Hypercoercions and a Framework for Equivalence of Cast Calculi

KUANG-CHEN LU, Indiana University, United States
JEREMY G. SIEK, Indiana University, United States
ANDRE KUHLENSCHMIDT, Indiana University, United States

Designing a space-efficient cast representation that is good for both mechanized metatheory and implementation is challenging. Existing solutions (i.e. coercions, threesomes, and supercoercions) are good for one or the other. This paper presents a new cast representation, named hypercoercions, that is good for both. On the way to proving the correctness of hypercoercions, this paper also makes progress on a general framework for proving the correctness of cast representations.

CCS Concepts: • **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages.*

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

ACM Reference Format:
Kuang-Chen Lu, Jeremy G. Siek, and Andre Kuhlenschmidt. 2020. Hypercoercions and a Framework for Equivalence of Cast Calculi. In *Informal Proceedings of the first ACM SIGPLAN Workshop on Gradual Typing (WGT20).* ACM, New York, NY, USA, 24 pages.

1 INTRODUCTION

Around 2006, several groups of researchers proposed ways to integrate dynamic typing and static typing, notably gradual typing [Siek and Taha 2006], hybrid typing [Knowles and Flanagan 2010], migratory typing [Tobin-Hochstadt and Felleisen 2006], and multi-language interoperability [Gray et al. 2005; Matthews and Findler 2007]. Researchers usually define the semantics of gradually typed languages by translation to an intermediate language with casts, such as the blame calculus [Wadler and Findler 2009] and other cast calculi [Siek et al. 2009]. Unfortunately, straightforward implementations of casts on higher-order values (functions, objects, etc.) impose significant runtime overheads that can change the asymptotic space complexity of a program [Herman et al. 2010]. There are several known space-efficient cast representations, with various strengths and weaknesses [Garcia 2013; García-Pérez et al. 2014; Kuhlenschmidt et al. 2018; Siek et al. 2015; Siek and Garcia 2012; Siek and Wadler 2010]. The current state of the art includes

- threesomes [Garcia 2013; Siek and Wadler 2010],
- supercoercions [Garcia 2013], and
- coercions in normal form [Siek et al. 2015; Siek and Garcia 2012].

These systems compress casts using a compose operator. Threesomes and supercoercions are good for mechanized metatheory because their compose operators are structurally recursive, making them easy to define in a proof assistant such as Agda. In contrast, the coercions in normal form have compose operators that are not structurally recursive, which makes it more difficult to define in Agda, requiring what amounts to an explicit proof of termination. On the other hand, coercions in normal form are easier to understand than threesomes (with a strange labeled bottom type), and supercoercions (10 different kinds).

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

This paper presents a new cast representation, named *hypercoercions*, that is good for both mechanized metatheory and good for implementation. The composition operator for hypercoercions is defined by structural recursion and hypercoercions are suggestive of a bit-level representation that minimizes the need for pointers and fits all first-order casts into 64 bits. We present two flavors of hypercoercions to support the two blame tracking strategies from the literature: D and UD [Siek et al. 2009]. With the D strategy, only **d**own casts (casts from $\star$ to some other types) are subject to blame. With the UD strategy, however, some **u**p casts are also blamable. We are interested in the D blame tracking strategy because it comes with a more straightforward notion of safe cast compared to UD [Siek et al. 2009], which is why D was chosen in the Grift compiler [Kuhlenschmidt et al. 2018]. We are also interested in UD because it plays a prominent role in the gradual typing literature [Wadler and Findler 2009]. The UD hypercoercions were inspired by the supercoercions of Garcia [2013] (hence the name) and the D hypercoercions were inspired by the normal forms of Siek and Garcia [2012]. The semantics of casts can be lazy or eager [Siek et al. 2009]. In this paper, we focus on lazy cast strategies because we suspect that they are more efficient than eager strategies and because New et al. [2019] show that the eager strategies are incompatible with η-equivalence of functions.

Of course, an alternative cast representation must be proved correct. This paper presents steps toward a general framework for proving equivalence of cast calculi and, in particular, proves that an abstract machine using Lazy D hypercoercions is equivalent to an abstract machine using standard Lazy D casts [Siek et al. 2009]. We conjecture that the framework can be generalized to Lazy UD and that it can be applied to coercions in normal form, threesomes, and supercoercions.

To summarize, the primary contributions of this paper are:

- hypercoercions, a new cast representation, which has a structurally recursive composition and a memory representation more compact than space-efficient coercions.
- a framework in Agda for proving the correctness of Lazy D cast representations.
- a formal proof that Lazy D hypercoercions respect the semantics of the Lazy D cast calculus.

In Section 2 we review cast calculi and coercions. We present hypercoercions in Section 3. We present a framework for proving the correctness of Lazy D cast representations in Section 4 and use it to prove the correctness of Lazy D hypercoercions in Section 5.

2 BACKGROUND

In this section, we first review lazy cast calculi (Section 2.1), where we present an abstract machine that is employed in our framework (Section 4). Then we review coercions and their normal forms (Section 2.2), which motivates and inspires the design of hypercoercions (Section 3).

2.1 Cast Calculi

Syntax and Static Semantics. The syntax and static semantics are the same for the Lazy D and Lazy UD cast calculi. They are reviewed in Fig. 1. As usual, the important features are the cast expressions, $e\langle T_1 \Rightarrow^l T_2\rangle$, which are responsible for runtime type checking, and blame expressions, `blame` l, that raise errors. Polarities of blame labels is not treated but are straightforward to incorporate. The syntax and static semantics is the same as that of Siek et al. [2009] except for a few minor exceptions:

- We add sum, product, and unit types.
- We separate types into those with a type constructor at the top, the *pretypes* (`Unit`, functions, products, and sums), versus the dynamic type $\star$ (a.k.a. `Dyn` or ?).

Syntax

$$
\begin{array}{llll}
\text{Types} & T, S & ::= & \star \mid P \\
\text{Pre-types} & P & ::= & \mathtt{Unit} \mid T \to T \mid T \times T \mid T + T \\
\text{Terms} & e & ::= & x \mid \mathtt{unit} \mid \lambda^{T \to T} x.e \mid (e\ e) \mid \mathtt{cons}^{T \times T}\ e\ e \mid \mathtt{fst}\ e \mid \mathtt{snd}\ e \\
& & \mid & \mathtt{inl}^{T+T}\ e \mid \mathtt{inr}^{T+T}\ e \mid \mathtt{case}\ e\ e\ e \mid e\langle T \Rightarrow^{l} T\rangle \mid \mathtt{blame}\ l
\end{array}
$$

Consistency $\boxed{T_1 \sim T_2}$

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

$$
\frac{}{\mathtt{Unit} \sim \mathtt{Unit}} \qquad \frac{S_1 \sim S_2 \quad T_1 \sim T_2}{S_1 \to T_1 \sim S_2 \to T_2} \qquad \frac{S_1 \sim S_2 \quad T_1 \sim T_2}{S_1 \times T_1 \sim S_2 \times T_2} \qquad \frac{S_1 \sim S_2 \quad T_1 \sim T_2}{S_1 + T_1 \sim S_2 + T_2}
$$

Term typing $\boxed{\Gamma \vdash e : T}$

$$
\ldots \qquad \frac{\Gamma \vdash e : T_1 \quad T_1 \sim T_2}{\Gamma \vdash e\langle T_1 \Rightarrow^{l} T_2\rangle : T_2} \qquad \frac{}{\Gamma \vdash \mathtt{blame}\ l : T}
$$

Fig. 1. Syntax and static semantics of the cast calculi.

- We annotate the type of many expressions explicitly because we refer to them in the dynamic semantics. These expressions include cons expressions, left injections, right injections, and lambda abstractions.

As usual, the source T_1 and target types T_2 of a cast $e\langle T_1 \Rightarrow^{l} T_2\rangle$ are required to be consistent, written $T_1 \sim T_2$. The consistency relation is standard and defined in Fig. 1.

Dynamic Semantics. The dynamic semantics of a cast calculus is typically defined with a reduction semantics. Here we use a CEK machine [Felleisen and Friedman 1986] instead because the first author is more familiar with CEK machines and believes that a CEK machine is more convenient to use for the space-efficient semantics in Section 4.2 (a major point of space-efficiency is about compressing continuations). So using an abstract machine for the cast calculi in this section makes it easier to prove the correctness of the space-efficient machines. Of course, one should prove that the abstract machine presented here is equivalent to the standard reduction semantics for cast calculi, but we have not yet done so.

Fig. 3 defines the transition relation of the CEK machine and Fig. 2 gives a grammar for machine states s, including a definition of values v and value typing. The transitions involving casts are highlighted in red and described in more detail below. The other transitions are standard for a CEK machine for an extended simply typed lambda calculus. Recall that CEK machines involve two kinds of transitions, (1) those that dive further into an expression (looking for a redex) and push an entry onto the continuation, and (2) those that return a value to the current continuation and possibly perform a computation. Corresponding to (1) and (2), the machine state is either in an evaluating $\langle e, \mathcal{E}, \kappa\rangle$ or continuing $\langle v, \kappa\rangle$ configuration, respectively. Additionally, there is the `Halt` o configuration, where the machine halts with an observable (o). Observables include all value constructors, blame, and dyn, as in Siek and Garcia [2012]. The function converting values to observables ($observe(v) = o$) is defined in the obvious way.

Let v range over values. A value is either the unit, a function, a pair, a left injection, a right injection, or a casted value. Value typing rules restrict the casts in casted values. If the cast is between pre-types, source and target types must have the same type constructor at the top ($P_1 \smile P_2$). If the cast is from a pre-type to the dynamic type, the pre-type must be injectable. The definition of

Machine state and other runtime data structures

$$\begin{array}{llcl}
\text{Environments} & \mathcal{E} & ::= & \text{a partial function } \{\langle x, v\rangle, \dots\} \\
\text{Values} & v & ::= & \texttt{unit} \mid \langle \lambda x.e, \mathcal{E}\rangle \mid \texttt{cons}\ v\ v \mid \texttt{inl}\ v \mid \texttt{inr}\ v \mid v\langle c\rangle \\
\text{Casts} & c & ::= & T \Rightarrow^{l} T \\
\text{Injectable types (Lazy D)} & I & ::= & P \\
\text{Injectable types (Lazy UD)} & I & ::= & \texttt{Unit} \mid \star \to \star \mid \star \times \star \mid \star + \star \\
\text{Observables} & o & ::= & \texttt{dyn} \mid \texttt{unit} \mid \texttt{fun} \mid \texttt{cons} \mid \texttt{inl} \mid \texttt{inr} \mid \texttt{blame}\ l \\
\text{Cast results} & r & ::= & \texttt{succ}\ v \mid \texttt{fail}\ l \\
\text{States} & s & ::= & \langle e, \mathcal{E}, \kappa\rangle \mid \langle v, \kappa\rangle \mid \texttt{Halt}\ o \\
\text{Continuations} & \kappa & ::= & \texttt{stop} \mid [\texttt{cons}^{T\times T}\ \square\ \langle e, \mathcal{E}\rangle]\kappa \mid [\texttt{cons}^{T\times T}\ v\ \square]\kappa \mid [\texttt{inl}^{T+T}\ \square]\kappa \\
 & & \mid & [\texttt{inr}^{T+T}\ \square]\kappa \mid [\square\ \langle e, \mathcal{E}\rangle]\kappa \mid [v\ \square]\kappa \mid [\texttt{fst}\ \square]\kappa \\
 & & \mid & [\texttt{snd}\ \square]\kappa \mid [\texttt{case}\ \square\ \langle e, \mathcal{E}\rangle\ \langle e, \mathcal{E}\rangle]\kappa \mid [\texttt{case}\ v\ \square\ \langle e, \mathcal{E}\rangle]\kappa \\
 & & \mid & [\texttt{case}\ v\ v\ \square]\kappa \mid [\square\langle c\rangle]\kappa
\end{array}$$

Shallow-consistency $\boxed{T \smile T}$

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

$$\frac{}{\texttt{Unit} \smile \texttt{Unit}} \qquad \frac{}{T_{11} \to T_{12} \smile T_{21} \to T_{22}} \qquad \frac{}{T_{11} \times T_{12} \smile T_{21} \times T_{22}} \qquad \frac{}{T_{11} + T_{1} \smile S_{2} + T_{2}}$$

Value typing $\boxed{v : T}$

$$\dots \qquad \frac{v : I}{v\langle I \Rightarrow^{l} \star\rangle : \star} \qquad \frac{v : P_1 \qquad P_1 \smile P_2}{v\langle P_1 \Rightarrow^{l} P_2\rangle : P_2}$$

Fig. 2. Definition of machine state and auxiliary data structures.

injectable types depends on blame strategies: for the Lazy D strategy, every pre-type is injectable; for the Lazy UD strategy, a pre-types is injectable if all its sub-parts are the dynamic type.

The transition relation $s \longmapsto_{\mathcal{X}} s$ is parameterized over *applyCast* to allow for the differences between D and UD. When evaluating a cast expression, the machine moves the cast to the continuation and evaluates the inner expression. To apply a casted function, the machine first casts v_1 (the operand), then applies v_2 (the underlying function) to the casted operand, and then finally casts the return value. To take out the first (resp. second) part of a casted pair, the machine firstly takes out the first (resp. second) part of v, the underlying pair, and cast the result. To case split a casted injection, the machine moves the cast from the injection to continuations functions. To cast a value, the machine invokes *applyCast* on the value. If the cast succeeds ($\texttt{succ}\ v'$), the machine returns the result to the next continuation. If the cast fails ($\texttt{fail}\ l$), the machine halts with the blame label.

