Blame and Coercion: Together Again for the Second Time

ANONYMOUS AUTHOR(S)

Gradual Typing is great. Implementing gradually typed with blame tracking and space efficiency is tricky. There exist two technique to do this: coercion and threesome. Coercion is easy to understand, and easy enough to implement, but difficult to reason about formally. Threesome is hard to understand, easy to implement, and easy to reason about formally. We propose hyper-coercion, which is easy to understand, as easy to implement as coercion, and easy to reason about.

Additional Key Words and Phrases: Gradual Typing, Blame, Coercion

1 INTRODUCTION

KC: I want to add more citations here

Wadler and Findler [2009] introduces blame calculus, an intermediate language for gradually typed languages. Blame calculus includes blame labels, which tells the position of the cast that fails at run time (blame tracking).

Implementing gradually typed languages on top of blame calculus suffices sever space leak. Translation to blame calculus might wrap a tail call with a cast. Thus, at runtime, mutually tail-recursive functions can accumulate casts at tail position.

In 2007, Herman et al. [2010] proposes a solution to this problem by translating Blame Calculus to a variant of Henglein [1994]'s Coercions Calculus. The key idea is to represent casts with coercions. Coercions can be composed and normalized. Their solution, however, doesn't include blame tracking.

After that, many efforts have been made to combine blame tracking and space efficiency.

Siek et al. [2009] incorporate blame tracking into Coercion Calculus by decorating coercions with labels. They also propose that there are four blame strategies for their Coercion Calculi: $\{Lazy, Eager\} \times \{D, UD\}$. Lazy strategies blame fewer programs than eager ones, but also detect less potential type errors. D and UD assign blame labels differently. They prove that $LazyUD$ simulates the Blame Calculus in [Wadler and Findler 2009]. But the counterparts of other strategies in Blame Calculus is unknown.

Siek and Wadler [2010] proposes another approach to combine blame tracking and space efficiency. Their solution is based on Threesome Calculus, an novel alternative to Coercion Calculus. The key idea is to represent casts with threesomes. Threesomes, like coercions, can be composed. There is no separate normalization for threesome because every threesome is normalized. They prove that their Threesome Calculus bisimulate Siek et al. [2009]'s Coercion Calculus and Wadler and Findler [2009]'s Blame Calculus.

Siek and Garcia [2012] introduces a *lazyD* Blame Calculus. They conjecture that this calculus bi-simulate the *lazyD* Coercion Calculi.

Following Siek and Wadler [2010], Garcia [2013] implement all other blame strategies with Threesome Calculus. He claims that coercion with labels is easy to understand but hard to implement, and that threesome with labels, however, is easy to implement but hard to understand. His claim is later affirmed by the group of people who develop threesome Siek et al. [2015]. The connection between his Threesome Calculi and Siek et al. [2009]'s Coercion Calculi are established by the fact that the former are derived from the latter. The connection between these calculi and Blame Calculus, however, is still unclear.

2020. 2475-1421/2020/1-ART1 $15.00
https://doi.org/

Siek et al. [2015] revisit the coercion-based approach. They simplify the Coercion Calculi by only working with coercions in canonical forms. Fortunately, coercions are canonical when initially constructed, and their empowered compose function produces canonical coercions as well. Again (Siek and Wadler [2010]), they delegate all strategies other than *LazyUD*.

Last year, Kuhlenschmidt et al. [2018] present Grift, a space-efficient and blame-tracking compiler for a gradually typed language of the same name. This implementation is based on the *LazyD* Coercion Calculus. Their result suggests that implementing coercion is practical. And this is the first time product types is considered. Their treatment to products in coercions, however, is shown incorrect (Gradual-Typing [[n. d.]]).

Recently, New et al. [2019] show that *Eager* strategies are incompatible with η-equivalence of functions, which suggest that these strategies are not very ideal. Their research kills questions about *Eager* strategies, but some questions about *Lazy* strategies remain open:

- Is the *LazyD* coercion an ideal cast representation?
- Does the *LazyD* Coercion Calculus bi-simulate the Blame Calculus?

The *LazyD* coercion is claimed easy to understand ([Garcia 2013], [Siek et al. 2015]) and is shown easy enough to implement in a compiler ([Kuhlenschmidt et al. 2018]). It is still undesirable, however, in a few aspects. Firstly, its compose function is not structurally recursive. Many developers of Grift report that it is tricky to convince their proof assistants that composition terminates. Secondly, canonical coercion obscures the nature of a "normalized cast" because its definition lies on top of coercion.

