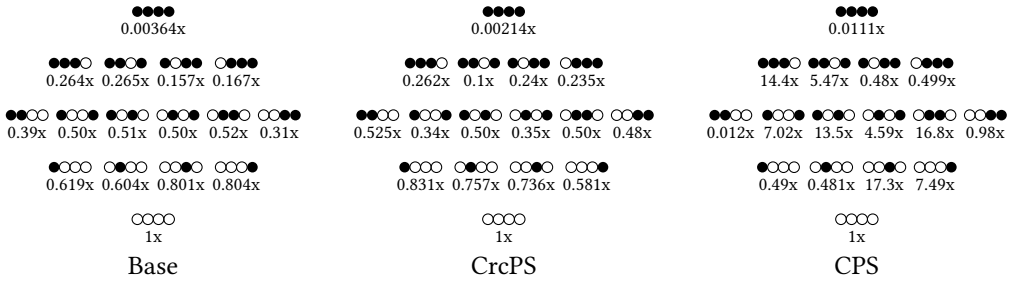


Fig. 10. Performance lattices. The four black/white circles represent whether each A_i is static (represented by a black circle) or dynamic (by a white circle).

6.1 Space-Efficient Coercion/Cast Calculi

As we have already mentioned, it is fairly well known that coercions [Henglein 1994] and casts [Wadler and Findler 2009] hamper the 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 [Garcia 2013; Herman et al. 2007, 2010; Siek et al. 2009, 2015; Siek and Wadler 2010] addressing the problem.

To the best of our knowledge, Herman et al. [2007, 2010] 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, their coercion composition operator is defined to be associative, equating $(c_1; c_2); c_3$ and $c_1; (c_2; c_3)$; thus, an algorithm for computing coercion composition was not very clear. They did not take blame tracking [Findler and Felleisen 2002] into account, either.

Later, Siek et al. [2009] extended Herman et al. [2007, 2010] 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 [Siek et al. 2015], adopts the UD semantics, which allows only $\star \to \star$ 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 [2010] introduced threesomes to a blame calculus. 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 to the mediating type, followed by an upcast from the mediating to the target. Threesome casts allow a simple recursive algorithm to compose two threesome casts but blame tracking is rather complicated.

Garcia [2013] gave a translation from the threesome calculus to a coercion calculus and the two solutions are equivalent. They introduced supercoercions and a recursive algorithm to compute composition of supercoercions but they were complex, too.

Siek et al. [2015] 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 some blame calculi [Wadler and Findler 2009], 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. [2007, 2010] sketched a few possible implementation strategies, including coercion passing but details were not discussed. Siek and Garcia [2012] 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 [2012], 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. [2019] 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 [2015] has studied the same space-efficiency problem in the context of manifest contract calculi [Greenberg et al. 2010, 2012; Knowles and Flanagan 2010] and proposed a few semantics for composing casts that involve contract checking. Feltey et al. [2018] have recently implemented Greenberg's eidetic contracts on top of Typed Racket [Tobin-Hochstadt and Felleisen 2008] but, similarly to Kuhlenschmidt et al. [2019], composition is limited on a sequence of contracts applied to values.

There are other recent work for making gradual typing efficient [Bauman et al. 2017; Muehlboeck and Tate 2017; Rastogi et al. 2015; Richards et al. 2017] 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. [2019] 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 [2012]. They implemented their virtual machine and evaluated their implementation by benchmarks such as the even/odd program.

6.2 Continuation-Passing Style

Obviously, our coercion-passing style translation is inspired by continuation-passing style translation, first formalized by Plotkin [1975]. However, coercions represent only a part of the rest of computation and are, in this sense, closer to delimited continuations [Danvy and Filinski 1990]. Roughly speaking, translating a subexpression with id corresponds to the reset operation [Danvy and Filinski 1990] 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 [1998] proposed security-passing style to implement Java stack inspection [Lindholm and Yellin 1999]. 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 [Appel 1992; Danvy and Filinski 1992; Danvy and Nielsen 2003; Plotkin 1975; Sabry and Felleisen 1993; Sabry and Wadler 1997; Wand 1991], 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 whether M and N are values or

not. In addition to such value/nonvalue distinction, our coercion-passing translation also relies on whether a given continuation coercion is id or not when a coercion application is translated.

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 and/or the target language.

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 a preliminary experiment. We have confirmed that our implementation successfully overcomes memory overflow caused by coercions at tail positions, which Kuhlenschmidt et al. [2019] did not support. Although the overhead is not small, causing slowdown of 2.61 times at maximum, the overall performance is stable under different type annotations and tends to be better than manually CPS translated programs.

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 a lot of 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 [Ahmed et al. 2011, 2017; Igarashi et al. 2017; Toro et al. 2019; Xie et al. 2018], for which a polymorphic coercion calculus has to be studied first—Kießling and Luo [2003]; Luo [2008], 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.

