JFP **31**, e30, 73 pages, 2021.

doi:10.1017/S0956796821000241

Parameterized cast calculi and reusable meta-theory for gradually typed lambda calculi

JEREMY G. SIEK AND TIANYU CHEN

Indiana University, Bloomington, IN 47405, USA
(*e-mails:* jsiek@indiana.edu, chen512@indiana.edu)

Abstract

The research on gradual typing has led to many variations on the Gradually Typed Lambda Calculus (GTLC) of Siek & Taha (2006) and its underlying cast calculus. For example, Wadler and Findler (2009) added blame tracking, Siek *et al*. (2009) investigated alternate cast evaluation strategies, and Herman *et al*. (2010) replaced casts with coercions for space efficiency. The meta-theory for the GTLC has also expanded beyond type safety to include blame safety (Tobin-Hochstadt & Felleisen, 2006), space consumption (Herman *et al*., 2010), and the gradual guarantees (Siek *et al*., 2015). These results have been proven for some variations of the GTLC but not others. Furthermore, researchers continue to develop variations on the GTLC, but establishing all of the meta-theory for new variations is time-consuming. This article identifies abstractions that capture similarities between many cast calculi in the form of two parameterized cast calculi, one for the purposes of language specification and the other to guide space-efficient implementations. The article then develops reusable meta-theory for these two calculi, proving type safety, blame safety, the gradual guarantees, and space consumption. Finally, the article instantiates this meta-theory for eight cast calculi including five from the literature and three new calculi. All of these definitions and theorems, including the two parameterized calculi, the reusable meta-theory, and the eight instantiations, are mechanized in Agda making extensive use of module parameters and dependent records to define the abstractions.

1 Introduction

The theory of gradual typing has grown at a fast pace, since the idea crystallized in the mid 2000s (Siek & Taha, 2006a; Tobin-Hochstadt & Felleisen, 2006; Matthews & Findler, 2007; Gronski *et al*., 2006). Researchers have discovered many choices regarding the design and formalization of gradually typed languages. For example, a language designer can choose between runtime casts that provide lazy, eager, or even partially eager semantics (Siek *et al*., 2009; García-Pérez *et al*., 2014). Alternatively, the designer might apply the methodology of Abstracting Gradual Typing (AGT) to derive the semantics (Garcia *et al*., 2016). When a runtime casts fails, there is the question of who to blame, using either the D or UD blame-tracking approaches (Siek *et al*., 2009). Furthermore, with the need to address the problems of space efficiency (Herman *et al*., 2010), one might choose to use threesomes (Siek & Wadler, 2010), supercoercions (Garcia, 2013), or coercions in one of several normal forms (Siek & Garcia, 2012; Siek *et al*., 2015a).

The last decade has also seen tremendous progress in the mechanization of programming language theory (Aydemir *et al.*, 2005). It has become routine for researchers to use proof assistants such as Coq (The Coq Dev. Team, 2004), Isabelle (Nipkow *et al.*, 2007), or Agda (Bove *et al.*, 2009) to verify the proofs of the meta-theory for a programming language. From the beginning, researchers in gradual typing used proof assistants to verify type safety (Siek & Taha, 2006c,b; Tobin-Hochstadt & Felleisen, 2008). They continue to mechanize the type soundness of new designs (Siek & Vitousek, 2013; Chung *et al.*, 2018) and to mechanize proofs of new properties such as open-world soundness and the gradual guarantee (Siek *et al.*, 2015b; Vitousek & Siek, 2016b; Xie *et al.*, 2018).

While machine-checked proofs provide the ultimate degree of trust, they come at a high development cost. With this in mind, it would be useful to reduce the cost by reusing definitions, theorems, and proofs about gradually typed languages. Agda provides particularly nice support for reuse: its combination of parameterized modules, dependent types, and more specifically, dependent records, provide a high degree of flexibility at a relatively low cost in complexity. For this reason, we choose to develop a new mechanization in Agda instead of building on prior mechanizations of gradual typing in other proof assistants.

The research on gradual typing has revealed subtle and complex interactions between gradual typing and other language features such as

- mutable state (Herman *et al.*, 2010; Siek *et al.*, 2015c; Toro & Tanter, 2020),
- subtyping (Siek & Taha, 2007; Garcia *et al.*, 2016; Chung *et al.*, 2018; Bañados Schwerter *et al.*, 2021),
- classes and objects (Takikawa *et al.*, 2012; Allende *et al.*, 2013; Vitek, 2016; Takikawa, 2016; Muehlboeck & Tate, 2017; Chung *et al.*, 2018),
- parametric polymorphism (Ahmed *et al.*, 2011, 2017; Toro *et al.*, 2019; New *et al.*, 2019),
- type inference (Siek & Vachharajani, 2008; Garcia & Cimini, 2015; Castagna *et al.*, 2019),
- set-theoretic types (Castagna & Lanvin, 2017),
- dependent types (Eremondi *et al.*, 2019; Lennon-Bertrand *et al.*, 2020),
- and many more.

This article starts at the beginning, focusing on a language with first-class functions and constants, that is, the Gradually Typed Lambda Calculus (GTLC) (Siek & Taha, 2006a), with the addition of pairs and sums whose presence helps to reveal some patterns that require abstraction.

The Gradually Typed Lambda Calculus. We present a formalization of the static semantics of the GTLC in Section 2, including the type system and definitions of important relations such as consistency and type precision. We present a proof of the static gradual guarantee: that changing type annotations to be less precise preserves typing. This is not the first mechanization of the static gradual guarantee for the GTLC (Siek *et al.*, 2015b); it is here for the sake of completeness.

The dynamic semantics of the GTLC is defined by translation to a cast calculus. We present that translation in Section 4 and prove that it preserves types and is monotone with respect to precision. There are many different cast calculi to choose from, with subtle

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

variations in semantics and dramatic variations in efficiency. This article identifies the similarities between many cast calculi in the form of the Parameterized Cast Calculus and the Space-Efficient Parameterized Cast Calculus. The former is for the purposes of language specification, whereas the later is meant as an guide to implementation. We discuss both in more detail below.

The Parameterized Cast Calculus. The first part of the article designs the Parameterized Cast Calculus. It is parameterized with respect to the cast representation and operations on casts. It can therefore model a range of cast calculi, from ones that represent a cast with a source type, target type, and a blame label such as the Blame Calculus (Wadler & Findler, 2009), to other calculi that represent casts using the Coercion Calculus of (Henglein, 1994). The article proves the following theorems about the Parameterized Cast Calculus, with mechanizations in Agda.

- type safety,
- blame-subtyping, and
- the dynamic gradual guarantee.

This article instantiates the Parameterized Cast Calculus to produce definitions and results for cast calculi in the literature as well as new cast calculi. We produce the type systems, reduction relations, and proofs of type safety and blame safety for all of the following systems.

1. The cast calculus of Siek & Taha (2006*a*) (Section 5.1).
2. A variant that classifies a function cast applied to a value as a value (Section 5.2).
3. The Blame Calculus λB of Siek *et al*. (2015*a*) (Section 5.3).
4. A coercion-based version of Siek and Taha's cast calculus (Section 5.4).
5. A lazy D coercion-based calculus (Siek *et al*., 2009) (Section 5.5).
6. The λC calculus of Siek *et al*. (2015*a*) (Section 5.6).

Our parameterized proof of the dynamic gradual guarantee was completed only recently; we have instantiated it on two variants of λB.

Scope of the Parameterized Cast Calculus. How many of the designs in the literature on gradually typed languages are instances of the Parameterized Cast Calculus? Although the constructive examples above give some evidence to its breadth, we do not yet have a complete answer to this question. However, we can make some informal statements about our intentions. The Parameterized Cast Calculus is designed to model sound gradual languages, and not unsound ones such as optional type systems (Bracha, 2004) that compile gradual programs to dynamically typed programs via type erasure (Bracha & Griswold, 1993; Chaudhuri; Maidl *et al*., 2014; Verlaguet & Menghrajani; Bierman *et al*., 2014; Greenman & Felleisen, 2018). The Parameterized Cast Calculus is only applicable to languages that use casts for runtime enforcement, and not other mechanisms such as the checks in transient semantics (Vitousek & Siek, 2016a; Vitousek *et al*., 2017; Greenman & Felleisen, 2018; Greenman & Migeed, 2018; Vitousek *et al*., 2019; Greenman, 2020). We expect that several, different, parameterized cast calculi are needed to capture the large design space present in the literature. We encourage other researchers to join us in this project

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

of developing reusable meta-theory for gradually typed languages. We conjecture that the Parameterized Cast Calculus, with some minor modifications, could model the intrinsic calculi used in the AGT methodology (Toro & Tanter, 2020).

An important consideration regarding the design of an abstraction such as the Parameterized Cast Calculus is where to draw the line between the generic versus the specific. The Parameterized Cast Calculus makes a natural choice regarding where to draw the line: at the representation of casts and the operations on them. However, there are two competing principles that influence the design:

1. The more generic the design, the more instances can be accommodated, which increases reuse.
2. The more generic the design, the less meta-theory can be carried out generically, which reduces reuse.