Perhaps unsurprisingly, threesome has a straightforward recursive implementation. And its definition is self-standing. Together with the research by Garcia [2013], they suggest that there can be a cast representation whose definition is self-standing, and whose composition is structurally recursive. Super-coercion introduced in Garcia [2013] could be a promising candidate. However, its definition is a bit complicated. It uses 10 constructors to deal with an elementary type system with only base types and function types. And four constructors are directly related to function types. Thus, super-coercion might not scale very well to more sophisticated type systems.

We present yet another cast representation, hyper-coercion. Its composition is structurally recursive. And its definition is self-standing. Besides, there is a clear connection between it and the canonical coercion, so we are optimistic that implementing hyper-coercion should be as easy (or as hard) as coercion.

Our hyper-coercion considers sum types and product types, which are not accounted in all proofs above. Adding each of them requires us to add only one new constructor to a component of hyper-coercion. This suggests that hyper-coercion might scale better than super-coercion.

We prove formally that *LazyD* (resp. *LazyUD*) hyper-coercion calculi bi-simulate the *LazyD* (resp. *LazyUD*) Blame Calculi. This is possibly the first bi-simulation proof for the *LazyD* Blame Calculus.

The structure of this paper is as follows. Section 2 reviews the state-of-art of *Lazy* Coercion Calculi. In section 4 we present Hyper-coercion. Section 5 concludes.

2 BLAME CALCULUS

Fig. 1 defines the form of blame calculus and its static semantics. It is little changed from previous definitions. The dynamic semantics of Blame Calculi depend on blame strategies, so we defer them to sub-sections.

Blame Calculus is based on Simply Typed Lambda Calculus with sum types and product types (`STLC+`). Let S, T range over types. A type is either the dynamic type $\star$ (a.k.a. `Dyn`, ?, or `Unknown`), or

$$
\begin{array}{llll}
S, T & ::= & \star \mid P & \text{types} \\
P, Q & ::= & \iota \mid T_1 \to T_2 \mid T_1 \times T_2 \mid T_1 + T_2 \mid & \text{pre-types} \\
e & ::= & x \mid \mathtt{tt} \mid \lambda^{S \to T} x.t \mid e_1\ e_2 \mid \langle T \Leftarrow^{l} S \rangle t \mid \mathtt{blame}\ l & \text{terms} \\
 & \mid & \mathtt{cons}\ e_1\ e_2 \mid \mathtt{car}\ e \mid \mathtt{cdr}\ t & \\
 & \mid & \mathtt{inl}\ e \mid \mathtt{inr}\ e \mid \mathtt{case}\ e_1\ e_2\ e_3 & \\
o & ::= & \mathtt{tt} \mid \mathtt{fun} \mid \mathtt{cons} \mid \mathtt{inl} \mid \mathtt{inr} \mid \mathtt{blame}\ l & \text{observations}
\end{array}
$$

$$\boxed{S \sim T}$$

$$
\frac{}{\star \sim \star} \qquad \frac{}{\star \sim P} \qquad \frac{}{P \sim \star}
$$

$$
\frac{}{\iota \sim \iota} \qquad \frac{S_1 \sim S_2 \qquad T_1 \sim T_2}{S_1 \to T_1 \sim S_2 \to T_2} \qquad \frac{S_1 \sim S_2 \qquad T_1 \sim T_2}{S_1 \times T_1 \sim S_2 \times T_2} \qquad \frac{S_1 \sim S_2 \qquad T_1 \sim T_2}{S_1 + T_1 \sim S_2 + T_2}
$$

$$\boxed{S \smile T}$$

$$
\frac{}{\star \smile \star} \qquad \frac{}{\star \smile P} \qquad \frac{}{P \smile \star}
$$

$$
\frac{}{\iota \smile \iota} \qquad \frac{}{S_1 \to T_1 \smile S_2 \to T_2} \qquad \frac{}{S_1 \times T_1 \smile S_2 \times T_2} \qquad \frac{}{S_1 + T_1 \smile S_2 + T_2}
$$

$$\boxed{\Gamma \vdash e \vdash T}$$

$$
\frac{S \sim T \qquad \Gamma \vdash e : S}{\Gamma \vdash \langle T \Leftarrow^{l} S \rangle t : T} \qquad \frac{}{\Gamma \vdash \mathtt{blame} l : T}
$$

Fig. 1. Blame Calculus and its static semantics

a pre-type. Let P, Q range over pre-types. Every pre-type is a type with a traditional type constructor at the top.