ACKNOWLEDGMENTS

This work was partially supported by JSPS KAKENHI Grant Number JP17H01723. We would like to thank John Toman for proofreading.

REFERENCES

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

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

Andrew W. Appel. 1992. *Compiling with Continuations.* Cambridge University Press.

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

Giuseppe Castagna, Guillaume Duboc, Victor Lanvin, and Jeremy G. Siek. 2019. 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.*

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

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *LISP and Functional Programming*. 151–160. https://doi.org/10.1145/91556.91622

Olivier Danvy and Andrzej Filinski. 1992. Representing Control: A Study of the CPS Transformation. *Mathematical Structures in Computer Science* 2, 4 (1992), 361–391. https://doi.org/10.1017/S0960129500001535

Olivier Danvy and Lasse R. Nielsen. 2001. Syntactic Theories in Practice. *Electr. Notes Theor. Comput. Sci.* 59, 4 (2001), 358–374. https://doi.org/10.1016/S1571-0661(04)00297-X

Olivier Danvy and Lasse R. Nielsen. 2003. A first-order one-pass CPS transformation. *Theor. Comput. Sci.* 308, 1-3 (2003), 239–257. https://doi.org/10.1016/S0304-3975(02)00733-8

Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. 1988. Abstract Continuations: A Mathematical Semantics for Handling Full Jumps. In *LISP and Functional Programming*. 52–62. https://doi.org/10.1145/62678.62684

Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. 2018. Collapsible contracts: fixing a pathology of gradual typing. *PACMPL* 2, OOPSLA (2018), 133:1–133:27. https://doi.org/10.1145/3276503

Robert Bruce Findler and Matthias Felleisen. 2002. 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*. 48–59. https://doi.org/10.1145/581478.581484

Ronald Garcia. 2013. Calculating threesomes, with blame. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. 417–428. https://doi.org/10.1145/2500365.2500603

Michael Greenberg. 2015. 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*. 181–194. https://doi.org/10.1145/2676726.2676967

Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. 2010. 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*. 353–364. https://doi.org/10.1145/1706299.1706341

Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. 2012. Contracts made manifest. *J. Funct. Program.* 22, 3 (2012), 225–274. https://doi.org/10.1017/S0956796812000135

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

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

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

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

Robert Kießling and Zhaohui Luo. 2003. 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*. 259–275. https://doi.org/10.1007/978-3-540-24849-1_17

Kenneth Knowles and Cormac Flanagan. 2010. Hybrid type checking. *ACM Trans. Program. Lang. Syst.* 32, 2 (2010), 6:1–6:34. https://doi.org/10.1145/1667048.1667051

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

Tim Lindholm and Frank Yellin. 1999. *The Java Virtual Machine Specification* (2nd ed.). Addison-Wesley.

Zhaohui Luo. 2008. Coercions in a polymorphic type system. *Mathematical Structures in Computer Science* 18, 4 (2008), 729–751. https://doi.org/10.1017/S0960129508006804

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

Gordon D. Plotkin. 1975. Call-by-Name, Call-by-Value and the λ-Calculus. *Theor. Comput. Sci.* 1, 2 (1975), 125–159. https://doi.org/10.1016/0304-3975(75)90017-1

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

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

Amr Sabry and Matthias Felleisen. 1993. Reasoning about Programs in Continuation-Passing Style. *Lisp and Symbolic Computation* 6, 3-4 (1993), 289–360.

Amr Sabry and Philip Wadler. 1997. A Reflection on Call-by-Value. *ACM Trans. Program. Lang. Syst.* 19, 6 (1997), 916–941. https://doi.org/10.1145/267959.269968

Jeremy G. Siek and Ronald Garcia. 2012. 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*. 68–80. https://doi.org/10.1145/2661103.2661112

Jeremy G. Siek, Ronald Garcia, and Walid Taha. 2009. 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*. 17–31. https://doi.org/10.1007/978-3-642-00590-9_2

Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*. 81–92.

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

Jeremy G. Siek and Philip Wadler. 2010. Threesomes, with and without blame. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. 365–376. https://doi.org/10.1145/1706299.1706342

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

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

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. 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*. 395–406. https://doi.org/10.1145/1328438.1328486

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

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

Dan S. Wallach and Edward W. Felten. 1998. Understanding Java Stack Inspection. In *Security and Privacy - 1998 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 3-6, 1998, Proceedings*. 52–63. https://doi.org/10.1109/SECPRI.1998.674823

Mitchell Wand. 1991. 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*. 294–311. https://doi.org/10.1007/3-540-55511-0_15

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (Nov. 1994), 38–94.

Ningning Xie, Xuan Bi, and Bruno C. d. S. Oliveira. 2018. 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*. 3–30. https://doi.org/10.1007/978-3-319-89884-1_1