More experimentation with these design choices is needed to better understand the trade-offs. The Parameterized Cast Calculus of this article is one data point.

Space-efficient gradual typing. The second part of the article develops the Space-Efficient Parameterized Cast Calculus, which compresses sequences of casts to obtain space efficiency. To support this compression, the cast representation is required to provide a compose operator. We instantiate this calculus to produce results for a specific cast calculi in the literature as well as a new cast calculi. In particular, we produce definitions and proofs of space efficiency in Agda for

1. the λS calculus of Siek *et al*. (2015*a*) (Section 7.1), and
2. a new cast calculus based on hypercoercions (Lu *et al*., 2020) (Section 7.2).

Correctness of space-efficient calculi. Concurrently to the development of this article, and with an overlapping set of authors, Lu *et al*. (2020) investigate the semantics of parameterized cast calculi using abstract machines. They define two parameterized CEK machines (Felleisen & Friedman, 1986), one to define the semantics and the other to implement casts in a space-efficient manner. The main result is a parameterized proof in Agda of a weak bisimulation between the two machines, under the assumptions that both cast representations satisfy certain laws specific to the lazy D semantics (Lu, 2020). They instantiate this generic theorem to prove the correctness of hypercoercions. Generalizing this correctness result to include cast semantics other than lazy D is future work.

Mechanization in Agda. The language definitions and proofs are mechanized in Agda version 2.6.2 following the style in the textbook Programming Language Foundations in Agda (Wadler & Kokke, 2019) and is publicly available in the following github repository, at release version 2.1.

https://github.com/jsiek/gradual-typing-in-agda

This project depends on the Abstract Binding Trees library, at release version 1.0.

https://github.com/jsiek/abstract-binding-trees

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$$
\begin{array}{rcll}
b \in \mathcal{B} & ::= & \texttt{Nat} \mid \texttt{Bool} \mid \texttt{Int} \mid \texttt{Unit} \mid \bot & \text{base types} \\
a \in \mathcal{A} & ::= & b \mid ? & \text{atomic types} \\
A, B, C, D \in \mathcal{T} & ::= & a \mid A \to B \mid A \times B \mid A + B & \text{types} \\
\Gamma, \Delta & ::= & \emptyset \mid \Gamma \cdot A & \text{typing contexts}
\end{array}
$$

$\boxed{A \sim B}$

$$\texttt{UnkR}\sim[A] : \frac{}{A \sim ?} \qquad \texttt{UnkL}\sim[B] : \frac{}{? \sim B} \qquad \texttt{Base}\sim[b] : \frac{}{b \sim b}$$

$$\texttt{Fun}\sim : \frac{A' \sim A \qquad B \sim B'}{A \to B \sim A' \to B'}$$

$$\texttt{Pair}\sim : \frac{A \sim A' \qquad B \sim B'}{A \times B \sim A' \times B'} \qquad \texttt{Sum}\sim : \frac{A \sim A' \qquad B \sim B'}{A + B \sim A' + B'}$$

$\boxed{\bigsqcup : A \sim B \to \mathcal{T}}$

$$
\begin{aligned}
\bigsqcup(\texttt{UnkR}\sim[A]) &= A \\
\bigsqcup(\texttt{UnkL}\sim[B]) &= B \\
\bigsqcup(\texttt{Base}\sim[b]) &= b \\
\bigsqcup(\texttt{Fun}\sim\ d_1\ d_2) &= \bigsqcup(d_1) \to \bigsqcup(d_2) \\
\bigsqcup(\texttt{Pair}\sim\ d_1\ d_2) &= \bigsqcup(d_1) \times \bigsqcup(d_2) \\
\bigsqcup(\texttt{Sum}\sim\ d_1\ d_2) &= \bigsqcup(d_1) + \bigsqcup(d_2)
\end{aligned}
$$

$\boxed{A \triangleright B}$

$$\frac{}{A \to B \ \triangleright\ A \to B} \qquad \frac{}{? \ \triangleright\ ? \to ?}$$

$$\frac{}{A \times B \ \triangleright\ A \times B} \qquad \frac{}{? \ \triangleright\ ? \times ?}$$

$$\frac{}{A + B \ \triangleright\ A + B} \qquad \frac{}{? \ \triangleright\ ? + ?}$$

Fig. 1. Gradual types, typing contexts, consistency, join, and matching.

2 Gradually Typed Lambda Calculus

In this section, we formalize the static semantics of the GTLC of Siek & Taha (2006*a*) with the addition of products and sums. We use de Bruijn representation of variables and define the terms by leveraging the Abstract Binding Tree library. Their typing rules are defined as an Agda data type. The dynamic semantics of the GTLC is defined by translation to a cast calculus in Section 4.

Types. Figure 1 defines the set of types $\mathcal{T}$ of the GTLC, which includes function types, product types, sum types, and atomic types. The atomic types include the base types (natural numbers, Booleans, etc.) and the unknown type, written ? (aka. the dynamic type). As usual, the unknown type represents the absence of static type information. Figure 1 defines typing contexts Γ, which are sequences of types, that is, they map variables (represented by de Bruijn indices) to types.

Figure 1 defines the consistency relation at the heart of gradual typing. We say that two types are *consistent*, written $A \sim B$, if they are equal except in spots where either type

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

contains the unknown type. For example,

$$\texttt{Nat} \to \texttt{Bool} \;\sim\; ? \to \texttt{Bool}$$

Because $? \sim \texttt{Nat}$ and $\texttt{Bool} \sim \texttt{Bool}$. The rules for consistency in Figure 1 are labeled with their proof constructors. For example, the following is a proof of the above example:

$$\texttt{Fun}\sim\,(\texttt{UnkL}\sim[\texttt{Nat}])\,(\texttt{Base}\sim[\texttt{Bool}])$$

We use a colon for Agda's "proves" relation, which can also be read as a "has-type" relation thanks to the Curry–Howard correspondence (Howard, 1980). In the rule Fun∼, the A and A' flip in the premise, which is unusual but does not matter; it just makes some of the Agda proofs easier.

Figure 1 also defines a join function $\bigsqcup$ that computes the least upper bound of two types with respect to the precision relation $\sqsubseteq$ (with ? at the bottom) (Siek & Taha, 2006a; Siek & Vachharajani, 2008), which is defined in Figure 4 of Section 2.1. The join function is typically defined on a pair of types and is a partial function where it is defined if and only if the two types are consistent. Here, we instead make $\bigsqcup$ a total function over proofs of consistency.

Finally, Figure 1 defines the matching relation $A \rhd B$ that is used in the typing rules for eliminators (such as function application and accessing elements of a pair). The matching relation handles the situation where the function or pair expression has type ?.

Variables. The function $\mathbb{V}$, defined in Figure 3, maps a typing context Γ and type A to the set of all variables that have type A in context Γ. As stated above, variables are de Bruijn indices, that is, natural numbers where the number x refers to the xth enclosing lambda abstraction. There are two constructors for variables: Z (zero) and S (plus one). The two rules in Figure 3 correspond to the signatures of these two constructors, where premises (above the line) are the parameter types and the conclusion (below the line) is the result type. The variable Z refers to the first lexical position in the enclosing context, so Z takes no parameters, and its result type is $\mathbb{V}\;\Gamma\;A$ if Γ is a non-empty typing context where type A is at the front. A variable of the form $\texttt{S}\;x$ refers to one position further out than that of x. So the constructor S has one parameter, a variable in $\mathbb{V}\;\Gamma\;A$ for some Γ and A, and its result type is a variable in $\mathbb{V}\;(\Gamma \cdot B)\;A$, for any type B. An expression formed by combinations of the constructors Z and S is a proof of a proposition of the form $\mathbb{V}\;\Gamma\;A$. For example, $S\;S\;Z$ is a proof of $\mathbb{V}\;(\emptyset \cdot \texttt{Bool} \cdot \texttt{Nat} \cdot \texttt{Int})\;\texttt{Bool}$ because Bool is at position 2 in the typing context $\emptyset \cdot \texttt{Bool} \cdot \texttt{Nat} \cdot \texttt{Int}$.

Constants. We represent the constants of GTLC by a particular set of Agda values. First, we carve out the subset of GTLC types that a constant is allowed to have, which are the base types and n-ary functions over base types. We call them *primitive types* and inductively define the predicate $\mathbb{P}\;A$ in Figure 3 to identify them. We then define a mapping $[\![\cdot]\!]$ from $\mathbb{P}\;A$ to Agda types (elements of Set), also in Figure 3.

Terms. The syntax of GTLC terms is defined in Figure 2 and their typing rules are presented in Figure 3. Terms, ranging over L, M, and N, are defined using the Abstract Binding

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$$
\begin{array}{lllll}
\text{blame labels} & & \ell & & \\
\text{GTLC terms} & L, M, N & ::= & \$k \mid \text{`}x \mid \lambda[A]\, N \mid (L\, M)_\ell \mid \mathtt{if}_\ell\, L\, M\, N \mid \\
& & & \mathtt{cons}\, M\, N \mid \pi_i^\ell\, M, i \in \{1,2\} \mid \\
& & & \mathtt{inl}[B]\, M \mid \mathtt{inr}[A]\, M \mid \mathtt{case}_\ell[B, C]\, L\, M\, N
\end{array}
$$

Fig. 2. Syntax of the Gradually Typed Lambda Calculus (GTLC).