The reflexive transitive closure of reduction ($s \longmapsto^{*}_{\mathcal{X}} s$) and evaluation ($eval_{\S}(e)$) are standard [Felleisen and Flatt 2007].

Definition 2.1 (Lazy D CEK Machine). The Lazy D CEK machine, written $\mathcal{D}$, is the CEK machine of Fig. 3 using the *applyCast* for Lazy D defined in Fig. 4. We write the transition relations of this machine as $s \longmapsto_{\mathcal{D}} s$ and $s \longmapsto^{*}_{\mathcal{D}} s$ and write the evaluation function as $eval_{\mathcal{D}}(e) = o$.

We conjecture that $\mathcal{D}$ agrees with the Lazy D cast calculus of Siek et al. [2009].

Proposition 2.2 ($\mathcal{D}$ is deterministic). *If $s_1 \longmapsto_{\mathcal{D}} s_2$ and $s_1 \longmapsto_{\mathcal{D}} s_3$ then $s_2 = s_3$.*

Next, we define the CEK Machine for Lazy UD. The only difference with respect to Lazy D is in the definition of the *applyCast* function, in which a cast whose source or target is the dynamic

Transition $\boxed{s \longmapsto_{\mathcal{X}} s}$

$$
\begin{array}{rcl}
\langle e\langle T_1 \Rightarrow^l T_2\rangle, \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{X}} & \langle e, \mathcal{E}, [\square\langle T_1 \Rightarrow^l T_2\rangle]\kappa\rangle \\
\langle x, \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{X}} & \langle \mathcal{E}(x), \kappa\rangle \\
\langle \texttt{unit}, \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{X}} & \langle \texttt{unit}, \kappa\rangle \\
\langle \lambda^{T_1 \to T_2} x.e, \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{X}} & \langle\langle \lambda x.e, \mathcal{E}\rangle, \kappa\rangle \\
\langle \texttt{cons}^{T_1 \times T_2}\ e_1\ e_2, \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{X}} & \langle e_1, \mathcal{E}, [\texttt{cons}^{T_1 \times T_2}\ \square\ \langle e_2, \mathcal{E}\rangle]\kappa\rangle \\
\langle \texttt{inl}^{T_1 + T_2}\ e, \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{X}} & \langle e, \mathcal{E}, [\texttt{inl}^{T_1 + T_2}\ \square]\kappa\rangle \\
\langle \texttt{inl}^{T_1 + T_2}\ e, \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{X}} & \langle e, \mathcal{E}, [\texttt{inr}^{T_1 + T_2}\ \square]\kappa\rangle \\
\langle (e_1\ e_2), \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{X}} & \langle e_1, \mathcal{E}, [\square\ \langle e_2, \mathcal{E}\rangle]\kappa\rangle \\
\langle \texttt{fst}\ e, \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{X}} & \langle e, \mathcal{E}, [\texttt{fst}\ \square]\kappa\rangle \\
\langle \texttt{snd}\ e, \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{X}} & \langle e, \mathcal{E}, [\texttt{snd}\ \square]\kappa\rangle \\
\langle \texttt{case}\ e_1\ e_2\ e_3, \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{X}} & \langle e_1, \mathcal{E}, [\texttt{case}\ \square\ \langle e_2, \mathcal{E}\rangle\ \langle e_3, \mathcal{E}\rangle]\kappa\rangle \\
\langle \texttt{blame}\ l, \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{X}} & \texttt{Halt}\ (\texttt{blame}\ l) \\
\langle v_1, [\texttt{cons}^{T_1 \times T_2}\ \square\ \langle e_2, \mathcal{E}\rangle]\kappa\rangle & \longmapsto_{\mathcal{X}} & \langle e_2, \mathcal{E}, [\texttt{cons}^{T_1 \times T_2}\ v_1\ \square]\kappa\rangle \\
\langle v_2, [\texttt{cons}^{T_1 \times T_2}\ v_1\ \square]\kappa\rangle & \longmapsto_{\mathcal{X}} & \langle \texttt{cons}\ v_1\ v_2, \kappa\rangle \\
\langle v, [\texttt{inl}^{T_1 + T_2}\ \square]\kappa\rangle & \longmapsto_{\mathcal{X}} & \langle \texttt{inl}\ v, \kappa\rangle \\
\langle v, [\texttt{inr}^{T_1 + T_2}\ \square]\kappa\rangle & \longmapsto_{\mathcal{X}} & \langle \texttt{inr}\ v, \kappa\rangle \\
\langle v_1, [\square\ \langle e_2, \mathcal{E}\rangle]\kappa\rangle & \longmapsto_{\mathcal{X}} & \langle e_2, \mathcal{E}, [v_1\ \square]\kappa\rangle \\
\langle v_2, [(\langle \lambda x.e, \mathcal{E}\rangle)\ \square]\kappa\rangle & \longmapsto_{\mathcal{X}} & \langle e, \mathcal{E}[x := v_2], \kappa\rangle \\
\langle v_1, [v_2\langle T_1 \to T_2 \Rightarrow^l T_3 \to T_4\rangle\ \square]\kappa\rangle & \longmapsto_{\mathcal{X}} & \langle v_1, [\square\langle T_3 \Rightarrow^l T_1\rangle][v_2\ \square][\square\langle T_2 \Rightarrow^l T_4\rangle]\kappa\rangle \\
\langle \texttt{cons}\ v_1\ v_2, [\texttt{fst}\ \square]\kappa\rangle & \longmapsto_{\mathcal{X}} & \langle v_1, \kappa\rangle \\
\langle v\langle T_1 \times T_2 \Rightarrow^l T_3 \times T_4\rangle, [\texttt{fst}\ \square]\kappa\rangle & \longmapsto_{\mathcal{X}} & \langle v, [\texttt{fst}\ \square][\square\langle T_1 \Rightarrow^l T_3\rangle]\kappa\rangle \\
\langle \texttt{cons}\ v_1\ v_2, [\texttt{snd}\ \square]\kappa\rangle & \longmapsto_{\mathcal{X}} & \langle v_2, \kappa\rangle \\
\langle v\langle T_1 \times T_2 \Rightarrow^l T_3 \times T_4\rangle, [\texttt{snd}\ \square]\kappa\rangle & \longmapsto_{\mathcal{X}} & \langle v, [\texttt{snd}\ \square][\square\langle T_2 \Rightarrow^l T_4\rangle]\kappa\rangle \\
\langle v_1, [\texttt{case}\ \square\ \langle e_2, \mathcal{E}\rangle\ \langle e_3, \mathcal{E}\rangle]\kappa\rangle & \longmapsto_{\mathcal{X}} & \langle e_2, \mathcal{E}, [\texttt{case}\ v_1\ \square\ \langle e_3, \mathcal{E}\rangle]\kappa\rangle \\
\langle v_2, [\texttt{case}\ v_1\ \square\ \langle e_3, \mathcal{E}\rangle]\kappa\rangle & \longmapsto_{\mathcal{X}} & \langle e_3, \mathcal{E}, [\texttt{case}\ v_1\ v_2\ \square]\kappa\rangle \\
\langle v_3, [\texttt{case}\ (\texttt{inl}\ v)\ v_2\ \square]\kappa\rangle & \longmapsto_{\mathcal{X}} & \langle v, [v_2\ \square]\kappa\rangle \\
\langle v_3, [\texttt{case}\ (\texttt{inr}\ v)\ v_2\ \square]\kappa\rangle & \longmapsto_{\mathcal{X}} & \langle v, [v_3\ \square]\kappa\rangle \\
\langle v_3, [\texttt{case}\ v\langle T_1 + T_2 \Rightarrow^l T_3 + T_4\rangle\ v_2\ \square]\kappa\rangle & \longmapsto_{\mathcal{X}} & \langle v_3', [\texttt{case}\ v\ v_2'\ \square]\kappa\rangle \\
 & & \text{where } \kappa : T \\
 & & \text{and } v2' = v_2\langle T_3 \to T \Rightarrow^l T_1 \to T\rangle \\
 & & \text{and } v3' = v_3\langle T_4 \to T \Rightarrow^l T_2 \to T\rangle \\
\langle v, [\square\langle c\rangle]\kappa\rangle & \longmapsto_{\mathcal{X}} & \begin{cases} \langle v', \kappa\rangle & \text{if } \mathit{applyCast}(v, c) = \texttt{succ}\ v' \\ \texttt{Halt}\ (\texttt{blame}\ l) & \text{if } \mathit{applyCast}(v, c) = \texttt{fail}\ l \end{cases} \\
\langle v, \texttt{stop}\rangle & \longmapsto_{\mathcal{X}} & \texttt{Halt}\ \mathit{observe}(v)
\end{array}
$$

Evaluation $\boxed{\mathit{eval}_{\mathcal{X}}(e) = o}$

$$
\frac{\langle e, \emptyset, \texttt{stop}\rangle \longmapsto^*_{\mathcal{X}} \texttt{Halt}\ o}{\mathit{eval}_{\mathcal{X}}(e) = o}
$$

Fig. 3. Dynamic semantics of the cast calculi as a CEK machine. The transitions that involve casts are highlighted in red.

$$\boxed{applyCast(v, c) = r}$$

$$\begin{array}{rcll}
applyCast(v, \star \Rightarrow^{l} \star) & = & \mathtt{succ}\ v & \\
applyCast(v\langle P_1 \Rightarrow^{l_1} \star\rangle, \star \Rightarrow^{l_2} P_2) & = & applyCast(v, P_1 \Rightarrow^{l_2} P_2) & \\
applyCast(v, P \Rightarrow^{l} \star) & = & \mathtt{succ}\ v\langle P \Rightarrow^{l} \star\rangle & \\
applyCast(v, P_1 \Rightarrow^{l} P_2) & = & \mathtt{succ}\ v\langle P_1 \Rightarrow^{l} P_2\rangle & \text{if } P_1 \smile P_2 \\
applyCast(v, P_1 \Rightarrow^{l} P_2) & = & \mathtt{fail}\ l & \text{if } P_1 \not\smile P_2
\end{array}$$

Fig. 4. Definition of *applyCast* for Lazy D

$$\boxed{applyCast(v, c) = r}$$

$$\begin{array}{rcll}
applyCast(v, \star \Rightarrow^{l} \star) & = & \mathtt{succ}\ v & \\
applyCast(v, P \Rightarrow^{l} \star) & = & \mathtt{succ}\ v\langle P \Rightarrow^{l} I\rangle\langle I \Rightarrow^{l} \star\rangle & \text{if } I \sim P, I \neq P \\
applyCast(v, \star \Rightarrow^{l} P) & = & \mathtt{succ}\ v\langle \star \Rightarrow^{l} I\rangle\langle I \Rightarrow^{l} P\rangle & \text{if } I \sim P, I \neq P \\
applyCast(v\langle I \Rightarrow^{l} \star\rangle, \star \Rightarrow^{l} I) & = & \mathtt{succ}\ v & \\
applyCast(v\langle I_1 \Rightarrow^{l} \star\rangle, \star \Rightarrow^{l} I_2) & = & \mathtt{fail}\ l & \text{if } I_1 \neq I_2 \\
applyCast(v, P_1 \Rightarrow^{l} P_2) & = & \mathtt{succ}\ v\langle P_1 \Rightarrow^{l} P_2\rangle & \text{if } P_1 \smile P_2
\end{array}$$

Fig. 5. Definition of *applyCast* for Lazy UD.

$$\begin{array}{llcl}
\text{Injectable types (Lazy D)} & I & ::= & T \to T \mid \mathtt{Unit} \\
\text{Injectable types (Lazy UD)} & I & ::= & \star \to \star \mid \mathtt{Unit} \\
\text{Coercions} & c & ::= & I! \mid I?^{l} \mid \iota \mid \bot^{l} \mid c;c \mid c \to c \\
\text{normal coercions} & \hat{c} & ::= & \hat{c} \to \hat{c} \mid I?^{l};\hat{c} \to \hat{c} \mid \hat{c} \to \hat{c};I! \mid I?^{l};\hat{c} \to \hat{c};I! \mid I?^{l};\bot^{l} \\
 & & \mid & \iota \mid I?^{l} \mid I! \mid I?^{l};I! \mid \bot^{l}
\end{array}$$

Fig. 6. Syntax of coercions and normal forms à la Siek and Garcia [2012].

type $\star$ is always split into two casts that go through an injectable type, that is, a type in which all sub-components are the dynamic type, such as $\star \to \star$.

Definition 2.3 (Lazy UD CEK Machine). The Lazy UD CEK machine, written $\mathcal{UD}$, is the CEK machine of Fig. 3 using the *applyCast* for Lazy UD defined in Fig. 5. We write the transition relations of this machine as $s \longmapsto_{\mathcal{UD}} s$ and $s \longmapsto^{*}_{\mathcal{UD}} s$ and write the evaluation function as $eval_{\mathcal{UD}}(e) = o$.

We conjecture that $\mathcal{UD}$ agrees with the Lazy UD cast calculus of Siek et al. [2009].