$S \sim T$ reads S and T are consistent. Two types are consistent if one of them is $\star$, or they have the same top-most type constructor and the corresponding sub-parts are consistent. Consistency is reflexive and symmetric, but not transitive.

$S \smile T$ reads S and T are shallowly-consistent. Two types are shallowly-consistent if one of them is $\star$, or they have the same top-most type constructor. Shallow-consistency is also reflexive, symmetric, and not transitive.

Let e ranges over terms. Unlike STLC+, we annotate the co-domain of lambda abstractions explicitly. Besides, we add casts and blames.

Let o ranges over observations.

Now let's move to the dynamic semantics.

2.1 *LazyD* Blame Calculus

We describe the dynamic semantics of Blame Calculus with the CEK machine (Felleisen and Friedman [1986]).

2.2 *LazyUD* Blame Calculus

subtyping, reduction ...

$$
\begin{array}{lcll}
c & ::= & id* \mid (h, (b, t)) & \text{hyper-coercions} \\
h & ::= & \epsilon \mid ?^{l} & \text{heads} \\
b & ::= & U \mid c_1 \rightarrow c_2 \mid c_1 \times c_2 \mid c_1 + c_2 & \text{bodies} \\
t & ::= & \epsilon \mid ! \mid \bot^{l} & \text{tails}
\end{array}
$$

$$\boxed{c \,;^{l}\, c = c}$$

$$
\begin{array}{rclcll}
id* & ;^{l} & id* & = & id* & \\
id* & ;^{l} & (?^{l'}, (b, t)) & = & (?^{l'}, (b, t)) & \\
id* & ;^{l} & (!, (b, t)) & = & (?^{l}, (b, t)) & \\
(h, (b, \bot^{l'})) & ;^{l} & c & = & (h, (b, \bot^{l'})) & \\
(h, (b_1, t_1)) & ;^{l} & id* & = & (h, (b_1, !)) & t_1 \neq \bot^{l'} \\
(h, (b_1, t_1)) & ;^{l} & (\epsilon, (b_2, t_2)) & = & (h, b_1 \,;^{l}\, (b_2, t_2)) & t_1 \neq \bot^{l'} \\
(h, (b_1, t_1)) & ;^{l} & (?^{l'}, (b_2, t_2)) & = & (h, b_1 \,;^{l'}\, (b_2, t_2)) & t_1 \neq \bot^{l'}
\end{array}
$$

$$\boxed{b \,;^{l}\, (b, t) = (b, t)}$$

$$
\begin{array}{rclcl}
U & ;^{l} & (U, t) & = & (U, t) \\
c_1 \rightarrow c_2 & ;^{l} & (c_3 \rightarrow c_4, t) & = & (c_3 \,;^{l}\, c_1 \rightarrow c_2 \,;^{l}\, c_4, t) \\
c_1 \times c_2 & ;^{l} & (c_3 \times c_4, t) & = & (c_3 \,;^{l}\, c_1 \times c_2 \,;^{l}\, c_4, t) \\
c_1 + c_2 & ;^{l} & (c_3 + c_4, t) & = & (c_3 \,;^{l}\, c_1 + c_2 \,;^{l}\, c_4, t) \\
b_1 & ;^{l} & (b_2, t_2) & = & (b_1, \bot^{l})
\end{array}
$$

$$\boxed{seq(c, c) = c}$$

$$seq(c_1, c_2) \quad = \quad c_1 \,;^{l}\, c_2$$

$$\boxed{id(T) = c}$$

$$
\begin{array}{rcl}
id(*) & = & id* \\
id(P) & = & (\epsilon, (id(P), \epsilon))
\end{array}
$$

$$\boxed{id(P) = b}$$

$$
\begin{array}{rcl}
id(U) & = & U \\
id(T_1 \rightarrow T_2) & = & id(T_1) \rightarrow id(T_2) \\
id(T_1 \times T_2) & = & id(T_1) \times id(T_2) \\
id(T_1 + T_2) & = & id(T_1) + id(T_2)
\end{array}
$$

$$\boxed{cast(T, l, T) = c}$$

$$cast(T_1, l, T_2) \quad = \quad id(T_1) \,;^{l}\, id(T_2)$$

Fig. 2. *LazyD* Hyper-coercion

3 COERCION CALCULUS

3.1 *LazyD* Coercion Calculus

3.2 *LazyUD* Coercion Calculus

4 HYPER-COERCION

4.1 *LazyD* Hyper-coercion

The syntax of *LazyD* Hyper-coercion is shown in Fig. 2.