Tree library, so they are extrinsically typed. As usual, we define the typing judgment as $\Gamma \vdash_G M : A$, represented by an Agda data type.

The constant $\$k$ has type P in context Γ provided that the Agda value k has type $[\![P]\!]$ and P proves that A is a primitive type. A variable $\text{`}x$ has type A in context Γ if x is a de Bruijn index in $\mathbb{V}\, \Gamma\, A$ (explained above).

As usual, a lambda abstraction $(\lambda[A]\, M)$ has type $A \to B$ in context Γ provided that M has type B in context $\Gamma \cdot A$. Lambda abstraction does not include a parameter name because we represent variables as de Bruijn indices. An application $(L\, M)_\ell$ has type B if L has some type A that matches a function type $A_1 \to A_2$, M has some type B, and B is consistent with A_1. The blame label ℓ is a unique identifier for this location in the source code. Each blame label has a complement $\overline{\ell}$ and the complement operation is involutive, that is, $\overline{\overline{\ell}} = \ell$.

The term $\mathtt{if}_\ell\, L\, M\, N$ requires that the type of L is consistent with `Bool`, M and N have consistent types, and the type of the `if` as a whole is the join of the types of M and N. The rules for pairs and projection are straightforward. Regarding sums, in $\mathtt{case}_\ell\, L\, M\, N$, the type of L is consistent with the sum type $B_1 + C_1$. A variable of type B_1 goes into scope for the branch M and a variable of type C_1 goes into scope for N. The types of the branches, B_2 and C_2, must be consistent and the type of the `case` is the join of B_2 and C_2.

Examples. The following are a few example terms and their typing derivations in the GTLC.

$$
\begin{array}{lcccc}
\emptyset & \vdash_G & \mathtt{cons}\ \$2\ \$3 & : & \mathtt{Nat} \times \mathtt{Nat} \\
\emptyset & \vdash_G & ((\lambda[?]\ \text{`}\mathtt{Z})\ \$4)_{\ell_1} & : & \mathtt{Nat} \\
\emptyset & \vdash_G & \mathtt{case}_{\ell_2}\ (\mathtt{inr}[\mathtt{Bool}]\ \$\mathtt{true})\ (\text{`}\mathtt{Z})\ (\$\neg\ \text{`}\mathtt{Z})_{\ell_3} & : & \mathtt{Bool}
\end{array}
$$

2.1 Static gradual guarantee

Figure 4 presents the definitions of the precision relations on GTLC types, terms, and typing contexts. In particular, a type A is less precise than A', written $A \sqsubseteq A'$, if A and A' are equal except in the places where A has type ?.

The precision relation $\sqsubseteq^*$ between typing contexts is straightforward. We require that each variable has types that are related by $\sqsubseteq$ in the respective typing contexts. We say term M' in GTLC is *more precise* than M if the type annotations in the former are more precise than the corresponding annotations in the latter, written $M \sqsubseteq^G M'$. For example,

$$
\begin{array}{c}
\emptyset \cdot ? \sqsubseteq^* \emptyset \cdot \mathtt{Int} \\
((\lambda[?]\ \text{`}\mathtt{Z})\ \$42)_\ell \sqsubseteq^G ((\lambda[\mathtt{Nat}]\ \text{`}\mathtt{Z})\ \$42)_{\ell'}
\end{array}
$$

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$\boxed{\mathbb{V}\,\Gamma\,A}$

$$\mathtt{Z} : \frac{}{\mathbb{V}\,(\Gamma \cdot A)\,A} \qquad \mathtt{S} : \frac{\mathbb{V}\,\Gamma\,A}{\mathbb{V}\,(\Gamma \cdot B)\,A}$$

$\boxed{\mathbb{P}\,A}$

$$\mathtt{PBase}[b] : \frac{}{\mathbb{P}\,b} \qquad \mathtt{PFun}[b] : \frac{\mathbb{P}\,A}{\mathbb{P}\,(b \to A)}$$

$\boxed{[\![-]\!] : \mathcal{B} \to \mathsf{Set}}$ $\qquad$ $\boxed{[\![-]\!] : \mathbb{P}\,A \to \mathsf{Set}}$

$$\begin{aligned} [\![\mathtt{Bool}]\!] &= \mathbb{B} \\ [\![\mathtt{Nat}]\!] &= \mathbb{N} \\ [\![\mathtt{Int}]\!] &= \mathbb{Z} \\ [\![\mathtt{Unit}]\!] &= \top \\ [\![\bot]\!] &= \bot \end{aligned} \qquad \begin{aligned} [\![\mathtt{PBase}[b]]\!] &= [\![b]\!] \\ [\![\mathtt{PFun}[b]\ P]\!] &= [\![b]\!] \to [\![P]\!] \end{aligned}$$

$\boxed{\Gamma \vdash_G A}$

$$\$k : \frac{}{\Gamma \vdash_G A}\ k : [\![P]\!], P : \mathbb{P}\,A \qquad {}`x : \frac{}{\Gamma \vdash_G A}\ x : \mathbb{V}\,\Gamma\,A$$

$$\lambda[A] : \frac{\Gamma \cdot A \vdash_G B}{\Gamma \vdash_G A \to B} \qquad (-\ -)_\ell : \frac{\Gamma \vdash_G A \quad \Gamma \vdash_G B}{\Gamma \vdash_G A_2}\ A \triangleright A_1 \to A_2, A_1 \sim B$$

$$\mathtt{if}_\ell : \frac{\Gamma \vdash_G A \quad \Gamma \vdash_G B \quad \Gamma \vdash_G C}{\Gamma \vdash_G \bigsqcup(cn)}\ A \sim \mathtt{Bool}, cn : B \sim C$$

$$\mathtt{cons} : \frac{\Gamma \vdash_G A \quad \Gamma \vdash_G B}{\Gamma \vdash_G A \times B} \qquad \pi_i^\ell : \frac{\Gamma \vdash_G A}{\Gamma \vdash_G A_i} A \triangleright A_1 \times A_2$$

$$\mathtt{inl}[B] : \frac{\Gamma \vdash_G A}{\Gamma \vdash_G A + B} \qquad \mathtt{inr}[A] : \frac{\Gamma \vdash_G B}{\Gamma \vdash_G A + B}$$

$$\mathtt{case}_\ell[B_1, C_1] : \frac{\Gamma \vdash_G A \quad \Gamma \cdot B_1 \vdash_G B_2 \quad \Gamma \cdot C_1 \vdash_G C_2}{\Gamma \vdash_G \bigsqcup(bc)}\ \begin{array}{l} A_1 \sim (B_1 + C_1), \\ bc : B_2 \sim C_2 \end{array}$$

Fig. 3. Typing rules of the Gradually Typed Lambda Calculus (GTLC).

An important property of a gradual type system is that reducing the precision of type annotations in a well-typed term should yield a well-typed term, a property known as the static gradual guarantee (Siek *et al.*, 2015b). Here, we extend this result for the GTLC to include products and sums.

Lemma 1. *Suppose* $\Gamma \sqsubseteq^* \Gamma'$. *If* $x : \mathbb{V}\,\Gamma'\,A'$, *there exists type* A *such that* $x : \mathbb{V}\,\Gamma\,A$ *and* $A \sqsubseteq A'$.

Proof sketch. By induction on x. □

Theorem 2 (Static Gradual Guarantee). *Suppose* M' *is well-typed* $\Gamma' \vdash_G M' : A'$. *If* $\Gamma \sqsubseteq^* \Gamma'$ *and* $M \sqsubseteq^G M'$, *there exists type* A *such that* $\Gamma \vdash_G M : A$ *and* $A \sqsubseteq A'$.

Proof sketch. By the induction on the typing derivation $\Gamma' \vdash_G M' : A'$. We briefly describe the main idea of the interesting cases, since the full proof is mechanized in Agda.