Proposition 2.4 ($\mathcal{UD}$ is deterministic). *If $s_1 \longmapsto_{\mathcal{D}} s_2$ and $s_1 \longmapsto_{\mathcal{UD}} s_3$ then $s_2 = s_3$.*

2.2 Coercions and Normal Forms

In this section, we review the coercions [Henglein 1994; Herman et al. 2010] and the normal form of Siek and Garcia [2012] to motivate the design of hypercoercions. We omit sum types and product types in this section because Siek and Garcia [2012] did not discuss them. We assume a basic familiarity with coercions, and suggest that readers unfamiliar with coercions to familiarize themselves with Siek and Garcia [2012] and Siek et al. [2015].

Fig. 6 reviews the grammar for coercions, written c. To review, an injection $I!$ takes a value from an injectable type I to type $\star$. An injectable type is simply a type that can be cast directly to and

Syntax of Hypercoercions

$$\begin{array}{llcl}
\text{Injectable types (Lazy D)} & I & ::= & P \\
\text{Injectable types (Lazy UD)} & I & ::= & \texttt{Unit} \mid \star \to \star \mid \star \times \star \mid \star + \star \\
\text{Hypercoercions} & c & ::= & \texttt{id}\star \mid h \overset{m}{\curvearrowright} t \\
\text{Heads} & h & ::= & \epsilon \mid ?^{l} \\
\text{Middles} & m & ::= & \texttt{Unit} \mid c \to c \mid c \times c \mid c + c \\
\text{Tails} & t & ::= & \epsilon \mid\, ! \mid \bot^{l}
\end{array}$$

Hypercoercion typing $\boxed{c : T \Longrightarrow T}$

$$\frac{}{\texttt{id}\star : \star \Longrightarrow \star} \qquad \frac{h : T_1 \Longrightarrow P_1 \qquad m : P_1 \Longrightarrow P_2 \qquad t : P_2 \Longrightarrow T_2}{h \overset{m}{\curvearrowright} t : T_1 \Longrightarrow T_2}$$

Head typing $\boxed{h : T \Longrightarrow P}$

$$\frac{}{\epsilon : P \Longrightarrow P} \qquad \frac{}{?^{l} : \star \Longrightarrow I}$$

Middle typing $\boxed{m : T \Longrightarrow T}$

$$\frac{}{\texttt{Unit} : \texttt{Unit} \Longrightarrow \texttt{Unit}} \qquad \frac{c_1 : T_3 \Longrightarrow T_1 \qquad c_2 : T_2 \Longrightarrow T_4}{c_1 \to c_2 : T_1 \to T_2 \Longrightarrow T_3 \to T_4}$$

$$\frac{c_1 : T_1 \Longrightarrow T_3 \qquad c_2 : T_2 \Longrightarrow T_4}{c_1 \times c_2 : T_1 \times T_2 \Longrightarrow T_3 \times T_4} \qquad \frac{c_1 : T_1 \Longrightarrow T_3 \qquad c_2 : T_2 \Longrightarrow T_4}{c_1 + c_2 : T_1 + T_2 \Longrightarrow T_3 + T_4}$$

Tail typing $\boxed{t : P \Longrightarrow T}$

$$\frac{}{\epsilon : P \Longrightarrow P} \qquad \frac{}{! : I \Longrightarrow \star} \qquad \frac{}{\bot^{l} : P \Longrightarrow T}$$

Fig. 7. Definition of hypercoercions (HC)

from $\star$. The definition of injectable types depends on the blame strategy. With Lazy D, all pre-types are injectable. With Lazy UD, only $\star \to \star$ and `Unit` are injectable. A projection $I?^{l}$ takes a value from type $\star$ to type I, or halts the program and blames l if the value is of a different type. The coercion ι is the identity, $\bot^{l}$ is the coercion that always fails and blames l, and $(c_1; c_2)$ applies c_1 and then c_2 in sequence. The function coercion $c_1 \to c_2$ applies c_1 to the argument of a function and c_2 to the return value.

Coercions come with a reduction relation so it is natural to ask about their normal forms. Fig. 6 defines the syntax of coercion in normal form, written $\hat{c}$. Conceptually, a coercion in normal form has three parts, all of which are optional. It may start with a projection $I?^{l}$, followed by a function coercion, and then concluded with an injection $I!$ or a failure $\bot^{l}$. While this is a simple idea, there are 10 clauses in the definition for $\hat{c}$!

3 DEFINITION OF HYPERCOERCIONS

This section presents our first contribution, the definition of hypercoercions. The design of hypercoercions is motivated by the observation that a normal coercion has at most three parts. Hypercoercions make this structure explicit: a hypercoercion c is either `id`$\star$, the identity cast for $\star$, or it contains three parts: a head, middle, and tail, as defined in Fig. 7.

- The head h is either a projection or the no-op,

- the middle m involves coercions that preserve a type constructor, i.e., coercions between function, pair, sum types, and unit types, and
- the tail t is either an injection, a failure, or the no-op.

There is a close correspondence between middle coercions and type constructors. We generalize shallow-consistency to middles $m \smile m$ in the obvious way.

Subsection 3.1 defines the *id*, *seq*, and *cast* functions that construct Lazy D hypercoercions. Subsection 3.2 defines Lazy UD counterparts. The functions that apply hypercoercions to values are defined later in Section 5, after we introduce the framework and its new space-efficient definition of values in Section 4.

3.1 Lazy D Hypercoercions

Fig. 8 defines the functions *id*, *seq*, and *cast* that construct Lazy D hypercoercions. We use these functions to satisfy the Cast abstract data type defined in Section 4. Fig. 8 also defines some auxiliary functions.

The *seq* operator is defined in terms of coercion composition, which is the key to compressing coercions and maintaining space-efficiency. A composition operator for coercions typically requires the target type of the first coercion to match the source type of the second. However, it is useful to relax this restriction for the D blame tracking strategy. We shall need optional blame labels, written ℓ , that range over ϵ or l. The label is mandatory when the target type of the first coercion does not match the source type of the second. Then we write the composition of hypercoercions as $c_1 \,\text{⨾}^{\ell}\, c_2$. When both c_1 and c_2 are $\mathtt{id}\star$, their composition is also $\mathtt{id}\star$. If the head of c_2 is a projection, the result is c_2. Otherwise, we need a label to build the projection. Since the head of c_2 is the no-op, its source type must be a pretype. Thus we know ℓ must be a label and we put it in the projection. When c_1 ends with a failure, the composition is c_1 itself. In all the remaining cases, c_1 does not end with a failure. When m_1 and m_2 have different top constructors ($m_1 \not\smile m_2$), the result is a failure coercion, which needs a label. When c_2 starts with a projection, we blame the projection for casting a value to a shallowly inconsistent type. When c_2 starts with the no-op, we know ℓ must be a label, which is blamed for composing shallowly inconsistent hypercoercions. The last two cases compose m_1 and m_2 with the auxiliary function $m \,\text{⨾}^{\ell}\, m$, which assumes its inputs have the same top constructor. The definition of $m \,\text{⨾}^{\ell}\, m$ is a straightforward structural recursion. Going back to the last two cases of $c_1 \,\text{⨾}^{\ell}\, c_2$. When c_2 starts with a projection, we do not know whether the target type of m_1 matches the source of m_2. So we use the l in the projection to compose the middles $m_1 \,\text{⨾}^{l}\, m_2$. When c_2 starts with the no-op, we compose the middles using ℓ.

The definitions of $id(T)$ and $cast(T_1, l, T_2)$ are straightforward, although $cast(T_1, l, T_2)$ is unusual in its use of *id* and composition.

A proof of correctness for hypercoercions (in Section 4.3) relies on the following two basic properties of hypercoercions.

Proposition 3.1 (Lazy D Hypercoercions Form a Monoid). *For all* $c : T_1 \Longrightarrow T_2$, $c_1 : T_1 \Longrightarrow T_2$, $c_2 : T_2 \Longrightarrow T_3$, *and* $c_3 : T_3 \Longrightarrow T_4$,

(1) $seq(id(T_1), c) = c$,
(2) $seq(c, id(T_2)) = c$, *and*
(3) $seq(seq(c_1, c_2), c_3) = seq(c_1, seq(c_2, c_3))$.

Proof. Part (1) and (2) are by induction on c. Part (3) is by induction on c_2 and case analysis on c_1 then c_3. □

Proposition 3.2 (Lazy D Identity Casts). *For all* T *and* l, $cast(T, l, T) = id(T)$

Proof. By induction on T. □

$$\text{Optional labels} \quad \ell \quad ::= \quad \epsilon \mid l$$

$$\text{Composition of hypercoercions} \quad \boxed{c \fatsemi^{\ell} c = c}$$

$$\begin{array}{rclcll}
\mathtt{id}\star & \fatsemi^{\ell} & \mathtt{id}\star & = & \mathtt{id}\star & \\
\mathtt{id}\star & \fatsemi^{\ell} & ?^{l'} \overset{m}{\curvearrowright} t & = & ?^{l'} \overset{m}{\curvearrowright} t & \\
\mathtt{id}\star & \fatsemi^{l} & \epsilon \overset{m}{\curvearrowright} t & = & ?^{l} \overset{m}{\curvearrowright} t & \\
h \overset{m}{\curvearrowright} \bot^{l'} & \fatsemi^{\ell} & c & = & h \overset{m}{\curvearrowright} \bot^{l'} & \\
h \overset{m}{\curvearrowright} t & \fatsemi^{\ell} & \mathtt{id}\star & = & h \overset{m}{\curvearrowright} ! & \text{if } \forall l.t \neq \bot^{l} \\
h_1 \overset{m_1}{\curvearrowright} t_1 & \fatsemi^{\ell} & ?^{l} \overset{m_2}{\curvearrowright} t_2 & = & h_1 \overset{m_1}{\curvearrowright} \bot^{l} & \text{if } m_1 \not\smile m_2 \text{ and } \forall l.t_1 \neq \bot^{l} \\
h_1 \overset{m_1}{\curvearrowright} t_1 & \fatsemi^{l} & \epsilon \overset{m_2}{\curvearrowright} t_2 & = & h_1 \overset{m_1}{\curvearrowright} \bot^{l} & \text{if } m_1 \not\smile m_2 \text{ and } \forall l.t_1 \neq \bot^{l} \\
h_1 \overset{m_1}{\curvearrowright} t_1 & \fatsemi^{\ell} & ?^{l} \overset{m_2}{\curvearrowright} t_2 & = & h_1 \overset{m_1 \fatsemi^{l} m_2}{\curvearrowright} t_2 & \text{if } m_1 \smile m_2 \text{ and } \forall l.t_1 \neq \bot^{l} \\
h_1 \overset{m_1}{\curvearrowright} t_1 & \fatsemi^{\ell} & \epsilon \overset{m_2}{\curvearrowright} t_2 & = & h_1 \overset{m_1 \fatsemi^{\ell} m_2}{\curvearrowright} t_2 & \text{if } m_1 \smile m_2 \text{ and } \forall l.t_1 \neq \bot^{l}
\end{array}$$

$$\text{Composition of middles} \quad \boxed{m \fatsemi^{\ell} m = m}$$

$$\begin{array}{rclcl}
\mathtt{Unit} & \fatsemi^{\ell} & \mathtt{Unit} & = & \mathtt{Unit} \\
c_1 \to c_2 & \fatsemi^{\ell} & c_3 \to c_4 & = & c_3 \fatsemi^{\ell} c_1 \to c_2 \fatsemi^{\ell} c_4 \\
c_1 \times c_2 & \fatsemi^{\ell} & c_3 \times c_4 & = & c_1 \fatsemi^{\ell} c_3 \times c_2 \fatsemi^{\ell} c_4 \\
c_1 + c_2 & \fatsemi^{\ell} & c_3 + c_4 & = & c_1 \fatsemi^{\ell} c_3 + c_2 \fatsemi^{\ell} c_4
\end{array}$$

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

$$seq(c_1, c_2) \quad = \quad c_1 \fatsemi^{\epsilon} c_2$$

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

$$\begin{array}{rcl}
id(\mathtt{Unit}) & = & \mathtt{Unit} \\
id(T_1 \to T_2) & = & id(T_1) \to id(T_2) \\
id(T_1 \times T_2) & = & id(T_1) \times id(T_2) \\
id(T_1 + T_2) & = & id(T_1) + id(T_2)
\end{array}$$

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

$$\begin{array}{rcl}
id(\star) & = & \mathtt{id}\star \\
id(P) & = & \epsilon \overset{id(P)}{\curvearrowright} \epsilon
\end{array}$$

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

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

Fig. 8. Lazy D Hypercoercions

3.2 Lazy UD Hypercoercions

Hypercoercions for the UD blame tracking strategy have the same syntax as for D (Fig. 7), but the definitions of *id*, *cast*, and *seq*, differ from D. Again, the *seq* operator (Figure 9) is defined in terms of composition, but here composition $c_1 \fatsemi c_2$ makes the usual assumption that the target type of c_1 matches the source type of c_2. The definition of composition for UD is particularly straightforward.