Theorem 1 (*LazyD* Hyper-coercion is a proper cast representation).

(1) *If* $v : T$, *then* $id(T)\ v = \texttt{succ}\ v$
(2) *If* $v : T_1$, c_1 *is a hyper-coercion from* T_1 *to* T_2, *and* c_2 *is from* T_2 *to* T_3, *then* $seq(c_1, c_2)\ v = (c_1\ v) >>= c_2$
(3) *If* $v : T_1$ *and* $\neg T_1 \smile T_2$, *then* $cast(T_1, l, T_2)\ v = \texttt{fail}\ l$
(4) *If* $v : \star$, *then* $cast(\star, l, \star)\ v = \texttt{succ}\ v$
(5) *If* $v : P$, *then* $cast(\star, l, Q)\ (inj\ P\ v) = cast(P, l, Q)\ v$
(6) *If* $v : P$, *then* $cast(P, l, \star)\ v = \texttt{succ}\ (inj\ P\ v)$
(7) *If* $v : \iota$, *then* $cast(\iota, l, \iota)\ v = \texttt{succ}\ v$
(8) $cast(S_1 \to T_1, l, S_2 \to T_2)\ (fun\ c_1\ E\ b\ c_2)$
$= \texttt{succ}\ fun\ seq(cast(S_2, l, S_1), c_1)\ E\ b\ seq(c_2, cast(T_1, l, T_2))$
(9) $cast(S_1 \times T_1, l, S_2 \times T_2)\ (cons\ v_1\ c_1\ v_2\ c_2)$
$= \texttt{succ}\ (cons\ v_1\ seq(c_1, cast(S_1, l, S_2))\ v_2\ seq(c_2, cast(T_1, l, T_2)))$
(10) $cast(S_1 + T_1, l, S_2 + T_2)\ (inl\ v\ c) = \texttt{succ}\ (inl\ v\ seq(c, cast(S_1, l, T_1)))$
(11) $cast(S_1 + T_1, l, S_2 + T_2)\ (inr\ v\ c) = \texttt{succ}\ (inr\ v\ seq(c, cast(S_2, l, T_2)))$

Proposition 1 (Every proper cast represenation is correct). *If* $\emptyset \vdash e : T$ *and* $o : T$

$$e \Downarrow_B^D o \text{ if and only if } e \Downarrow_C^D o$$

Corollary 1 (*LazyD* hyper-coercion is correct). *If* $\emptyset \vdash e : T$ *and* $o : T$

$$e \Downarrow_B^D o \text{ if and only if } e \Downarrow_H^D o$$

4.2 *LazyUD* Hyper-coercion

5 CONCLUSION

REFERENCES

Matthias Felleisen and Daniel P Friedman. 1986. *Control Operators, the SECD-machine, and the [1]-calculus*. Indiana University, Computer Science Department.

Ronald Garcia. 2013. Calculating threesomes, with blame. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 417–428.

Gradual-Typing. [n. d.]. Type-based-casts and coercions are not equivalent. ([n. d.]). https://github.com/Gradual-Typing/Grift/issues/81

Fritz Henglein. 1994. Dynamic typing: Syntax and proof theory. *Science of Computer Programming* 22, 3 (1994), 197–230.

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher-Order and Symbolic Computation* 23, 2 (2010), 167.

Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G Siek. 2018. An efficient compiler for the gradually typed lambda calculus. In *Scheme and Functional Programming Workshop*, Vol. 18.

Max S New, Daniel R Licata, and Amal Ahmed. 2019. Gradual type theory. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 15.

Jeremy Siek, Ronald Garcia, and Walid Taha. 2009. Exploring the design space of higher-order casts. In *European Symposium on Programming*. Springer, 17–31.

Jeremy Siek, Peter Thiemann, and Philip Wadler. 2015. Blame and coercion: together again for the first time. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 425–435.

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*. ACM, 68–80.

Jeremy G Siek and Philip Wadler. 2010. Threesomes, with and without blame. In *ACM Sigplan Notices*, Vol. 45. ACM, 365–376.

Philip Wadler and Robert Bruce Findler. 2009. Well-typed programs can't be blamed. In *European Symposium on Programming*. Springer, 1–16.

A APPENDIX

Text of appendix …