Case $`x$ By Lemma 1 and typing rule $\vdash\mathtt{var}$.
Case $\lambda\, N'$ By the induction hypothesis for N' and the extended typing context.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$\boxed{A \sqsubseteq A'}$

$$\frac{}{? \sqsubseteq A} \qquad \frac{}{b \sqsubseteq b} \qquad \frac{A \sqsubseteq A' \quad B \sqsubseteq B'}{A \to B \sqsubseteq A' \to B'}$$

$$\frac{A \sqsubseteq A' \quad B \sqsubseteq B'}{A \times B \sqsubseteq A' \times B'} \qquad \frac{A \sqsubseteq A' \quad B \sqsubseteq B'}{A + B \sqsubseteq A' + B'}$$

$\boxed{M \sqsubseteq^G M'}$

$$\frac{}{\$k \sqsubseteq^G \$k} \qquad \frac{}{`x \sqsubseteq^G `x}$$

$$\frac{A \sqsubseteq A' \quad N \sqsubseteq^G N'}{\lambda[A]\, N \sqsubseteq^G \lambda[A']\, N'} \qquad \frac{L \sqsubseteq^G L' \quad M \sqsubseteq^G M'}{(L\ M)_\ell \sqsubseteq^G (L'\ M')_{\ell'}}$$

$$\frac{L \sqsubseteq^G L' \quad M \sqsubseteq^G M' \quad N \sqsubseteq^G N'}{\mathtt{if}_\ell\, L\, M\, N \sqsubseteq^G \mathtt{if}_{\ell'}\, L'\, M'\, N'}$$

$$\frac{M \sqsubseteq^G M' \quad N \sqsubseteq^G N'}{\mathtt{cons}\, M\, N \sqsubseteq^G \mathtt{cons}\, M'\, N'} \qquad \frac{M \sqsubseteq^G M'}{\pi_1\, M \sqsubseteq^G \pi_1\, M'} \qquad \frac{M \sqsubseteq^G M'}{\pi_2\, M \sqsubseteq^G \pi_2\, M'}$$

$$\frac{B \sqsubseteq B' \quad M \sqsubseteq^G M'}{\mathtt{inl}[B]\, M \sqsubseteq^G \mathtt{inl}[B']\, M'} \qquad \frac{A \sqsubseteq A' \quad M \sqsubseteq^G M'}{\mathtt{inr}[A]\, M \sqsubseteq^G \mathtt{inr}[A']\, M'}$$

$$\frac{B_1 \sqsubseteq B_1' \quad C_1 \sqsubseteq C_1' \quad L \sqsubseteq^G L' \quad M \sqsubseteq^G M' \quad N \sqsubseteq^G N'}{\mathtt{case}_\ell[B_1, C_1]\, L\, M\, N \sqsubseteq^G \mathtt{case}_{\ell'}[B_1', C_1']\, L'\, M'\, N'}$$

$\boxed{\Gamma \sqsubseteq^* \Gamma'}$

$$\frac{}{\emptyset \sqsubseteq^* \emptyset} \qquad \frac{A \sqsubseteq A' \quad \Gamma \sqsubseteq^* \Gamma'}{\Gamma \cdot A \sqsubseteq^* \Gamma' \cdot A'}$$

Fig. 4. Precision on GTLC types, terms, and typing contexts.

Case $L'\ M'$ The induction hypothesis for L' produces two subcases: The term L in $L\ M \sqsubseteq^G L'\ M'$ can be either of ? or of function type. If L is of type ?, the theorem is proved by type matching and ? being less precise than any type. On the other hand, if L is of function type, it is proved by the fact that the codomain types satisfy the precision relation.

Other cases follow the same structure as function application: by the induction hypotheses for subterms and casing on whether the left side of a precision relation can be ? whenever necessary. □

3 Parameterized Cast Calculus

Recall that the dynamic semantics of the GTLC is defined by translation to a cast calculus. There are many cast calculi in the literature with differing semantics and efficiency guarantees. In this section, we define the Parameterized Cast Calculus $CC(\Rightarrow)$ that captures the similarities in many of those calculi, for the purposes of language specification. (As a guide to implementation, we present the Space-Efficient Parameterized Cast Calculus in Section 6.) This section begins with the static semantics of $CC(\Rightarrow)$ and a number of parameters packaged into PreCastStruct (Section 3.1), which is then used to express the notion of Value, evaluation contexts (Section 3.2), and eta-style cast reductions (Section 3.3). We

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$$\boxed{M : \Gamma \vdash A}$$

$$\$k : \frac{}{\Gamma \vdash P}\ k : [\![P]\!], P : \mathbb{P}\, A \qquad `x : \frac{}{\Gamma \vdash A}\ x : \mathbb{V}\, \Gamma\, A$$

$$\lambda : \frac{\Gamma \cdot A \vdash B}{\Gamma \vdash A \to B} \qquad (- \ -) : \frac{\Gamma \vdash A_1 \to A_2 \quad \Gamma \vdash A_1}{\Gamma \vdash A_2}$$

$$\texttt{if} : \frac{\Gamma \vdash \texttt{Bool} \quad \Gamma \vdash B \quad \Gamma \vdash B}{\Gamma \vdash B}$$

$$\texttt{cons} : \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B} \qquad \pi_i : \frac{\Gamma \vdash A_1 \times A_2}{\Gamma \vdash A_i}$$

$$\texttt{inl}[B] : \frac{\Gamma \vdash A}{\Gamma \vdash A + B} \qquad \texttt{inr}[A] : \frac{\Gamma \vdash B}{\Gamma \vdash A + B}$$

$$\texttt{case} : \frac{\Gamma \vdash A_1 + A_2 \quad \Gamma \vdash A_1 \to B \quad \Gamma \vdash A_2 \to B}{\Gamma \vdash B}$$

$$-\langle \texttt{c} \rangle : \frac{\Gamma \vdash A}{\Gamma \vdash B}\ c : A \Rightarrow B \qquad \texttt{blame}\, \ell : \frac{}{\Gamma \vdash A}$$

Fig. 5. Term constructors for the Parameterized Cast Calculus $CC(\Rightarrow)$.

introduce another parameter, for applying a cast to a value, in CastStruct (Section 3.4), and then define the dynamic semantics of $CC(\Rightarrow)$, with definitions of substitution (Section 3.5) and the reduction semantics (Section 3.6).

We present $CC(\Rightarrow)$ as an intrinsically typed calculus with de Bruijn representation of variables, analogous to the formalization of the Simply Typed Lambda Calculus in the *DeBruijn* Chapter of PLFA. As such, the terms are represented by derivations of their typing judgment. The judgment has the form $\Gamma \vdash A$, which says that type A can be inhabited in context Γ. So terms, ranged over by L, M, N, are proofs of propositions of the form $\Gamma \vdash A$. For example, a typing judgment normally written $\Gamma \vdash M : A$ here has the form $M : \Gamma \vdash A$.

The term constructors for $CC(\Rightarrow)$ are defined in Figure 5. Like most cast calculi, $CC(\Rightarrow)$ extends the Simply Typed Lambda Calculus with the unknown type ? and explicit runtime casts. Unlike other cast calculi, the $CC(\Rightarrow)$ calculus is parameterized over the representation of casts, that is, the parameter $\Rightarrow$ is a function that, given a source and target type, returns the representation type for casts from A to B. So $c : A \Rightarrow B$ says that c is a cast from A to B.

The types and variables of the Parameterized Cast Calculus are the same as those of the GTLC (Section 2). The intrinsically typed terms of the Parameterized Cast Calculus are defined in Figure 5. Cast application is written $M\langle c \rangle$ where the cast representation c is not concrete but is instead specified by the parameter $\Rightarrow$. In the literature, some cast calculi annotate the cast application form with a blame label. In other cast calculi, such as coercion-based calculi or threesomes, blame labels instead appear inside the cast c. For purposes of uniformity, we place the responsibility for blame tracking inside the cast representation, which means they do not appear directly in the cast application form of the Parameterized Cast Calculus. As usual there is an uncatchable exception $\texttt{blame}\, \ell$.

3.1 *The* PreCastStruct *structure*

We introduce the first of several structures (as in algebraic structures) that group together parameters needed to define the notion of values, frames, and the reduction relation for the Parametric Cast Calculus. The structures are represented in Agda as dependent records.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

The PreCastStruct structure includes the operations and predicates that do not depend on the terms of the cast calculus so that this structure can be reused for different cast calculi, such as the space-efficient cast calculus in Section 6. The CastStruct structure extends PreCastStruct with the one additional operation that depends on terms, which is applyCast.

One of the main responsibilities of the PreCastStruct structure is categorizing casts as either *active* or *inert*. An active cast is one that needs to be reduced by invoking applyCast (see reduction rule (`cast`) in Figure 8). An inert cast is one that does not need to be reduced, which means that a value with an inert cast around it forms a larger value (see the definition of Value in Figure 6). Different cast calculi may make different choices regarding which casts are active and inert, so the PreCastStruct structure includes the two predicates Active and Inert to parameterize these differences. The proof of Progress (Theorem 14) needs to know that every cast can be categorized as either active or inert, so PreCastStruct also includes the ActiveOrInert field which is a (total) function from casts to their categorization as active or inert.

The reduction semantics must also identify casts whose source and target type have the same head type constructor, such as a cast from $A \times B$ to $C \times D$. We refer to such casts as *cross* casts and include the Cross predicate in the PreCastStruct structure. When a cross cast is inert, the reduction semantics must decompose the cast when reducing the elimination form for that type. For example, when accessing the `fst` element of a pair V that is wrapped in an inert cast c, we decompose the cast using an operator, also called fst, provided in the PreCastStruct structure. Here is the relevant reduction rule, which is (`fst-cast`) in Figure 8:

$$\texttt{fst}\,(V\langle c\rangle) \longrightarrow (\texttt{fst}\ V)\langle \mathsf{fst}\, c\, x\rangle$$

Finally, the proof of Progress (Theorem 14) also needs to know that the cast is a cross cast when the target type of an inert cast is a function, product, or sum type. Thus, the PreCastStruct includes the fields InsertCross$\rightarrow$, InsertCross$\times$, and InsertCross$+$. The target type of an inert cast may not be a base type (field baseNotInert) to ensure that the canonical forms at base type are just constants.