Composition of hypercoercions $\boxed{c \fatsemi c = c}$

$$\begin{array}{rcrcl}
c & \fatsemi & \mathtt{id}\star & = & c \\
\mathtt{id}\star & \fatsemi & h_2 \overset{m_2}{\curvearrowright} t_2 & = & h_2 \overset{m_2}{\curvearrowright} t_2 \\
h_1 \overset{m_1}{\curvearrowright} \epsilon & \fatsemi & \epsilon \overset{m_2}{\curvearrowright} t_2 & = & h_1 \overset{m_1 \fatsemi m_2}{\curvearrowright} t_2 \\
h_1 \overset{m_1}{\curvearrowright} ! & \fatsemi & ?^l \overset{m_2}{\curvearrowright} t_2 & = & \begin{cases} h_1 \overset{m_1 \fatsemi m_2}{\curvearrowright} t_2 & \text{if } m_1 \smile m_2 \\ h_1 \overset{m_1}{\curvearrowright} \bot^l & \text{otherwise} \end{cases} \\
h_1 \overset{m_1}{\curvearrowright} \bot^l & \fatsemi & h_2 \overset{m_2}{\curvearrowright} t_2 & = & h_1 \overset{m_1}{\curvearrowright} \bot^l
\end{array}$$

Composition of middles $\boxed{m \fatsemi m = m}$

$$\begin{array}{rcrcl}
\mathtt{Unit} & \fatsemi & \mathtt{Unit} & = & \mathtt{Unit} \\
c \longrightarrow d & \fatsemi & c' \longrightarrow d' & = & (c' \fatsemi c) \longrightarrow (d \fatsemi d') \\
c \times d & \fatsemi & c' \times d' & = & (c \fatsemi c') \times (d \fatsemi d') \\
c + d & \fatsemi & c' + d' & = & (c \fatsemi c') + (d \fatsemi d')
\end{array}$$

Shallow consistency of middles $\boxed{m \smile m}$

$$\mathtt{Unit} \smile \mathtt{Unit} \quad (c \longrightarrow d) \smile (c' \longrightarrow d') \quad (c \times d) \smile (c' \times d') \quad (c + d) \smile (c' + d')$$

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

$$seq(c_1, c_2) \quad = \quad c_1 \fatsemi c_2$$

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

$$\begin{array}{rcl}
id(\mathtt{Unit}) & = & \mathtt{Unit} \\
id(T_1 \to T_2) & = & id(T_1) \to id(T_2) \\
id(T_1 \times T_2) & = & id(T_1) \times id(T_2) \\
id(T_1 + T_2) & = & id(T_1) + id(T_2)
\end{array}$$

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

$$\begin{array}{rcl}
id(\star) & = & \mathtt{id}\star \\
id(P) & = & \epsilon \overset{id(P)}{\curvearrowright} \epsilon
\end{array}$$

Fig. 9. Lazy UD Hypercoercions

The first two lines say that $\mathtt{id}\star$ acts as the identity on both the left and right. The third line handles the case when both the tail of c_1 and the head of c_2 are no-ops, in which case the middles types m_1 and m_2 are composed via an auxiliary composition operator. This compose operator for middle coercions assumes that its inputs are shallowly consistent. The fourth line handles the important case when the tail of c_1 is an injection and the head of c_2 is a projection. If the two middle coercions are shallowly consistent, then they can be composed. If not, the tail of the result is a failure coercion with the projection's label. The last line handles the case when the tail of c_1 is a failure, in which case the result of composition is c_1.

The definition of *id* is straightforward (Figure 9) .

Proposition 3.3 (Lazy UD Hypercoercions Form a Monoid). *For all $c : T_1 \Longrightarrow T_2$, $c_1 : T_1 \Longrightarrow T_2$, $c_2 : T_2 \Longrightarrow T_3$, and $c_3 : T_3 \Longrightarrow T_4$,*

(1) $seq(id(T_1), c) = c$,

$$\boxed{castToDyn(P, l) = c}$$

$$\begin{array}{rcl}
castToDyn(\star, l) & = & \mathtt{id}\star \\
castToDyn(P, l) & = & \epsilon \overset{m}{\curvearrowright} ! \\
& & \text{where } m = castToInj(P, l, ground(P))
\end{array}$$

$$\boxed{castFromDyn(P, l) = c}$$

$$\begin{array}{rcl}
castFromDyn(\star, l) & = & \mathtt{id}\star \\
castFromDyn(P, l) & = & ?^l \overset{m}{\curvearrowright} \epsilon \\
& & \text{where } m = castFromInj(ground(P), l, P)
\end{array}$$

$$\boxed{castToInj(P, l, I) = m}$$

$$\begin{array}{rcl}
castToInj(\mathtt{Unit}, l, \mathtt{Unit}) & = & \mathtt{Unit} \\
castToInj(T_1 \to T_2, l, \star \to \star) & = & castFromDyn(T_1, l) \to castToDyn(T_2, l) \\
castToInj(T_1 \times T_2, l, \star \times \star) & = & castToDyn(T_1, l) \times castToDyn(T_2, l) \\
castToInj(T_1 + T_2, l, \star + \star) & = & castToDyn(T_1, l) + castToDyn(T_2, l)
\end{array}$$

$$\boxed{castFromInj(I, l, P) = m}$$

$$\begin{array}{rcl}
castFromInj(\mathtt{Unit}, l, \mathtt{Unit}) & = & \mathtt{Unit} \\
castFromInj(\star \to \star, l, T_1 \to T_2) & = & castToDyn(T_1, l) \to castFromDyn(T_2, l) \\
castFromInj(\star \times \star, l, T_1 \times T_2) & = & castFromDyn(T_1, l) \times castFromDyn(T_2, l) \\
castFromInj(\star + \star, l, T_1 + T_2) & = & castFromDyn(T_1, l) + castFromDyn(T_2, l)
\end{array}$$

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

$$\begin{array}{rcll}
cast(\star, l, T_2) & = & castFromDyn(T_2, l) & \\
cast(T_1, l, \star) & = & castToDyn(T_1, l) & \\
cast(\mathtt{Unit}, l, \mathtt{Unit}) & = & \epsilon \overset{\mathtt{Unit}}{\curvearrowright} \epsilon & \\
cast(T_1 \to T_2, l, T_3 \to T_4) & = & \epsilon \overset{c_1 \to c_2}{\curvearrowright} \epsilon & \text{where } c_1 = cast(T_3, l, T_1) \\
& & & \text{and } c_2 = cast(T_2, l, T_4) \\
cast(T_1 \times T_2, l, T_3 \times T_4) & = & \epsilon \overset{c_1 \times c_2}{\curvearrowright} \epsilon & \text{where } c_1 = cast(T_1, l, T_3) \\
& & & \text{and } c_2 = cast(T_2, l, T_4) \\
cast(T_1 + T_2, l, T_3 + T_4) & = & \epsilon \overset{c_1 + c_2}{\curvearrowright} \epsilon & \text{where } c_1 = cast(T_1, l, T_3) \\
& & & \text{and } c_2 = cast(T_2, l, T_4)
\end{array}$$

Fig. 10. *cast* and its auxiliary functions for Lazy UD.

(2) $seq(c, id(T_2)) = c$, *and*
(3) $seq(seq(c_1, c_2), c_3) = seq(c_1, seq(c_2, c_3))$.

Proof. See the Agda proof at the following URL:
https://github.com/jsiek/gradual-typing-in-agda/blob/master/HyperCoercions.agda

□

The *cast* function is defined in Figure 10. Prior presentations of this function do not use auxiliary functions, as we do here. The reason that we introduce the auxiliary functions *castToDyn*, *castFromDyn*, *castToInj*, and *castFromInj*, is to ensure that each of them is structurally recursive, which makes them straightforward to define in Agda.

Proposition 3.4 (Lazy UD Identity Casts). *For all* T *and* l, $cast(T, l, T) = id(T)$

Proof. By induction on T. □

3.3 Compact Representation of Hypercoercions

Hypercoercions enable a bit-level representation that is particularly compact for identity coercions and coercions that project from base types or inject to base types. (Here `Unit` is the only base type, but in a real language the base types would include integers, Booleans, etc.) We conjecture that such coercions occur more frequently than the more complex coercions (e.g. between function types), especially because such coercions appear in the leaves of complex coercions.

Futhermore, values of base types are often stored in CPU registers, so it would be nice for coercions on base types to also fit in registers, that is, in 64 bits, so that applying a coercion to a value of base type would not require access to main memory, which is an order of magnitude slower than accessing registers on a modern CPU. Coercions involving non-base types, such as function types, may be arbitrarily deep, so in those cases, the hypercoercion representation has to be a pointer to a heap-allocated structure.

Here is a sketch for the bit-level representation for hypercoercions.

- 1 bit to differentiate between `id⋆` and three-part coercions.
- 1 bit to differentiate between middle coercions of base type versus non-base type.
- If the middle is of non-base type, then 61 bits represent a pointer to a heap-allocated structure. Heap allocated structures are usually 8-byte aligned, so there are 3 unused bits in a pointer.
- If the middle is of base types, then the remaining 62 bits are used to represent the head (25 bits), middle (11 bits), and tail (26 bits).
  - The head requires 1 bit to differentiate between ϵ and $?^l$ and then 24 bits could be used for the label l.
  - The middle would use 11 bits to differentiate all the base types.
  - The tail requires 2 bits to differentiate between ϵ, !, and $\perp^l$, and could use 24 bits for the label l.

  The bits for blame labels represent an index into a table of blame information. In the event that a program requires too many blame labels, then the implementation can fallback to using the heap-allocated structure for more coercions.

4 A FRAMEWORK FOR PROVING CORRECTNESS OF CAST REPRESENTATIONS

This section presents our second contribution, a framework for proving the correctness of cast representations, especially space-efficient ones. Section 5 applies this framework to prove the correctness of the Lazy D hypercoercions.

The framework includes an abstract data type named Cast ADT (Section 4.1), and a CEK machine $\mathcal{S}(C)$ that is parameterized over Cast ADTs (Section 4.2). This machine is space-efficient provided that the Cast ADT implementation performs compression. We conjecture that all cast representations defined in the literature are instances of this ADT. The framework is available at the following URL:

https://github.com/LuKC1024/hypercoercion-and-framework-wgt2020/tree/master/Proof

In Section 4.3 we prove that, for any instance C of the Cast ADT, if C satisfies a more refined abstract data type for Lazy D casts, then $\mathcal{S}(C)$ is equivalent to $\mathcal{D}$, that is,

$$eval_{\mathcal{S}(C)}(e) = o \text{ if and only if } eval_{\mathcal{D}}(e) = o$$

We conjecture that all of the cast representations in the literature for Lazy D (supercoercions, coercions in normal form, threesomes) are instances of the Lazy D Cast ADT. We are working on a similar theorem for Lazy UD cast representations.

We then apply this framework to Lazy D hypercoercions H (Section 5), where we show that H is an instance of the Lazy D Cast ADT, and therefore

$$eval_{\mathcal{S}(H)}(e) = o \text{ if and only if } eval_{\mathcal{D}}(e) = o$$

Before turning to the definition of the abstract data types, we first give the definition of values and cast results used by the $\mathcal{S}(C)$, as they are mentioned in the definition of the ADTs. The $\mathtt{Dyn}_I(v)$ stands for values that are cast to the dynamic type, as in Wadler and Findler [2009].

Definition 4.1 (Values and cast results for the $\mathcal{S}(C)$ machine).

Values	v	::=	$\mathtt{unit} \mid \langle \lambda x.\, e, c, c, \mathcal{E}\rangle \mid \mathtt{cons}\ v\langle c\rangle\ v\langle c\rangle \mid \mathtt{inl}\ v\langle c\rangle \mid \mathtt{inr}\ v\langle c\rangle \mid \mathtt{Dyn}_I(v)$
Cast results	r	::=	$\mathtt{succ}\ v \mid \mathtt{fail}\ l$
Environments	$\mathcal{E}$	::=	a partial function $\{\langle x, v\rangle, \dots\}$

Let v range over values. In $\mathcal{X}$ we have one value constructor for all casted values. In $\mathcal{S}(C)$, however, we do not have this generic value constructor, instead, we push casts into the ordinary values. Thus non-casted values in $\mathcal{X}$ correspond to values in $\mathcal{S}(C)$ where casts are identities. For instance, ($\mathtt{cons}\ v_1\ v_2$) corresponds to ($\mathtt{cons}\ v_1'\langle id(T_1)\rangle\ v_2'\langle id(T_2)\rangle$).

Cast results (r) and environments ($\mathcal{E}$) are the same as for $\mathcal{X}$. Value typing ($v : T$) is defined in the obvious way.

4.1 The Cast Abstract Data Types

The *Cast Abstract Data Type*, defined below, captures the set of operators that a cast representation C must provide for it to be used with the $\mathcal{S}(C)$ machine. The first three operators enable $\mathcal{S}(C)$ to construct casts so we call them *cast constructors*. The fourth and last operator enables $\mathcal{S}(C)$ to apply casts to values.

Definition 4.2 (Cast Abstract Data Type (Cast ADT)). A cast abstract data type is a set *Cast*, which is indexed by two types, with four operators:

$id(T) = c$ constructs an identity cast from a type
$seq(c, c) = c$ composes two casts
$cast(T, l, T) = c$ constructs a cast from a source type, a label, and a target type
$applyCast(v, c) = r$ applies a cast to a value, producing a cast result

We use the syntax $c : T_1 \Longrightarrow T_2$ to mean c is in the set $Cast\ T_1\ T_2$ and we say "c is from T_1 to T_2".

Next, we define the *Lazy D Cast ADT*, which captures the further requirements that are needed for the theorem that establishes equivalence to $\mathcal{D}$. Property (1) below states that $id(T)$ acts as the identity function. Property (2) states that $seq(c, c)$ acts as a sequence of casts. Properties (3) through (11) state that $cast(T_1, l, T_2)$ acts like the Lazy D *applyCast* for $\mathcal{X}$ (Fig. 4).

Definition 4.3 (Lazy D Cast ADT). A Cast is a Lazy D Cast if:

(1) If $v : T$ then $applyCast(v, id(T)) = \mathtt{succ}\ v$
(2) If $v : T_1$ and $c_1 : T_1 \Longrightarrow T_2$ and $c_2 : T_2 \Longrightarrow T_3$
then $applyCast(v, seq(c_1, c_2)) = applyCast(v, c_1) \underline{\gg} \lambda v'.applyCast(v', c_2)$
where

$$\begin{array}{rcl} \mathtt{succ}\ v \underline{\gg} f & = & f(v) \\ \mathtt{fail}\ l \underline{\gg} f & = & \mathtt{fail}\ l \end{array}$$

(3) If $v : T_1$ and $T_1 \not\sim T_2$, then $applyCast(v, cast(T_1, l, T_2)) = \mathtt{fail}\ l$

(4) If $v : \star$, then $applyCast(v, cast(\star, l, \star)) = \mathtt{succ}\ v$
(5) If $v : P$, then $applyCast(\mathtt{Dyn}_P(v), cast(\star, l, P')) = applyCast(v, cast(P, l, P'))$
(6) If $v : P$, then $applyCast(v, cast(P, l, \star)) = \mathtt{succ}\ (\mathtt{Dyn}_P(v))$
(7) If $v : \mathtt{Unit}$, then $applyCast(v, cast(\mathtt{Unit}, l, \mathtt{Unit})) = \mathtt{succ}\ v$
(8) If $(\langle \lambda x.\, e, c_1, c_2, \mathcal{E}\rangle) : T_1 \to T_2$, then
$applyCast(\langle \lambda x.\, e, c_1, c_2, \mathcal{E}\rangle, cast(T_1 \to T_2, l, T_3 \to T_4))$
$= \mathtt{succ}\ (\langle \lambda x.\, e, seq(cast(T_3, l, T_1), c_1), seq(c_2, cast(T_2, l, T_4)), \mathcal{E}\rangle)$
(9) If $(\mathtt{cons}\ v_1\langle c_1\rangle\ v_2\langle c_2\rangle) : T_1 \times T_2$, then
$applyCast(\mathtt{cons}\ v_1\langle c_1\rangle\ v_2\langle c_2\rangle, cast(T_1 \times T_2, l, T_3 \times T_4))$
$= \mathtt{succ}\ (\mathtt{cons}\ v_1\langle seq(c_1, cast(T_1, l, T_3))\rangle\ v_2\langle seq(c_2, cast(T_2, l, T_4))\rangle)$
(10) If $(\mathtt{inl}\ v\langle c\rangle) : T_1 + T_2$, then
$applyCast(\mathtt{inl}\ v\langle c\rangle, cast(T_1 + T_2, l, T_3 + T_4)) = \mathtt{succ}\ (\mathtt{inl}\ v\langle seq(c, cast(T_1, l, T_3))\rangle)$
(11) If $(\mathtt{inr}\ v\langle c\rangle) : T_1 + T_2$, then
$applyCast(\mathtt{inr}\ v\langle c\rangle, cast(T_1 + T_2, l, T_3 + T_4)) = \mathtt{succ}\ (\mathtt{inr}\ v\langle seq(c, cast(T_2, l, T_4))\rangle)$

4.2 A Space-efficient CEK Machine

$\mathcal{S}(C)$ is a space-efficient CEK machine parameterized over the Cast ADT (C). The key differences between $\mathcal{S}(C)$ and $\mathcal{X}$ are the places where casts can accumulate at runtime, that is, values and continuations. The differences regarding values was discussed above (Definition 4.1). We define contiuations below, with κ ranging over continuations and k ranging over pre-continuations. Pre-continuations are like the continuations in $\mathcal{X}$, but there is no constructor for casts. A continuation is now a pre-continuation prefixed with a cast.

Definition 4.4 (Continuations for the $\mathcal{S}(C)$ machine).

$$\begin{array}{llll}
\text{Continuation} & \kappa & ::= & [\square\langle c\rangle]k \\
\text{Pre-continuations} & k & ::= & \mathtt{stop} \mid [\mathtt{cons}^{T\times T}\ \square\ \langle e, \mathcal{E}\rangle]\kappa \mid [\mathtt{cons}^{T\times T}\ v\ \square]\kappa \mid [\mathtt{inl}^{T+T}\ \square]\kappa \\
 & & \mid & [\mathtt{inr}^{T+T}\ \square]\kappa \mid [\square\ \langle e, \mathcal{E}\rangle]\kappa \mid [v\ \square]\kappa \mid [\mathtt{fst}\ \square]\kappa \mid [\mathtt{snd}\ \square]\kappa \\
 & & \mid & [\mathtt{case}\ \square\ \langle e, \mathcal{E}\rangle\ \langle e, \mathcal{E}\rangle]\kappa \mid [\mathtt{case}\ v\ \square\ \langle e, \mathcal{E}\rangle]\kappa \mid [\mathtt{case}\ v\ v\ \square]\kappa
\end{array}$$

Continuations in $\mathcal{D}$ have zero or more casts at the top. In $\mathcal{S}(C)$, however, every continuation has exactly one cast at the top. Continuations in $\mathcal{X}$ that have no casts at the top correspond to continuations in $\mathcal{S}(C)$ whose casts are identities. Continuations in $\mathcal{X}$ that have many casts at the top correspond to continuations in $\mathcal{S}(C)$ where those casts are composed by $seq(c, c)$.

Fig. 11 defines the transition relation $s \longmapsto_{\mathcal{S}(C)} s$. They rely on functions provided by C to work with casts. When evaluating a cast expression, the machine extends the continuation with cast $cast(T_1, l, T_2)$. The function $ext(c, \kappa)$ composes c with the cast at the top of k by calling seq. To construct a function, the machine fills the casts with ids. To return a value to a continuation, the machine first applies the top cast to the value. If the application fails, the machine halts with the blame label from the failure. Otherwise, the machine handles the pre-continuation with $cont$. When the machine constructs a pair, a left injection, or a right injection, it fills the casts with id as well, just like how it did for functions. To apply a function, the machine applies the domain cast c_1 to the operand v_2, extends the continuation with the codomain cast c_2, and evaluates the function body. To take out the first part of a pair, the machine returns the first value to a continuation extended with the first cast. Taking out the second part of a pair is similar. To case split a variant, the machine first looks at the variant value. If the variant is a left injection, the machine composes the cast in the left injection with the domain cast of the first continuation function, then move to a state that will apply the new function to the value in the left injection. The case for right injections is similar.

Reflexive transitive closure of reduction ($s \longmapsto^*_{\mathcal{S}(C)} s$) and evaluation ($eval_{\mathcal{S}(C)}(e) = o$) are the same as for $\mathcal{X}$.

$$\text{Build continuation } \boxed{\mathit{wrap}(k) = \kappa}$$

$$\mathit{wrap}(k) \quad = \quad [\Box\langle \mathit{id}(T_1)\rangle]k \quad \text{where } k : T_1 \Longrightarrow T_2$$

$$\text{Extend continuation } \boxed{\mathit{ext}(c, \kappa) = \kappa}$$

$$\mathit{ext}(c_1, [\Box\langle c_2\rangle]k) \quad = \quad [\Box\langle \mathit{seq}(c_1, c_2)\rangle]k$$

$$\text{Transition } \boxed{s \longmapsto_{\mathcal{S}(C)} s}$$

$$\begin{array}{rcl}
\langle e\langle T_1 \Rightarrow^l T_2\rangle, \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{S}(C)} & \langle e, \mathcal{E}, \mathit{ext}(\mathit{cast}(T_1, l, T_2), \kappa)\rangle \\
\langle x, \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{S}(C)} & \langle \mathcal{E}(x), \kappa\rangle \\
\langle \mathtt{unit}, \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{S}(C)} & \langle \mathtt{unit}, \kappa\rangle \\
\langle \lambda^{T_1 \to T_2} x.e, \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{S}(C)} & \langle\langle \lambda x.\, e, \mathit{id}(T_1), \mathit{id}(T_2), \mathcal{E}\rangle, \kappa\rangle \\
\langle \mathtt{cons}^{T_1 \times T_2}\ e_1\ e_2, \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{S}(C)} & \langle e_1, \mathcal{E}, \mathit{wrap}([\mathtt{cons}^{T_1 \times T_2}\ \Box\ \langle e_2, \mathcal{E}\rangle]\kappa)\rangle \\
\langle \mathtt{inl}^{T_1 + T_2}\ e, \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{S}(C)} & \langle e, \mathcal{E}, \mathit{wrap}([\mathtt{inl}^{T_1 + T_2}\ \Box]\kappa)\rangle \\
\langle \mathtt{inr}^{T_1 + T_2}\ e, \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{S}(C)} & \langle e, \mathcal{E}, \mathit{wrap}([\mathtt{inr}^{T_1 + T_2}\ \Box]\kappa)\rangle \\
\langle (e_1\ e_2), \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{S}(C)} & \langle e_1, \mathcal{E}, \mathit{wrap}([\Box\ \langle e_2, \mathcal{E}\rangle]\kappa)\rangle \\
\langle \mathtt{fst}\ e), \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{S}(C)} & \langle e, \mathcal{E}, \mathit{wrap}([\mathtt{fst}\ \Box]\kappa)\rangle \\
\langle \mathtt{snd}\ e, \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{S}(C)} & \langle e, \mathcal{E}, \mathit{wrap}([\mathtt{snd}\ \Box]\kappa)\rangle \\
\langle \mathtt{case}\ e_1\ e_2\ e_3, \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{S}(C)} & \langle e_1, \mathcal{E}, \mathit{wrap}([\mathtt{case}\ \Box\ \langle e_2, \mathcal{E}\rangle\ \langle e_3, \mathcal{E}\rangle]\kappa)\rangle \\
\langle \mathtt{blame}\ l, \mathcal{E}, \kappa\rangle & \longmapsto_{\mathcal{S}(C)} & \mathtt{Halt}\ (\mathtt{blame}\ l) \\
\langle v, [\Box\langle c\rangle]k\rangle & \longmapsto_{\mathcal{S}(C)} & \begin{cases} \mathit{cont}(v', k) & \text{if } \mathit{applyCast}(v, c) = \mathtt{succ}\ v' \\ \mathtt{Halt}\ (\mathtt{blame}\ l) & \text{if } \mathit{applyCast}(v, c) = \mathtt{fail}\ l \end{cases}
\end{array}$$

$$\text{Apply continuation } \boxed{\mathit{cont}(k, v) = s}$$

$$\begin{array}{rcl}
\mathit{cont}([\mathtt{cons}^{T_1 \times T_2}\ \Box\ \langle e_2, \mathcal{E}\rangle]\kappa, v_1) & = & \langle e_2, \mathcal{E}, \mathit{wrap}([\mathtt{cons}^{T_1 \times T_2}\ v_1\ \Box]\kappa)\rangle \\
\mathit{cont}([\mathtt{cons}^{T_1 \times T_2}\ v_1\ \Box]\kappa, v_2) & = & \langle \mathtt{cons}\ v_1\langle \mathit{id}(T_1)\rangle\ v_2\langle \mathit{id}(T_2)\rangle, \kappa\rangle \\
\mathit{cont}([\mathtt{inl}^{T_1 + T_2}\ \Box]\kappa, v) & = & \langle \mathtt{inl}\ v\langle \mathit{id}(T_1)\rangle, \kappa\rangle \\
\mathit{cont}([\mathtt{inr}^{T_1 + T_2}\ \Box]\kappa, v) & = & \langle \mathtt{inr}\ v\langle \mathit{id}(T_2)\rangle, \kappa\rangle \\
\mathit{cont}([\Box\ \langle e_2, \mathcal{E}\rangle]\kappa, v_1) & = & \langle e_2, \mathcal{E}, \mathit{wrap}([v_1\ \Box]\kappa)\rangle \\
\mathit{cont}([(\langle \lambda x.\, e, c_1, c_2, \mathcal{E}\rangle)\ \Box]\kappa, v_2) & = & \begin{cases} \langle e, \mathcal{E}[x := v'], \mathit{ext}(c_2, \kappa)\rangle \\ \text{if } \mathit{applyCast}(v_2, c_1) = \mathtt{succ}\ v' \\ \mathtt{Halt}\ l \\ \text{if } \mathit{applyCast}(v_2, c_1) = \mathtt{fail}\ l \end{cases} \\
\mathit{cont}([\mathtt{fst}\ \Box]\kappa, \mathtt{cons}\ v_1\langle c_1\rangle\ v_2\langle c_2\rangle) & = & \langle v_1, \mathit{ext}(c_1, \kappa)\rangle \\
\mathit{cont}([\mathtt{snd}\ \Box]\kappa, \mathtt{cons}\ v_1\langle c_1\rangle\ v_2\langle c_2\rangle) & = & \langle v_2, \mathit{ext}(c_2, \kappa)\rangle \\
\mathit{cont}([\mathtt{case}\ \Box\ \langle e_2, \mathcal{E}\rangle\ \langle e_3, \mathcal{E}\rangle]\kappa, v_1) & = & \langle e_2, \mathcal{E}, \mathit{wrap}([\mathtt{case}\ v_1\ \Box\ \langle e_3, \mathcal{E}\rangle]\kappa)\rangle \\
\mathit{cont}([\mathtt{case}\ v_1\ \Box\ \langle e_3, \mathcal{E}\rangle]\kappa, v_2) & = & \langle e_3, \mathcal{E}, \mathit{wrap}([\mathtt{case}\ v_1\ v_2\ \Box]\kappa)\rangle \\
\mathit{cont}([\mathtt{case}\ (\mathtt{inl}\ v\langle c\rangle)\ (\langle \lambda x.\, e, c_1, c_2, \mathcal{E}\rangle)\ \Box]\kappa, v_3) & = & \langle v, \mathit{wrap}([(\langle \lambda x.\, e, \mathit{seq}(c, c_1), c_2, \mathcal{E}\rangle)\ \Box]\kappa)\rangle \\
\mathit{cont}([\mathtt{case}\ (\mathtt{inr}\ v\langle c\rangle)\ v_2\ \Box]\kappa, (\langle \lambda x.\, e, c_1, c_2, \mathcal{E}\rangle)) & = & \langle v, \mathit{wrap}([(\langle \lambda x.\, e, \mathit{seq}(c, c_1), c_2, \mathcal{E}\rangle)\ \Box]\kappa)\rangle \\
\mathit{cont}(\mathtt{stop}, v) & = & \mathtt{Halt}\ \mathit{observe}(v)
\end{array}$$