The fields of the PreCastStruct record as follows:

$- \Rightarrow - : \mathcal{T} \to \mathcal{T} \to \mathsf{Set}$
: Given the source and target type, this returns the Agda type for casts.

$\mathsf{Inert} : \forall AB.\, A \Rightarrow B \to \mathsf{Set}$
: This predicate categorizes the inert casts, that is, casts that when combined with a value, form a value.

$\mathsf{Active} : \forall AB.\, A \Rightarrow B \to \mathsf{Set}$
: This predicate categorizes the active casts, that is, casts that require a reduction rule to specify what happens when they are applied to a value.

$\mathsf{ActiveOrInert} : \forall AB.\, (c : A \Rightarrow B) \to \mathsf{Active}\, c \uplus \mathsf{Inert}\, c$
: All casts must be active or inert, which is used in the proof of Progress.

$\mathsf{Cross} : \forall AB.\, A \Rightarrow B \to \mathsf{Set}$
: This predicate categorizes the cross casts, that is, casts from one type constructor to the same type constructor, such as $A \to B \Rightarrow C \to D$. This categorization is needed to define other fields below, such as dom.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$\mathsf{InertCross}{\to} : \forall ABC.\, (c : A \Rightarrow B{\to}C) \to \mathsf{Inert}\, c \to \mathsf{Cross}\, c \times \Sigma A_1 A_2.\, A \equiv A_1{\to}A_2$
An inert cast whose target is a function type must be a cross cast. This field and the following two fields are used in the proof of Progress.

$\mathsf{InertCross}{\times} : \forall ABC.\, (c : A \Rightarrow B \times C) \to \mathsf{Inert}\, c \to \mathsf{Cross}\, c \times \Sigma A_1 A_2.\, A \equiv A_1 \times A_2$
An inert cast whose target is a pair type must be a cross cast.

$\mathsf{InertCross}{+} : \forall ABC.\, (c : A \Rightarrow B + C) \to \mathsf{Inert}\, c \to \mathsf{Cross}\, c \times \Sigma A_1 A_2.\, A \equiv A_1 + A_2$
An inert cast whose target is a sum type must be a cross cast.

$\mathsf{baseNotInert} : \forall Ab.\, (c : A \Rightarrow b) \to \neg \mathsf{Inert}\, c$
A cast whose target is a base type must never be inert. This field is used in the proof of Progress.

$\mathsf{dom} : \forall A_1 A_2 B_1 B_2.\, (c : (A_1 \to A_2) \Rightarrow (B_1 \to B_2)) \to \mathsf{Cross}\, c \to B_1 \Rightarrow A_1$
Given a cross cast between function types, dom returns the part of the cast between their domain types. As usual, domains are treated contravariantly, so the result is a cast from B_1 to A_1.

$\mathsf{cod} : \forall A_1 A_2 B_1 B_2.\, (c : (A_1 \to A_2) \Rightarrow (B_1 \to B_2)) \to \mathsf{Cross}\, c \to A_2 \Rightarrow B_2$
Given a cross cast between function types, cod returns the part of the cast between the codomain types.

$\mathsf{fst} : \forall A_1 A_2 B_1 B_2.\, (c : (A_1 \times A_2) \Rightarrow (B_1 \times B_2)) \to \mathsf{Cross}\, c \to A_1 \Rightarrow B_1$
Given a cross cast between pair types, fst returns the part of the cast between the first components of the pair.

$\mathsf{snd} : \forall A_1 A_2 B_1 B_2.\, (c : (A_1 \times A_2) \Rightarrow (B_1 \times B_2)) \to \mathsf{Cross}\, c \to A_2 \Rightarrow B_2$
Given a cross cast between pair types, snd returns the part of the cast between the second components of the pair.

$\mathsf{inl} : \forall A_1 A_2 B_1 B_2.\, (c : (A_1 + A_2) \Rightarrow (B_1 + B_2)) \to \mathsf{Cross}\, c \to A_1 \Rightarrow B_1$
Given a cross cast between sum types, inl returns the part of the cast for the first branch.

$\mathsf{inr} : \forall A_1 A_2 B_1 B_2.\, (c : (A_1 + A_2) \Rightarrow (B_1 + B_2)) \to \mathsf{Cross}\, c \to A_2 \Rightarrow B_2$
Given a cross cast between sum types, inr returns the part of the cast for the second branch.

3.2 Values and frames of CC(⇒)

This section is parameterized by a PreCastStruct. So, for example, when we refer to $\Rightarrow$ and Inert, we mean those fields of the PreCastStruct.

The values of the Parameterized Cast Calculus are defined in Figure 6. The judgment $\mathsf{Value}\, M$ says that the term M is a value. Let V and W range over values. The only rule specific to gradual typing is `Vcast` which states that a cast application $V\langle c\rangle$ is a value if c is an inert cast.

A value of type ? must be a cast application where the cast is inert and its target type is ?.

Lemma 3 (Canonical Form for type ?). *If* $M : \Gamma \vdash ?$ *and* $\mathsf{Value}\, M$, *then* $M \equiv M'\langle c\rangle$ *where* $M' : \Gamma \vdash A$, $c : A \Rightarrow ?$, *and* $\mathsf{Inert}\, c$.

In the reduction semantics, we use single-term evaluation contexts, called *frames*, to collapse the many congruence rules that one usually finds in a structural operational semantics

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$\boxed{V : (\Gamma \vdash A) \to \mathsf{Set}}$

$$\mathtt{V}\lambda : \frac{}{\mathsf{Value}\,(\lambda M)} \qquad \mathtt{Vconst} : \frac{}{\mathsf{Value}\,(\$k)}$$

$$\mathtt{Vpair} : \frac{\mathsf{Value}\,M \qquad \mathsf{Value}\,N}{\mathsf{Value}\,(\mathtt{cons}\,M\,N)}$$

$$\mathtt{Vinl} : \frac{\mathsf{Value}\,M}{\mathsf{Value}\,(\mathtt{inl}[B]\,M)} \qquad \mathtt{Vinr} : \frac{\mathsf{Value}\,M}{\mathsf{Value}\,(\mathtt{inr}[A]\,M)}$$

$$\mathtt{Vcast} : \frac{\mathsf{Value}\,M}{\mathsf{Value}\,(M\langle c\rangle)}\ \mathsf{Inert}\,c$$

$\boxed{F : \Gamma \vdash A \rightarrowtail B}$

$$(\Box\ -) : \frac{\Gamma \vdash A}{\Gamma \vdash (A \to B) \rightarrowtail B} \qquad (-\ \Box) : \frac{M : \Gamma \vdash (A \to B)}{\Gamma \vdash A \rightarrowtail B}\ \mathsf{Value}\,M$$

$$\mathtt{if}\,\Box\ -\ - : \frac{\Gamma \vdash A \qquad \Gamma \vdash A}{\Gamma \vdash \mathtt{Bool} \rightarrowtail A}$$

$$\mathtt{cons}\ -\ \Box : \frac{M : \Gamma \vdash A}{\Gamma \vdash B \rightarrowtail A \times B}\ \mathsf{Value}\,M \qquad \mathtt{cons}\ \Box\ - : \frac{\Gamma \vdash B}{\Gamma \vdash A \rightarrowtail A \times B}$$

$$\pi_i\,\Box : \frac{}{\Gamma \vdash A_1 \times A_2 \rightarrowtail A_i}$$

$$\mathtt{inl}[B]\,\Box : \frac{}{\Gamma \vdash A \rightarrowtail A \times B} \qquad \mathtt{inr}[A]\,\Box : \frac{}{\Gamma \vdash B \rightarrowtail A \times B}$$

$$\mathtt{case}\,\Box\ -\ - : \frac{\Gamma \vdash A \to C \qquad \Gamma \vdash B \to C}{\Gamma \vdash A + B \rightarrowtail C} \qquad \Box\langle c\rangle : \frac{}{\Gamma \vdash A \rightarrowtail B}\ c : A \Rightarrow B$$

$\boxed{\mathit{plug} : \forall \Gamma AB.\,(\Gamma \vdash A) \to (\Gamma \vdash A \rightarrowtail B) \to (\Gamma \vdash B)}$

$$\begin{aligned}
\mathit{plug}\ L\ (\Box\ M) &= (L\ M)\\
\mathit{plug}\ M\ (L\ \Box) &= (L\ M)\\
\mathit{plug}\ L\ (\mathtt{if}\,\Box\,M\,N) &= \mathtt{if}\,L\,M\,N\\
\mathit{plug}\ N(\mathtt{cons}\,M\,\Box) &= \mathtt{cons}\,M\,N\\
\mathit{plug}\ M(\mathtt{cons}\,\Box\,N) &= \mathtt{cons}\,M\,N\\
\mathit{plug}\ M(\pi_i\,\Box) &= \pi_i\,M\\
\mathit{plug}\ M(\mathtt{inl}[B]\,\Box) &= \mathtt{inl}[B]\,M\\
\mathit{plug}\ M(\mathtt{inr}[A]\,\Box) &= \mathtt{inr}[A]\,M\\
\mathit{plug}\ L(\mathtt{case}\,\Box\,M\,N) &= \mathtt{case}\,L\,M\,N\\
\mathit{plug}\ M(\Box\langle c\rangle) &= M\langle c\rangle
\end{aligned}$$

Fig. 6. Values and frames of $CC(\Rightarrow)$.

into a single congruence rule. Unlike regular evaluation contexts, frames are not recursive. Instead that recursion is in the congruence reduction rule.