Fig. 11. Space-efficient CEK machine$\mathcal{S}(C)$

Proposition 4.5 ($\mathcal{S}(C)$ is deterministic). *If $s_1 \longmapsto_{\mathcal{S}(C)} s_2$ and $s_1 \longmapsto_{\mathcal{S}(C)} s_3$ then $s_2 = s_3$.*

4.3 $\mathcal{S}(C)$ Is Equivalent to $\mathcal{D}$, Provided C Is an Instance of the Lazy D ADT

In this section, we prove that for all C, if C is a Lazy D Cast ADT, then

$$eval_{\mathcal{S}(C)}(e) = o \text{ if and only if } eval_{\mathcal{D}}(e) = o$$

The main work of the proof is in a lemma that establishes a weak bisimulation between $\mathcal{S}(C)$ and $\mathcal{D}$. The bisimulation goes more smoothly if we require two more properties of the cast representation: that the casts form a monoid (the *id* and *seq* operators) and that the cast constructor is equivalent to the identity operator when applied to an identical source and target type. However, the final theorem does not require these two properties because we can prove 1) that $\mathcal{S}(C_1)$ and $\mathcal{S}(C_2)$ are equivalent for any two Lazy D Cast ADTs C_1 and C_2 (Proposition 4.12), and 2) there is an instance of the Lazy D Cast ADT, named L, that is a monoid and satisfies the two extra properties, and is therefore equivalent to $\mathcal{D}$ (Lemma 4.14). So by transitivity, for any Lazy D Cast ADT C we have $eval_{\mathcal{S}(C)}(e) = o$ if and only if $eval_{\mathcal{D}}(e) = o$.

4.3.1 Weak Bisimulation between $\mathcal{S}(C)$ and $\mathcal{D}$. We shall prove that if an instance of Cast ADT C is Lazy D, is a monoid, and if $cast(T, l, T) = id(T)$, then there is a weak bisimulation between $\mathcal{S}(C)$ and $\mathcal{D}$. We define monoid as follows.

Definition 4.6 (Monoid). A Cast ADT is a monoid if for all $c_1 : T_1 \Longrightarrow T_2$, $c_2 : T_2 \Longrightarrow T_3$, and $c_3 : T_3 \Longrightarrow T_4$,

(1) $seq(id(T_1), c_1) = c_1$,
(2) $seq(c_1, id(T_2)) = c_1$, and
(3) $seq(seq(c_1, c_2), c_3) = seq(c_1, seq(c_2, c_3))$.

Fig. 12 defines the bisimulation relation $s_1 \approx s_2$, where $s_1 \in \mathcal{S}(C)$ and $s_2 \in \mathcal{D}$. It is the smallest congruence relation that also includes the pairs induced by the rules in Fig. 12. One nice aspect of using abstract machines for this proof is that the expressions in the machine are simply related by syntactic equality. The bisimulation relation is mutually defined on continuations, written $\kappa \approx \kappa : T$, where T is their source type, and values, written $v \approx v : T$, where T is their type. If $k \approx \kappa : T$, adding an identity cast on top of k gives a related continuation. This handles the way in which $\mathcal{S}(C)$ uses *wrap* to push identity coercions onto continuations whereas $\mathcal{D}$ does not. We remark that this bisimulation relation is unusual in that it uses some ADT operations in its definition, such as *id*, *cast*, and *seq*, which is necessary because $\mathcal{S}(C)$ and the bisimulation is parameterized over the Cast ADT. If $[\square\langle c\rangle]k_1 \approx \kappa_2 : T$, extending the right-hand side with a cast $T_1 \Rightarrow^l T_2$ and sequencing $cast(T_1, l, T_2)$ with c on the left-hand side, produces related continuations. This takes care of the difference between the transition rules for $e\langle T_1 \Rightarrow^l T_2\rangle$ on the two machines.

The bisimulation relation for values is designed to make Lemma 4.7 true, that is, if $v_1 \approx v_2$, then

$$applyCast_{\mathcal{S}(C)}(v_1, cast(T_1, l, T_2)) \approx applyCast_{\mathcal{D}}(v_2, T_1 \Rightarrow^l T_2)$$

So if two values of type I are related, injecting them to $\star$ gives related values. To relate to a closure in $\mathcal{D}$, the closure in $\mathcal{S}(C)$ must have identities as its casts. If a closure on the left is related to a value, we can apply equivalent casts on both sides to obtain related values. The rules for products and sums are similar. Lemma 4.7 is used in our bisimulation proof to handle the transitions that apply casts to values.

Lemma 4.7 (*applyCast* Preserves Bisimulation). *Assume C implements Lazy D Cast ADT, v_1 is a value in $\mathcal{S}(C)$, v_2 is a value in $\mathcal{D}$, and $v_1 \approx v_2 : T_1$,*

$$applyCast_{\mathcal{S}(C)}(v_1, cast(T_1, l, T_2)) \approx applyCast_{\mathcal{D}}(v_2, T_1 \Rightarrow^l T_2)$$

$$\boxed{\Gamma \vdash \mathcal{E} \approx \mathcal{E}}$$

$$\frac{}{\emptyset \vdash \emptyset \approx \emptyset} \qquad \frac{\begin{array}{c}\Gamma \vdash \mathcal{E}_1 \approx \mathcal{E}_2 \\ v_1 \approx v_2 : T\end{array}}{\Gamma, x : T \vdash \mathcal{E}_1[x := v_1] \approx \mathcal{E}_2[x := v_2]}$$

$$\boxed{s \approx s}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : T \\ \Gamma \vdash \mathcal{E}_1 \approx \mathcal{E}_2 \\ \kappa_1 \approx \kappa_2 : T\end{array}}{\langle e, \mathcal{E}_1, \kappa_1 \rangle \approx \langle e, \mathcal{E}_2, \kappa_2 \rangle} \qquad \frac{}{\texttt{Halt}\ o \approx \texttt{Halt}\ o}$$

$$\boxed{\kappa \approx \kappa : T}$$

$$\frac{k \approx \kappa : T}{[\square\langle id(T)\rangle]k \approx \kappa} \qquad \frac{[\square\langle c \rangle]k_1 \approx \kappa_2 : T_2}{[\square\langle seq(cast(T_1, l, T_2), c)\rangle]k_1 \approx [\square\langle T_1 \Rightarrow^l T_2 \rangle]\kappa_2}$$

$$\boxed{v \approx v : T}$$

$$\frac{v_1 \approx v_2 : I}{\texttt{Dyn}_I(v_1) \approx v_2\langle I \Rightarrow^l \star \rangle : \star}$$

$$\frac{}{\texttt{unit} \approx \texttt{unit}\langle \texttt{Unit} \Rightarrow^l \texttt{Unit} \rangle : \texttt{Unit}}$$

$$\frac{\begin{array}{c}\Gamma, x : T_1 \vdash e : T_2 \\ \Gamma \vdash \mathcal{E}_1 \approx \mathcal{E}_2\end{array}}{\langle \lambda x.\, e, id(T_1), id(T_2), \mathcal{E}_1 \rangle \approx \langle \lambda x. e, \mathcal{E}_2 \rangle : T_1 \to T_2}$$

$$\frac{\langle \lambda x.\, e, c_1, c_2, \mathcal{E} \rangle \approx v_2 : T_1 \to T_2}{\langle \lambda x.\, e, c_1', c_2', \mathcal{E} \rangle \approx v_2\langle T_1 \to T_2 \Rightarrow^l T_3 \to T_4 \rangle : T_3 \to T_4} \quad \begin{array}{l} c_1' = seq(cast(T_3, l, T_1), c_1) \\ c_2' = seq(c_2, cast(T_2, l, T_4)) \end{array}$$

$$\frac{v_1 \approx v_2 : T_1 \qquad v_3 \approx v_4 : T_2}{\texttt{cons}\ v_1\langle id(T_1)\rangle\ v_3\langle id(T_2)\rangle \approx \texttt{cons}\ v_2\ v_4 : T_1 \times T_2}$$

$$\frac{\texttt{cons}\ v_1\langle c_1 \rangle\ v_3\langle c_2 \rangle \approx v_2 : T_1 \times T_2}{\texttt{cons}\ v_1\langle c_1' \rangle\ v_3\langle c_2' \rangle \approx v_2\langle T_1 \times T_2 \Rightarrow^l T_3 \times T_4 \rangle : T_3 \times T_4} \quad \begin{array}{l} c_1' = seq(c_1, cast(T_1, l, T_3)) \\ c_2' = seq(c_2, cast(T_2, l, T_4)) \end{array}$$

$$\frac{v_1 \approx v_2 : T_1}{\texttt{inl}\ v_1\langle id(T_1)\rangle \approx \texttt{inl}\ v_2 : T_1 + T_2} \qquad \frac{\texttt{inl}\ v_1\langle c \rangle \approx v_2 : T_1 + T_2 \qquad c' = seq(c, cast(T_1, l, T_3))}{\texttt{inl}\ v_1\langle c' \rangle \approx v_2\langle T_1 + T_2 \Rightarrow^l T_3 + T_4 \rangle : T_3 + T_4}$$

$$\frac{v_1 \approx v_2 : T_2}{\texttt{inr}\ v_1\langle id(T_2)\rangle \approx \texttt{inr}\ v_2 : T_1 + T_2} \qquad \frac{\texttt{inr}\ v_1\langle c \rangle \approx v_2 : T_1 + T_2 \qquad c' = seq(c, cast(T_2, l, T_4))}{\texttt{inr}\ v_1\langle c' \rangle \approx v_2\langle T_1 + T_2 \Rightarrow^l T_3 + T_4 \rangle : T_3 + T_4}$$

Fig. 12. Bisimulation between $\mathcal{S}(C)$ and $\mathcal{D}$

Proof. Case splitting whether T_1 is shallowly consistent with T_2. If yes, case splitting how they are shallowly consistent. Then apply properties (3) - (11) of Lazy D Cast when applicable. □

We now come to the main lemma, the weak bisimulation between $\mathcal{S}(C)$ and $\mathcal{D}$.

Lemma 4.8 (Weak Bisimulation between $\mathcal{S}(C)$ and $\mathcal{D}$). *If C implements Lazy D Cast ADT and C is a monoid and $cast(T,l,T) = id(T)$ and $s_1 \approx s_2$ then either*

(1) $s_1 = \texttt{Halt}\ o$ *and* $s_2 = \texttt{Halt}\ o$ *for some* o*, or*
(2) $s_1 \longmapsto^+_{\mathcal{S}(C)} s_3$ *and* $s_2 \longmapsto^+_{\mathcal{D}} s_4$ *and* $s_3 \approx s_4$ *for some* s_3 *and* s_4

Proof. We proceed by case analysis on $s_1 \approx s_2$.

Case $\dfrac{}{\texttt{Halt}\ o \approx \texttt{Halt}\ o}$ This is case (1).

Case $\dfrac{\Gamma \vdash e : T \quad \Gamma \vdash \mathcal{E} \approx \mathcal{E}' \quad \kappa \approx \kappa' : T}{\langle e, \mathcal{E}, \kappa\rangle \approx \langle e, \mathcal{E}', \kappa'\rangle}$ This is case (2). By case analysis on e.