The definition of frames for the Parameterized Cast Calculus is given in Figure 6. The definition is typical for a call-by-value calculus. We also define the *plug* function at the bottom of Figure 6, which replaces the hole in a frame with a term, producing a term.

The *plug* function is type preserving. This is proved in Agda by embedding the statement of this lemma into the type of *plug* (see Figure 6) and then relying on Agda to check that the definition of *plug* satisfies its declared type.

Lemma 4 (Frame Filling). *If* $M : \Gamma \vdash A$ *and* $\vdash F : A \rightarrowtail B$, *then* $\mathit{plug}\,M\,F : \Gamma \vdash B$.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

3.3 The eta cast reduction rules

This section is parameterized by a PreCastStruct.

Some cast calculi include reduction rules that resemble η-reduction (Flanagan, 2006; Siek & Taha, 2006a). For example, the following rule reduces a cast between two function types, applied to a value V, by η-expanding V and inserting the appropriate casts:

$$V\langle A \to B \Rightarrow C \to D\rangle \longrightarrow \lambda(V\ (\text{`Z}\langle C \Rightarrow A\rangle))\langle B \Rightarrow D\rangle$$

Here, we define three auxiliary functions that apply casts between two function types, two pair types, and two sum types, respectively. Each of these functions requires the cast c to be a cross cast. These auxiliary functions are used by cast calculi that choose to categorize these cross casts as active casts.

$$\begin{aligned}
&\mathsf{eta}\to && : \forall \Gamma ABCD.\, (M : \Gamma \vdash A \to B) \to (c : (A \to B) \Rightarrow (C \to D)) \\
&&& \quad \to \mathsf{Cross}\, c \to (\Gamma \vdash C \to D) \\
&\mathsf{eta}\to\, M\, c\, x && = \lambda\, ((\mathsf{rename}\, \mathtt{S}\, M)\, (\text{`Z}\langle \mathsf{dom}\, c\, x\rangle))\langle \mathsf{cod}\, c\, x\rangle \\
\\
&\mathsf{eta}\times && : \forall \Gamma ABCD.\, (M : \Gamma \vdash A \times B) \to (c : (A \times B) \Rightarrow (C \times D)) \\
&&& \quad \to \mathsf{Cross}\, c \to (\Gamma \vdash C \times D) \\
&\mathsf{eta}\times\, M\, c\, x && = \mathtt{cons}\, (\pi_1 M)\langle \mathtt{fst}\, c\, x\rangle\, (\pi_2 M)\langle \mathtt{snd}\, c\, x\rangle \\
\\
&\mathsf{eta}+ && : \forall \Gamma ABCD.\, (M : \Gamma \vdash A + B) \to (c : (A + B) \Rightarrow (C + D)) \\
&&& \quad \to \mathsf{Cross}\, c \to (\Gamma \vdash C + D) \\
&\mathsf{eta}+\, M\, c\, x && = \mathtt{case}\, M\, (\lambda \mathtt{inl}[D](\mathtt{Z}\langle \mathtt{inl}\, c\, x\rangle))\, (\lambda \mathtt{inr}[C](\mathtt{Z}\langle \mathtt{inr}\, c\, x\rangle))
\end{aligned}$$

3.4 The CastStruct structure

The CastStruct record type extends PreCastStruct with one more field, for applying an active cast to a value. Thus, this structure depends on terms of the cast calculus:

$$\mathsf{applyCast} : \forall \Gamma AB.\, (M : \Gamma \vdash A) \to \mathsf{Value}\, M \to (c : A \Rightarrow B) \to \mathsf{Active}\, c \to \Gamma \vdash B$$

3.5 Substitution in $CC(\Rightarrow)$

We define substitution functions (Figure 7) for $CC(\Rightarrow)$ in the style of PLFA (Wadler & Kokke, 2019), due to Conor McBride. A *renaming* is a map ρ from variables (natural numbers) to variables. A *substitution* is a map σ from variables to terms. The notation $M[N]$ substitutes term N for all occurrences of variable Z inside M. Its definition relies on several auxiliary functions. Renaming extension, ext ρ, transports ρ under one lambda abstraction. The result maps Z to itself, because Z is bound by the lambda abstraction. For any other variable, ext ρ transports the variable above the lambda by subtracting one, looking it up in ρ, and then transports it back under the lambda by adding one. Simultaneous renaming, rename ρ M, applies ρ to all the free variables in M. Substitution extension, exts σ, transports σ under one lambda abstraction. The result maps Z to itself. For any other variable, exts σ transports the variable above the lambda by subtracting one, looking it up in σ, and then transporting the resulting term under the lambda by incrementing every free variable, using simultaneous renaming. Simultaneous substitution, subst σ M, applies σ to the free

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$\boxed{\mathsf{rename}\ \rho\ M}$

$$
\begin{aligned}
\mathsf{rename}\ \rho\ k &= k \\
\mathsf{rename}\ \rho\ x &= \rho(x) \\
\mathsf{rename}\ \rho\ (\lambda\ M) &= \lambda\ (\mathsf{rename}\ (\mathsf{ext}\ \rho)\ N) \\
\mathsf{rename}\ \rho\ (M\ N) &= (\mathsf{rename}\ \rho\ M)\ (\mathsf{rename}\ \rho\ N) \\
\mathsf{rename}\ \rho\ (\mathtt{if}\ L\ M\ N) &= \mathtt{if}\ (\mathsf{rename}\ \rho\ L)\ (\mathsf{rename}\ \rho\ M)\ (\mathsf{rename}\ \rho\ N) \\
\mathsf{rename}\ \rho\ (\mathtt{cons}\ M\ N) &= \mathtt{cons}\ (\mathsf{rename}\ \rho\ M)\ (\mathsf{rename}\ \rho\ N) \\
\mathsf{rename}\ \rho\ (\pi_i\ M) &= \pi_i\ (\mathsf{rename}\ \rho\ M) \\
\mathsf{rename}\ \rho\ (\mathtt{inl}[B]\ M) &= \mathtt{inl}[B]\ (\mathsf{rename}\ \rho\ M) \\
\mathsf{rename}\ \rho\ (\mathtt{inr}[A]\ M) &= \mathtt{inr}[A]\ (\mathsf{rename}\ \rho\ M) \\
\mathsf{rename}\ \rho\ (\mathtt{case}\ L\ M\ N) &= \mathtt{case}\ (\mathsf{rename}\ \rho\ L)\ (\mathsf{rename}\ \rho\ M)\ (\mathsf{rename}\ \rho\ N) \\
\mathsf{rename}\ \rho\ (M\langle c\rangle) &= (\mathsf{rename}\ \rho\ M)\langle c\rangle \\
\mathsf{rename}\ \rho\ (\mathtt{blame}\ \ell) &= \mathtt{blame}\ \ell
\end{aligned}
$$

$\boxed{\mathsf{ext}\ \rho}$

$$
\begin{aligned}
\mathsf{ext}\ \rho\ \mathtt{Z} &= \mathtt{Z} \\
\mathsf{ext}\ \rho\ (\mathtt{S}\ x) &= \mathtt{S}\ \rho(x)
\end{aligned}
$$

$\boxed{\mathsf{exts}\ \sigma}$

$$
\begin{aligned}
\mathsf{exts}\ \sigma\ \mathtt{Z} &= \mathtt{Z} \\
\mathsf{exts}\ \sigma\ (\mathtt{S}\ x) &= \mathsf{rename}\ \mathtt{S}\ (\sigma x)
\end{aligned}
$$

$\boxed{\mathsf{subst}\ \sigma\ M}$

$$
\begin{aligned}
\mathsf{subst}\ \sigma\ k &= k \\
\mathsf{subst}\ \sigma\ x &= \sigma(x) \\
\mathsf{subst}\ \sigma\ (\lambda.\ M) &= \lambda.\ \mathsf{subst}\ (\mathsf{exts}\ \sigma)\ M \\
\mathsf{subst}\ \sigma\ (M\ N) &= \mathsf{subst}\ \sigma\ M\ \mathsf{subst}\ \sigma\ N \\
\mathsf{subst}\ \rho\ (\mathtt{if}\ L\ M\ N) &= \mathtt{if}\ (\mathsf{subst}\ \rho\ L)\ (\mathsf{subst}\ \rho\ M)\ (\mathsf{subst}\ \rho\ N) \\
\mathsf{subst}\ \rho\ (\mathtt{cons}\ M\ N) &= \mathtt{cons}\ (\mathsf{subst}\ \rho\ M)\ (\mathsf{subst}\ \rho\ N) \\
\mathsf{subst}\ \rho\ (\pi_i\ M) &= \pi_i\ (\mathsf{subst}\ \rho\ M) \\
\mathsf{subst}\ \rho\ (\mathtt{inl}[B]\ M) &= \mathtt{inl}[B]\ (\mathsf{subst}\ \rho\ M) \\
\mathsf{subst}\ \rho\ (\mathtt{inr}[A]\ M) &= \mathtt{inr}[A]\ (\mathsf{subst}\ \rho\ M) \\
\mathsf{subst}\ \rho\ (\mathtt{case}\ L\ M\ N) &= \mathtt{case}\ (\mathsf{subst}\ \rho\ L)\ (\mathsf{subst}\ \rho\ M)\ (\mathsf{subst}\ \rho\ N) \\
\mathsf{subst}\ \sigma\ (M\langle c\rangle) &= \mathsf{subst}\ \sigma\ M\langle c\rangle \\
\mathsf{subst}\ \sigma\ (\mathtt{blame}\ \ell) &= \mathtt{blame}\ \ell
\end{aligned}
$$

$\boxed{\mathsf{substZero}\ N}$

$$
\begin{aligned}
\mathsf{substZero}\ N\ \mathtt{Z} &= N \\
\mathsf{substZero}\ N\ (\mathtt{S}\ x) &= x
\end{aligned}
$$

$\boxed{M[N]}$

$$
M[N] = \mathsf{subst}\ (\mathsf{substZero}\ N)\ M
$$

Fig. 7. Substitution and its auxiliary functions.

variables in M. The notation $M[N]$ is meant to be used for β reduction, where M is the body of the lambda abstraction and N is the argument. What $M[N]$ does is substitute $\mathtt{Z}$ for N in M and also transports M above the lambda by incrementing the other free variables. All this is accomplished by building a substitution $\mathsf{substZero}\ N$ (also defined in Figure 7) and then applying it to M.

Substitution is type preserving, which is established by the following sequences of lemmas. As usual, we prove one theorem per function. In Agda, these theorems are proved by embedding their statements into the types of the four functions. Given a sequence S, we write $S!i$ to access its ith element. Recall that Γ and Δ range over typing contexts (which are sequences of types).

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

Lemma 5 (Renaming Extension). *Suppose that for position x in Γ, $\Gamma!x = \Delta!\rho(x)$. For any y and B, $(\Gamma \cdot B)!y = (\Delta \cdot B)!(\mathsf{ext}\ \rho)(y)$.*

Lemma 6 (Renaming Variables). *Suppose that for any x, $\Gamma!x = \Delta!\rho(x)$. If $M : \Gamma \vdash A$, then $\mathsf{rename}\ \rho\ M : \Delta \vdash A$.*

Lemma 7 (Substitution Extension). *Suppose that for any x, $\sigma(x) : \Delta \vdash \Gamma!x$. For any y and B, $\sigma(y) : (\Delta \cdot B) \vdash (\Gamma \cdot B)!y$.*

Proposition 8 (Simultaneous Substitution). *Suppose that for any x, $\sigma(x) : \Delta \vdash \Gamma!x$. If $M : \Gamma \vdash A$, then $\mathsf{subst}\ \sigma\ M : \Delta \vdash A$.*

Corollary 9 (Substitution). *If $M : \Gamma \cdot B \vdash A$ and $\Gamma \vdash N : B$, then $M[N] : \Gamma \vdash A$.*

3.6 Reduction semantics of CC(⇒)

This section is parameterized by CastStruct.

Figure 8 defines the reduction relation for $CC(\Rightarrow)$. The last eight rules are typical of the Simply-Typed Lambda Calculus, including rules for function application, conditional branching, projecting the first or second element of a pair, and case analysis on a sum. The congruence rule (ξ) says that reduction can happen underneath a single frame. The rule (ξ-blame) propagates an exception up one frame. Perhaps the most important rule is (cast), for applying an active cast to a value. This reduction rule simply delegates to the applyCast field of the CastStruct. The next four rules (fun-cast, fst-cast, snd-cast, and case-cast) handle the possibility that the CastStruct categorizes casts between functions, pairs, or sums as inert casts. In such situations, we need reduction rules for when cast-wrapped values flow into an elimination form. First, recall that the PreCastStruct record includes a proof that every inert cast between two function types is a cross cast. Also recall that the PreCastStruct record includes fields for decomposing a cross cast between function types into a cast on the domain and codomain. Putting these pieces together, the reduction rule (fun-cast) says that applying the cast-wrapped function $V\langle c\rangle$ to argument W reduces to an application of V to $W\langle \mathsf{dom}\ c\ x\rangle$ followed by the cast $\mathsf{cod}\ c\ x$, where x is the proof that c is a cross cast. The story is similar for for pairs and sums.

3.7 Determinism of CC(⇒)

The reduction relation of the Parameterized Cast Calculus is deterministic. We prove this fact by defining a function named hop that maps any term that is not a value to the same term that would result from one step of reduction. (See the Agda formalization for details.)

$$\mathsf{hop} : \forall A, M : \emptyset \vdash A \to \neg\mathsf{Value}\ M \to \emptyset \vdash A$$

Indeed, if M reduces to N, then $\mathsf{hop}\ M$ is N.

Lemma 10. *If $M \longrightarrow N$ and $p : \neg\mathsf{Value}\ M$, then $\mathsf{hop}\ M\ p = N$.*

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$\boxed{M \longrightarrow N}$

$$\frac{M \longrightarrow M'}{plug\ M\ F \longrightarrow plug\ M'\ F} \quad (\xi)$$

$$\frac{}{plug\ (\texttt{blame}\ \ell)\ F \longrightarrow \texttt{blame}\ \ell} \quad (\xi\text{-blame})$$

$$\frac{}{V\langle c\rangle \longrightarrow \mathsf{applyCast}\ V\ c\ a}\ a : \mathsf{Active}\ c \quad (\texttt{cast})$$

$$\frac{}{V\langle c\rangle\ W \longrightarrow (V\ W\langle \mathsf{dom}\ c\ x\rangle)\langle \mathsf{cod}\ c\ x\rangle}\ x : \mathsf{Cross}\ c, \mathsf{Inert}\ c \quad (\texttt{fun-cast})$$

$$\frac{}{\texttt{fst}\ (V\langle c\rangle) \longrightarrow (\texttt{fst}\ V)\langle \mathsf{fst}\ c\ x\rangle}\ x : \mathsf{Cross}\ c, \mathsf{Inert}\ c \quad (\texttt{fst-cast})$$

$$\frac{}{\texttt{snd}\ (V\langle c\rangle) \longrightarrow (\texttt{snd}\ V)\langle \mathsf{snd}\ c\ x\rangle}\ x : \mathsf{Cross}\ c, \mathsf{Inert}\ c \quad (\texttt{snd-cast})$$

$$\frac{}{\texttt{case}\ (V\langle c\rangle)\ W_1\ W_2 \longrightarrow \texttt{case}\ V\ W_1'\ W_2'} \quad (\texttt{case-cast})$$

$$\text{where}\quad \begin{array}{l} x : \mathsf{Cross}\ c, \mathsf{Inert}\ c \\ W_1' = \lambda(\mathsf{rename}\ \mathsf{S}\ W_1)\ (\mathsf{Z}\langle \mathsf{inl}\ c\ x\rangle) \\ W_2' = \lambda(\mathsf{rename}\ \mathsf{S}\ W_2)\ (\mathsf{Z}\langle \mathsf{inr}\ c\ x\rangle) \end{array}$$

$$\frac{}{(\lambda M)\ V \longrightarrow M[V]} \quad (\beta)$$

$$\frac{}{\texttt{if}\ \$\texttt{true}\ M\ N \longrightarrow M} \quad (\beta\text{-true})$$

$$\frac{}{\texttt{if}\ \$\texttt{false}\ M\ N \longrightarrow N} \quad (\beta\text{-false})$$

$$\frac{}{\texttt{fst}\ (\texttt{cons}\ V\ W) \longrightarrow V} \quad (\beta\text{-fst})$$

$$\frac{}{\texttt{snd}\ (\texttt{cons}\ V\ W) \longrightarrow W} \quad (\beta\text{-snd})$$

$$\frac{}{\texttt{case}\ (\texttt{inl}\ V)\ L\ M \longrightarrow L\ V} \quad (\beta\text{-caseL})$$

$$\frac{}{\texttt{case}\ (\texttt{inr}\ V)\ L\ M \longrightarrow M\ V} \quad (\beta\text{-caseR})$$

$$\frac{}{k\ k' \longrightarrow [\![k]\!]\ [\![k']\!]} \quad (\delta)$$

Fig. 8. Reduction for the Parameterized Cast Calculus $CC(\Rightarrow)$, parameterized by the CastStruct structure.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

Of course, a term that reduces cannot be a value.

Lemma 11. *If* $M \longrightarrow N$, *then* $\neg\mathsf{Value}\, M$.

The determinism of the Parameterized Cast Calculus follows directly from these lemmas.

Theorem 12. (Determinism). *If* $M \longrightarrow N$ *and* $M \longrightarrow N'$, *then* $N = N'$.

Proof. By Lemma 11, M is not a value, with proof p. So by Lemma 10, we have $\mathsf{hop}\, M\, p = N$ and also $\mathsf{hop}\, M\, p = N'$. Therefore, $N = N'$. □

3.8 Type safety of CC$(\Rightarrow)$

The Preservation theorem is a direct consequence of the type that we give to the reduction relation and that it was checked by Agda.