Case $\dfrac{v \approx v' : T \quad \kappa \approx \kappa' : T}{\langle v, \kappa\rangle \approx \langle v', \kappa'\rangle}$ This is case (2). In this case we have the following reduction in $\mathcal{S}(C)$.

$$\begin{array}{rcl} \langle v, \kappa\rangle & = & \langle v, [\square\langle c\rangle]k\rangle \\ & \longmapsto_{\mathcal{S}(C)} & \begin{cases} cont(v_1, k) & \text{if } applyCast(v,c) = \texttt{succ}\ v_1 \\ \texttt{Halt (blame } l\texttt{)} & \text{if } applyCast(v,c) = \texttt{fail}\ l \end{cases} \end{array}$$

We proceed by induction on $[\square\langle c\rangle]k \approx \kappa' : T$.

Subcase $\dfrac{k \approx \kappa' : T}{[\square\langle id(T)\rangle]k \approx \kappa'}$ In this case, from the property (1) of Lazy D Cast ADT (Definition 4.3)

$$applyCast(v,c) = applyCast(v, id(T)) = \texttt{succ}\ v$$

Thus,

$$\langle v, \kappa\rangle = \langle v, [\square\langle id(T)\rangle]k\rangle \longmapsto_{\mathcal{S}(C)} cont(v,k)$$

We will show shortly that if $v \approx v' : T$ and $k \approx \kappa' : T$ then there exists an s' such that $cont(v,k) \approx s'$ and $\langle v', \kappa'\rangle \longmapsto_{\mathcal{D}} s'$.

Subcase $\dfrac{[\square\langle c_1\rangle]k_1 \approx \kappa_1' : T_2}{[\square\langle seq(cast(T_1,l,T_2),c_1)\rangle]k_1 \approx [\square\langle T_1 \Rightarrow^l T_2\rangle]\kappa_1'}$ In this case, from the property (2) of Lazy D Cast ADT (Definition 4.3)

$$\begin{array}{rcl} applyCast(v,c) & = & applyCast(v, seq(cast(T_1,l,T_2),c_1)) \\ & = & applyCast(v, cast(T_1,l,T_2)) \gg \lambda v.applyCast(v,c_1) \end{array}$$

In the $\mathcal{D}$ machine, we have the following reduction,

$$\langle v', k'\rangle = \langle v', [\square\langle T_1 \Rightarrow^l T_2\rangle]\kappa_1'\rangle \longmapsto_{\mathcal{D}} \begin{cases} \langle v_1', \kappa_1'\rangle & \text{if } applyCast(v', T_1 \Rightarrow^l T_2) = \texttt{succ}\ v_1' \\ \texttt{Halt (blame } l\texttt{)} & \text{if } applyCast(v', T_1 \Rightarrow^l T_2) = \texttt{fail}\ l \end{cases}$$

By Lemma 4.7, $applyCast(v, cast(T_1, l, T_2)) \approx applyCast(v', T_1 \Rightarrow^l T_2)$. When both cast applications fail, both machines transition to halting states with the same blame label. When both cast applications succeed, we apply the induction hypothesis.

Now we prove the missing part: if $v \approx v' : T$ and $k \approx \kappa' : T$ then there exists an s' such that $cont(v, k) \approx s'$ and $\langle v', \kappa' \rangle \longmapsto_{\mathcal{D}} s'$. We proceed by case analysis on $k \approx \kappa' : T$.

Many cases are trivial. The interesting cases are eliminations (e.g. function applications, pair projections). Due to space limitation, we only illustrate the general idea with function applications:

$$\frac{v_1 \approx v_1' : T_1 \to T_2 \qquad \kappa_1 \approx \kappa_1' : T_2}{[v_1\ \square]\kappa_1 \approx [v_1'\ \square]\kappa_1'}$$

We proceed by induction on the $v_1 \approx v_1' : T_1 \to T_2$.

Case $$\frac{\Gamma, x : T_1 \vdash e : T_2 \qquad \Gamma \vdash \mathcal{E} \approx \mathcal{E}'}{\langle \lambda x.\, e, id(T_1), id(T_2), \mathcal{E} \rangle \approx \langle \lambda x.e, \mathcal{E}' \rangle : T_1 \to T_2}$$

$$\begin{aligned} cont(v, k) &= cont(v, [\langle \lambda x.\, e, id(T_1), id(T_2), \mathcal{E} \rangle\ \square]\kappa_1) \\ &= \langle e, \mathcal{E}[x := v], ext(id(T_2), \kappa_1) \rangle \\ &= \langle e, \mathcal{E}[x := v], \kappa_1 \rangle \end{aligned}$$

The first step is substitution of variables. The second step is by the property (1) of Definition 4.3. And the third and final step is by the property (1) of Definition 4.6.

$$\begin{aligned} \langle v', \kappa' \rangle &= \langle v', [\langle \lambda x.e, \mathcal{E}' \rangle\ \square]\kappa_1' \rangle \\ &\longmapsto_{\mathcal{D}} \langle e, \mathcal{E}'[x := v'], \kappa_1' \rangle \end{aligned}$$

Case $$\frac{\langle \lambda x.\, e, c_3, c_4, \mathcal{E} \rangle \approx v_2' : T_1 \to T_2}{\langle \lambda x.\, e, c_1, c_2, \mathcal{E} \rangle \approx v_2' \langle T_1 \to T_2 \Rightarrow^l T_3 \to T_4 \rangle : T_3 \to T_4} \quad \begin{aligned} c_1 &= seq(cast(T_3, l, T_1), c_3) \\ c_2 &= seq(c_4, cast(T_2, l, T_4)) \end{aligned}$$

$$\begin{aligned} cont(v, k) &= cont(v, [\langle \lambda x.\, e, c_1, c_2, \mathcal{E} \rangle\ \square]\kappa_1) \\ &= \begin{cases} \langle e, \mathcal{E}[x := v_1], ext(c_2, \kappa) \rangle & \text{if } applyCast(v, c_1) = \texttt{succ}\ v_1 \\ \texttt{Halt}\ l & \text{if } applyCast(v, c_1) = \texttt{fail}\ l \end{cases} \end{aligned}$$

By property (3) of Monoid (Definition 4.6), we have

$$ext(c_2, \kappa) = ext(c_4, ext(cast(T_2, l, T_4), \kappa))$$

By property (2) of Lazy D Cast ADT (Definition 4.3), we have

$$applyCast(v, c_1) = applyCast(cast(T_3, l, T_1), v) \geqslant \lambda v.applyCast(c_3, v)$$

In the $\mathcal{D}$ side, we have the following reduction

$$\begin{aligned} \langle v', \kappa' \rangle &= \langle v', [v_2' \langle T_1 \to T_2 \Rightarrow^l T_3 \to T_4 \rangle\ \square]\kappa_1' \rangle \\ &\longmapsto_{\mathcal{D}} \langle v', [\square\langle T_3 \Rightarrow^l T_1 \rangle][v_2'\ \square][\square\langle T_2 \Rightarrow^l T_4 \rangle]\kappa_1' \rangle \\ &\longmapsto_{\mathcal{D}} \begin{cases} \langle v_3', [v_2'\ \square][\square\langle T_2 \Rightarrow^l T_4 \rangle]\kappa_1' \rangle & \text{if } applyCast(v', T_3 \Rightarrow^l T_1) = \texttt{succ}\ v_3' \\ \texttt{Halt}\ (\texttt{blame}\ l) & \text{if } applyCast(v', T_3 \Rightarrow^l T_1) = \texttt{fail}\ l \end{cases} \end{aligned}$$

By Lemma 4.7, $applyCast(v, cast(T_3, l, T_1)) \approx applyCast(v', T_3 \Rightarrow^l T_1)$. When both cast applications fail, both machines transition to halting states with the same blame label. When both cast applications succeed, we apply the induction hypothesis.

□

Corollary 4.9 (Weak Bisimulation between $\mathcal{S}(C)$ and $\mathcal{D}$, reformulated). *Assume C implements Lazy D Cast ADT and C is a monoid and* $cast(T, l, T) = id(T)$, $s_1 \approx s_1'$,

(1) *If* $s_1 \longmapsto_{\mathcal{S}(C)} s$ *for some s, then* $s \longmapsto^*_{\mathcal{S}(C)} s_2$, *and* $s_1' \longmapsto^+_{\mathcal{D}} s_2'$, $s_2 \approx s_2'$ *for some* s_2 *and* s_2'.
(2) *If* $s_1' \longmapsto_{\mathcal{D}} s'$ *for some* s', *then* $s' \longmapsto^*_{\mathcal{D}} s_2'$, *and* $s_1 \longmapsto^+_{\mathcal{S}(C)} s_2$, $s_2 \approx s_2'$ *for some* s_2 *and* s_2'.
(3) $s_1 = \texttt{Halt}\ o$ *if and only if* $s_1' = \texttt{Halt}\ o$

Proof. By applying Lemma 4.8 on $s_1 \approx s_1'$ and case analysis. □

Corollary 4.10 (Correctness of $\mathcal{S}(C)$). *Suppose C is an instance of the Cast ADT and the Lazy D ADT, C is a monoid, and* $cast(T, l, T) = id(T)$. *If* $\emptyset \vdash e : T$ *and* $o : T$, *then*

$$eval_{\mathcal{S}(C)}(e) = o \text{ if and only if } eval_{\mathcal{D}}(e) = o$$

Proof. By the definitions of *eval*, we need to show

$$\langle e, \emptyset, wrap(\texttt{stop})\rangle \longmapsto^*_{\mathcal{S}(C)} \texttt{Halt}\ o \text{ if and only if } \langle e, \emptyset, \texttt{stop}\rangle \longmapsto^*_{\mathcal{D}} \texttt{Halt}\ o$$

Obviously, the initial states are related in the bisimulation relation.

$$\langle e, \emptyset, wrap(\texttt{stop})\rangle \approx \langle e, \emptyset, \texttt{stop}\rangle$$

So we can generalize our goal, and prove that if $s_1 \approx s_1'$

$$s_1 \longmapsto^*_{\mathcal{S}(C)} \texttt{Halt}\ o \text{ if and only if } s_1' \longmapsto^*_{\mathcal{D}} \texttt{Halt}\ o$$

Let us prove left to right first. We proceed by induction on $s_1 \longmapsto^*_{\mathcal{S}(C)} \texttt{Halt}\ o$.

- $s_1 = \texttt{Halt}\ o$ implies $s_1' = \texttt{Halt}\ o$ by Lemma 4.8.
- If $s_1 \longmapsto_{\mathcal{S}(C)} s$ and $s \longmapsto^*_{\mathcal{S}(C)} \texttt{Halt}\ o$ for some s, by Lemma 4.8, there exists s_2 and s_2' such that $s_1 \longmapsto^+_{\mathcal{S}(C)} s_2$ and $s_1' \longmapsto^+_{\mathcal{D}} s_2'$ and $s_2 \approx s_2'$. The transition $s_1 \longmapsto^+_{\mathcal{S}(C)} s_2$ must be a prefix of $s_1 \longmapsto^+_{\mathcal{S}(C)} \texttt{Halt}\ o$ because $\mathcal{S}(C)$ is deterministic (Proposition 4.5) and halting states are stuck. So we can apply our induction hypothesis on $s_2 \approx s_2'$

The other direction is similar. □

4.3.2 $\mathcal{S}(C_1)$ *and* $\mathcal{S}(C_2)$ *are equivalent if* C_1 *and* C_2 *are Lazy D.* We prove the equivalence among two $\mathcal{S}(C)$ machines with a strong bisimulation. The bisimulation relation is the smallest congruence relation that also includes the rules below.

$$\frac{}{cast_1(T_1, l, T_2) \approx cast_2(T_1, l, T_2)} \qquad \frac{}{id_1(T) \approx id_2(T)} \qquad \frac{c_1 \approx c_2 \qquad c_3 \approx c_4}{seq_1(c_1, c_3) \approx seq_2(c_2, c_4)}$$

Lemma 4.11 (Strong Bisimulation among $\mathcal{S}(\cdot)$). *Assume* C_1 *and* C_2 *implements Lazy D Cast ADT* $s_1 \in \mathcal{S}(C_1)$, $s_2 \in \mathcal{S}(C_2)$, $s_1 \approx s_2$,

(1) *If* $s_1 = \texttt{Halt}\ o$ *then* $s_2 = \texttt{Halt}\ o$
(2) *If* $s_2 = \texttt{Halt}\ o$ *then* $s_1 = \texttt{Halt}\ o$
(3) *If* $s_1 \longmapsto_{\mathcal{S}(C_1)} s_3$ *then* $s_2 \longmapsto_{\mathcal{S}(C_2)} s_4$ *and* $s_3 \approx s_4$ *for some* s_4
(4) *If* $s_2 \longmapsto_{\mathcal{S}(C_2)} s_4$ *then* $s_1 \longmapsto_{\mathcal{S}(C_1)} s_3$ *and* $s_3 \approx s_4$ *for some* s_3

Proof. This proof is straightforward. The key ideas are undoing uses of *seq* with the property (2) of Lazy D Cast ADT, and handling all uses of *cast* with properties (3)-(11). □

Proposition 4.12 (Equivalence of Two Lazy D Cast ADTs). *Assume* C_1 *and* C_2 *implements Lazy D Cast ADT. If* $\emptyset \vdash e : T$ *and* $o : T$ *then*

$$eval_{\mathcal{S}(C_1)}(e) = o \text{ if and only if } eval_{\mathcal{S}(C_2)}(e) = o$$

Proof. Without loss of generality, assume $eval_{\mathcal{S}(C_1)}(e) = o$ and prove $eval_{\mathcal{S}(C_2)}(e) = o$. By the definition of $eval_{\mathcal{S}(C)}$, we know

$$\langle e, \emptyset, wrap(\texttt{stop})\rangle \longmapsto^*_{\mathcal{S}(C_1)} \texttt{Halt}\ o$$

By induction on this reduction sequence and Lemma 4.11, we have

$$\langle e, \emptyset, wrap(\texttt{stop})\rangle \longmapsto^*_{\mathcal{S}(C_2)} \texttt{Halt}\ o$$

that is, $eval_{\mathcal{S}(C_2)}(e) = o$ □

4.3.3 The L instance of the Lazy D Cast ADT. The L cast representation is defined in Fig. 13. Casts in L are lists of steps (s), where every step is like a cast in $\mathcal{D}$. The function *applyCast* applies steps from left to right. Steps are restricted by its typing rule such that its source and target are different. This restriction is necessary to show $cast(T, l, T) = id(T)$. With this representation of casts, the operator *id* produces an empty list of steps and *seq* appends two lists of steps.

Proposition 4.13 (L Properties).

(1) *L is an instance of the Lazy D ADT*
(2) *L is a monoid*
(3) $cast(T, l, T) = id(T)$

Proof. Part (1) is straightforward because property (1) and (2) are trivially true and because *appS* effectively repeats property (3) - (11). Part (2) is true because lists are known to be a monoid with respect to append. Part (3) follows immediately from the definition of $cast(T, l, T)$ □

Lemma 4.14 (L is Correct wrt. $\mathcal{D}$). *If $\emptyset \vdash e : T$ and $o : T$, then*

$$eval_{\mathcal{S}(L)}(e) = o \textit{ if and only if } eval_{\mathcal{D}}(e) = o$$

Proof. Immediately from Correctness of $\mathcal{S}(C)$ (Corollary 4.10) and Properties of L (Proposition 4.13). □

4.3.4 Correctness of $\mathcal{S}(C)$.

Theorem 4.15 ($\mathcal{S}(C)$ is Correct wrt. $\mathcal{D}$). *Suppose C is a Lazy D Cast ADT. If $\emptyset \vdash e : T$ and $o : T$ then*

$$eval_{\mathcal{S}(C)}(e) = o \textit{ if and only if } eval_{\mathcal{D}}(e) = o$$

Proof. Immediately from Equivalence of Lazy D Cast ADTs (Lemma 4.12) and L is Correct wrt. $\mathcal{D}$ (Lemma 4.14) □

5 CORRECTNESS PROOF OF LAZY D HYPERCOERCIONS

In this section, we prove Lazy D hypercoercions are correct. First, we define $applyCast(v, c)$ to make it an instance of Cast ADT, then prove that it is also Lazy D, and finally apply Theorem 4.15 to finish the proof.

Fig. 14 defines $applyCast(v, c)$ for Lazy D hypercoercions. Applying the identity cast for the dynamic type succeeds immediately. If the value is injected and the cast is not the identity cast, then *applyCast* projects the value and relies on *applyMiddle* to ensure that value is shallow consistent with target type of the head. Regardless of whether or not the value is projected, *applyCast* applies the middle, then apply the tail if the middle succeeds. We denote by $r \ggg f$ to mean that if r is $\texttt{succ}\ v$, the result is $f(v)$, otherwise, the result is the failure. In the definition of $applyMiddle(v, \ell, m)$, we generalize shallow-consistency to compare middles and values ($v \smile m$) in the obvious way. When a value is shallowly inconsistent from the middle, it is the result of projecting a value that was shallowly inconsistent with the expected type of the projection in the head. Thus, a blame label

Syntax

$$\begin{array}{llcl} \text{Casts} & c & ::= & [] \mid s :: c \\ \text{Steps} & s & ::= & T \Rightarrow^{l} T \end{array}$$

Steps Typing $\boxed{s : T \Longrightarrow T}$

$$\frac{T_1 \neq T_2}{T_1 \Rightarrow^{l} T_2 : T_1 \Longrightarrow T_2}$$

Casts typing $\boxed{c : T \Longrightarrow T}$

$$\frac{}{[] : T \Longrightarrow T} \qquad \frac{s : T_1 \Longrightarrow T_2 \quad c : T_2 \Longrightarrow T_3}{s :: c : T_1 \Longrightarrow T_3}$$

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

$$\begin{array}{rcl} seq([], c) & = & c \\ seq(s :: c', c) & = & s :: seq(c', c) \end{array}$$

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

$$id(T) = []$$

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

$$\begin{array}{rcll} cast(T_1, l, T_2) & = & [] & \text{if } T_1 = T_2 \\ cast(T_1, l, T_2) & = & T_1 \Rightarrow^{l} T_2 :: [] & \text{if } T_1 \neq T_2 \end{array}$$

$$\boxed{appS(v, s) = r}$$

$$\begin{array}{rcll} appS(v, \star \Rightarrow^{l} \star) & = & \mathsf{succ}\ v & \\ appS(v\langle P_1 \Rightarrow^{l_1} \star\rangle, \star \Rightarrow^{l_2} P_2) & = & \mathsf{succ}\ v & \text{if } P_1 = P_2 \\ appS(v\langle P_1 \Rightarrow^{l_1} \star\rangle, \star \Rightarrow^{l_2} P_2) & = & appS(v, P_1 \Rightarrow^{l_2} P_2) & \text{if } P_1 \neq P_2 \\ appS(v, P \Rightarrow^{l} \star) & = & \mathsf{succ}\ \mathsf{Dyn}_P(v) & \\ appS(v, \mathtt{Unit} \Rightarrow^{l} \mathtt{Unit}) & = & \mathsf{succ}\ v & \\ appS(\langle \lambda x.\, e, c_1, c_2, \mathcal{E}\rangle, T_1 \to T_2 \Rightarrow^{l} T_3 \to T_4) & = & \mathsf{succ}\ (\langle \lambda x.\, b, c_1', c_2', \mathcal{E}\rangle) & \\ & & \text{where } c_1' = seq(cast(T_3, l, T_1), c_1) & \\ & & \text{and } c_2' = seq(c_2, cast(T_2, l, T_4)) & \\ appS(\mathtt{cons}\ v_1\langle c_1\rangle\ v_2\langle c_2\rangle, T_1 \times T_2 \Rightarrow^{l} T_3 \times T_4) & = & \mathsf{succ}\ (\mathtt{cons}\ v_1\langle c_1'\rangle\ v_2\langle c_2'\rangle) & \\ & & \text{where } c_1' = seq(c_1, cast(T_1, l, T_3)) & \\ & & \text{and } c_2' = seq(c_2, cast(T_2, l, T_4)) & \\ appS(\mathtt{inl}\ v\langle c\rangle, T_1 + T_2 \Rightarrow^{l} T_3 + T_4) & = & \mathsf{succ}\ (\mathtt{inl}\ v\langle seq(c, cast(T_1, l, T_3))\rangle) & \\ appS(\mathtt{inr}\ v\langle c\rangle, T_1 + T_2 \Rightarrow^{l} T_3 + T_4) & = & \mathsf{succ}\ (\mathtt{inr}\ v\langle seq(c, cast(T_2, l, T_4))\rangle) & \\ appS(v, P_1 \Rightarrow^{l} P_2) & = & \mathtt{fail}\ l & \text{if } P_1 \not\sim P_2 \end{array}$$

$$\boxed{applyCast(v, c) = r}$$

$$\begin{array}{rcl} applyCast(v, []) & = & \mathsf{succ}\ v \\ applyCast(v, s :: c) & = & applyStep(v, s) \ggg \lambda v.applyCast(v, c) \end{array}$$

Fig. 13. The L Lazy D Cast ADT

$$\boxed{applyCast(v, c) = r}$$

$$\begin{array}{rcll}
applyCast(v, \mathtt{id}\star,) & = & \mathtt{succ}\ v & \\
applyCast(\mathtt{Dyn}_P(v), ?^l \overset{m}{\curvearrowright} t) & = & applyMiddle(v, l, m) \ \underline{\gg}\ \lambda v.applyTail(t, v) & \\
applyCast(v, \epsilon \overset{m}{\curvearrowright} t) & = & applyMiddle(v, \epsilon, m) \ \underline{\gg}\ \lambda v.applyTail(t, v) &
\end{array}$$

$$\boxed{applyMiddle(v, \ell, m) = r}$$

$$\begin{array}{rcll}
applyMiddle(\mathtt{unit}, \ell, \mathtt{Unit}) & = & \mathtt{succ}\ \mathtt{unit} & \\
applyMiddle(\langle \lambda x.\, e, c_1, c_2, \mathcal{E}\rangle, \ell, c_3 \to c_4) & = & \mathtt{succ}\ (\langle \lambda x.\, e, (c_3 \mathbin{⨟^\ell} c_1), (c_2 \mathbin{⨟^\ell} c_4), \mathcal{E}\rangle) & \\
applyMiddle(\mathtt{cons}\ v_1\langle c_1\rangle\ v_2\langle c_2\rangle, \ell, c_3 \times c_4) & = & \mathtt{succ}\ (\mathtt{cons}\ v_1\langle (c_1 \mathbin{⨟^\ell} c_3)\rangle\ v_2\langle (c_2 \mathbin{⨟^\ell} c_4)\rangle) & \\
applyMiddle(\mathtt{inl}\ v\langle c_1\rangle, \ell, c_3 + c_4) & = & \mathtt{succ}\ (\mathtt{inl}\ v\langle (c_1 \mathbin{⨟^\ell} c_3)\rangle) & \\
applyMiddle(\mathtt{inr}\ v\langle c_2\rangle, \ell, c_3 + c_4) & = & \mathtt{succ}\ (\mathtt{inr}\ v\langle (c_2 \mathbin{⨟^\ell} c_4)\rangle) & \\
applyMiddle(v, l, m) & = & \mathtt{fail}\ l & \text{if } v \not\prec m
\end{array}$$

$$\boxed{applyTail(v, t) = r}$$

$$\begin{array}{rcl}
applyTail(v, \bot^l) & = & \mathtt{fail}\ l \\
applyTail(v, \epsilon) & = & \mathtt{succ}\ v \\
applyTail(v, !) & = & \mathtt{succ}\ (\mathtt{Dyn}_P(v))
\end{array}$$

Fig. 14. Lazy D hypercoercion's *applyCast*

l must have been pass in as ℓ and is to blame. Finally if a failure in *applyMiddle* has not occurred then $applyTail(v, t)$ interprets the tail, t, to the appropriate cast of the value v.

Lemma 5.1 (Lazy D Hypercoercion is a Lazy D Cast ADT).

Proof. See the Agda proof at the following URL:
https://github.com/LuKC1024/hypercoercion-and-framework-wgt2020/tree/master/Proof □

Theorem 5.2 (Lazy D Hypercoercion Respect $\mathcal{D}$). *If $\emptyset \vdash e : T$ and $o : T$*

$$eval_{\mathcal{S}(H)}(e) = o \textit{ if and only if } eval_{\mathcal{D}}(e) = o$$

Proof. Immediately from that every Lazy D Cast ADT is correct (Theorem 4.15) and that Lazy D hypercoercion is a Lazy D Cast ADT (Lemma 5.1). Alternatively, from Lemma 4.10, Lemma 5.1, Proposition 3.1, and Proposition 3.2 □

6 CONCLUSION

In this paper, we presented a new cast representation, hypercoercions, in two flavors, Lazy D and Lazy UD. Hypercoercions have a structurally recursive composition operator and have a more compact memory representation in comparison to coercions in normal form.

We also present steps toward the first framework for proving the correctness of cast representations. The framework includes abstract data types for cast representations (Cast ADT and its refinement Lazy D Cast ADT), a parameterized abstract machine $\mathcal{S}(C)$, and a theorem which states that $\mathcal{S}(C)$ is equivalent to $\mathcal{D}$ when C is a Lazy D Cast ADT. We conjecture that Lazy D threesomes and Lazy D coercions in normal form are instances of the Lazy D Cast ADT.

Finally, we proved that Lazy D hypercoercions (H) is a Lazy D Cast ADT. By using our framework, we showed that it is a correct cast representation because $S(H)$ is equivalent to D.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Matthias Felleisen and Matthew Flatt. 2007. Programming Languages and Lambda Calculi. (July 2007).

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 *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13).* ACM, New York, NY, USA, 417–428. https://doi.org/10.1145/2500365.2500603

Álvaro García-Pérez, Pablo Nogueira, and Ilya Sergey. 2014. Deriving interpretations of the gradually-typed lambda calculus. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation.* ACM, 157–168.

Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. 2005. Fine-grained interoperability through mirrors and contracts. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications.* ACM Press, New York, NY, USA, 231–245.

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.

Kenneth Knowles and Cormac Flanagan. 2010. Hybrid type checking. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 32, 2 (2010), 6.

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.

Jacob Matthews and Robert Bruce Findler. 2007. Operational Semantics for Multi-Language Programs. In *The 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.*

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 *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15).* ACM, New York, NY, USA, 425–435. https://doi.org/10.1145/2737924.2737968

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 Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Vol. 6. 81–92.

Jeremy G. Siek and Philip Wadler. 2010. Threesomes, with and Without Blame. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10).* ACM, New York, NY, USA, 365–376. https://doi.org/10.1145/1706299.1706342

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: From Scripts to Programs. In *Dynamic Languages Symposium.*

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