Theorem 13 (Preservation). *If* $M : \Gamma \vdash A$ *and* $M \longrightarrow M'$, *then* $M' : \Gamma \vdash A$.

We prove the Progress theorem by defining an Agda function named progress that takes a closed, well-typed term M and either (1) returns a redex inside M, (2) identifies M as a value, or (3) identifies M as an exception.

Theorem 14 (Progress). *If* $M : \emptyset \vdash A$, *then*

1. $M \longrightarrow M'$ *for some* M',
2. $\mathsf{Value}\, M$, *or*
3. $M \equiv \mathtt{blame}\ \ell$.

Proof sketch. To convey the flavor of the proof, we detail the cases for function application and cast application. The reader may read the proofs of the other cases in the Agda development.

Case $M_1\, M_2$ The induction hypothesis for M_1 yields the following subcases.

Subcase $M_1 \longrightarrow M_1'$**.** By rule (ξ), we conclude that

$$M_1\, M_2 \longrightarrow M_1'\, M_2$$

Subcase $M_1 \equiv \mathtt{blame}\ \ell$**.** By rule ($\xi$-blame), we conclude that

$$(\mathtt{blame}\ \ell)\, M_2 \longrightarrow \mathtt{blame}\ \ell$$

Subcase $\mathsf{Value}\, M_1$**.** The induction hypothesis for M_2 yields three subcases.

Subcase $M_2 \longrightarrow M_2'$**.** By rule (ξ), using $\mathsf{Value}\, M_1$, we conclude that

$$M_1\, M_2 \longrightarrow M_1\, M_2'$$

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

Subcase $M_2 \equiv \texttt{blame}\ \ell$**.** By rule ($\xi$-`blame`), using Value M_1, we conclude that

$$M_1\ (\texttt{blame}\ \ell) \longrightarrow \texttt{blame}\ \ell$$

Subcase Value M_2**.** We proceed by cases on Value M_1, noting it is of function type.

Subcase $M_1 \equiv \lambda M_{11}$**.** By rule ($\beta$), using Value M_2, we conclude that

$$(\lambda M_{11})\, M_2 \longrightarrow M_{11}[M_2]$$

Subcase $M_1 \equiv V\langle c\rangle$ **and** $i : \mathsf{Inert}\, c$**.** The field InertCross$\rightarrow$ of record PreCastStruct gives us a proof x that c is a cross cast. So by rule (`fun-cast`), we have

$$V\langle c\rangle\ M_2 \longrightarrow (V\ M_2\langle \mathsf{dom}\ c\ x\rangle)\langle \mathsf{cod}\ c\ x\rangle$$

Subcase $M_1 \equiv k_1$**.** We proceed by cases on Value M_2, most of which lead to contradictions and are therefore vacuously true. Suppose $M_2 \equiv k_2$. By rule (δ), we conclude that

$$k_1\ k_2 \longrightarrow [\![k_1]\!]\ [\![k_2]\!]$$

Suppose $M_2 \equiv M_{21}\langle c\rangle$ and c is inert. Then c is a cast on base types, which contradicts that c is inert thanks to the baseNotInert field of PreCastStruct.

Case $M\langle c\rangle$ The induction hypothesis for M yields three subcases.

Subcase $M \longrightarrow M'$**.** By rule (ξ), we conclude that

$$M\langle c\rangle \longrightarrow M'\langle c\rangle$$

Subcase $M \equiv \texttt{blame}\ \ell$**.** By rule ($\xi$-`blame`), we conclude that

$$(\texttt{blame}\ \ell)\langle c\rangle \longrightarrow \texttt{blame}\ \ell$$

Subcase Value M**.** Here we use the ActiveOrInert field of the PreCastStruct on the cast c. Suppose c is active, so we have $a : \mathsf{Active}\, c$. By rule (`cast`), using Value M, we conclude that

$$M\langle c\rangle \longrightarrow \mathsf{applyCast}\ M\ c\ a$$

Suppose c is inert. Then, we conclude that Value $M\langle c\rangle$. □

3.9 The CC′(⇒) variant

Siek *et al.* (2015*b*), in their Isabelle mechanization of the GTLC and the dynamic gradual guarantee, make a syntactic distinction between cast applications as unevaluated expressions versus cast applications that are part of values. (The paper glosses over this detail.) In particular, they introduce a term constructor named Wrap for casts between function types that are part of a value and another term constructor named Inj for casts into ? that are part of a value. The reason they make this distinction is to enable a smaller simulation relation

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

in their proof of the dynamic gradual guarantee.[1] To aid the proof the Dynamic Gradual Guarantee in Section 3.11, we introduce a variant of $CC(\Rightarrow)$, named $CC'(\Rightarrow)$ (Figure 9), that makes a syntactic distinction between arbitrary casts and inert casts. We write inert cast application as $M\langle\!\langle c \rangle\!\rangle$. Instead of making the further distinction as in Wrap and Inj, we rely on the source and target types of the casts to distinguish those cases. With the addition of inert casts, we change the definition of Value M, removing the `Vcast` rule and replacing it with `Vwrap`, which states that $V\langle\!\langle c \rangle\!\rangle$ is a value if c is inert. We also change the reduction rules (`fun-cast`), (`fst-cast`), (`snd-cast`), and (`case-cast`) to eliminate values of the form $V\langle\!\langle c \rangle\!\rangle$ instead of $V\langle c \rangle$. This change regarding inert casts does not affect the observable behavior of programs with respect to $CC(\Rightarrow)$.

We also change `case` expressions to include the variable binding, so all of the reduction rules for `case` must be changed. We change the rules (β-`caseL`), (β-`caseR`), and (`case-cast`) to the rules (β-`caseL-alt`), (β-`caseR-alt`), and (`case-cast-alt`), which use substitution. This change allows us to avoid relating terms to their η-expansions. We conjecture that this change is not necessary for the proof but would require a more complex simulation relation to take η-expansion into account. The change to (`case-cast`) is an observable difference. Consider an expression of the form:

$$\texttt{case}\,(\mathsf{inl}\ V)\langle\!\langle c \rangle\!\rangle\ M\ N$$

Suppose variable `Z` does not occur in M and the cast $(\mathsf{inl}\ c\ x)$ always fails. With (`case-cast`), there is an error but with (`case-cast-alt`) there is not.

3.10 Blame-Subtyping Theorem for $CC'(\Rightarrow)$

Recall that the Blame-Subtyping Theorem (Siek *et al.*, 2015b) states that a cast from A to B cannot be blamed (cause a runtime error), if the source type is a subtype of the target type, that is, $A <: B$. We identify a cast with a blame label ℓ. Thanks to the Blame-Subtyping Theorem, a programmer or static analyzer can inspect the source type, the target type, and the blame label of a cast and decide whether the cast is entirely safe or whether it might cause a runtime error. During execution, casts with different labels can be composed (see Section 6), producing casts that contain multiple blame labels. So more generally, a cast is safe w.r.t a label ℓ if the cast will not give rise to a runtime error labeled ℓ.

In this section, we develop a parameterized proof of the Blame-Subtyping Theorem for the Parameterized Cast Calculus $CC'(\Rightarrow)$ following the approach of Wadler & Findler (2009). The idea is to use a preservation-style proof but preserve cast safety instead of typing. However, the appropriate subtyping relation and definition of safety for casts depends on the representation and semantics of casts (Siek *et al.*, 2009), which is a parameter of $CC'(\Rightarrow)$. Thus, the definition of safety is a parameter of the proof: the CastBlameSafe predicate. The proof is not parameterized on the subtyping relation, clients of the proof typically use subtyping to define the CastBlameSafe predicate.

We require that applying a blame-safe cast to a value does not raise a runtime error, that is, if c is blame-safe for ℓ, then $\mathsf{applyCast}\ V\ c\ a \neq \texttt{blame}\ \ell$. However, to handle the application of casts to higher-order values, we must generalize this requirement. The output

[1] Wadler & Findler (2009) make similar distinctions for different reasons in the Blame Calculus.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$\boxed{M : \Gamma \vdash' A}$

$$\texttt{case} : \frac{\Gamma \vdash' A_1 + A_2 \qquad \Gamma \cdot A_1 \vdash' B \qquad \Gamma \cdot A_2 \vdash' B}{\Gamma \vdash' B}$$

$$-\langle\!\langle c \rangle\!\rangle : \frac{\Gamma \vdash' A}{\Gamma \vdash' B}\ c : A \Rightarrow B, \mathsf{Inert}\ c$$

$\boxed{\mathsf{Value}\ : (\Gamma \vdash' A) \to \mathsf{Set}}$

$$\texttt{Vwrap} : \frac{\mathsf{Value}\ M}{\mathsf{Value}\ (M\langle\!\langle c \rangle\!\rangle)}\ \mathsf{Inert}\ c$$

$\boxed{\mathsf{rename}\ \rho\ M}$

$$\mathsf{rename}\ \rho\ (\texttt{case}\ L\ M\ N) = \texttt{case}\ (\mathsf{rename}\ \rho\ L)\ (\mathsf{rename}\ (\mathsf{ext}\ \rho)\ M)\ (\mathsf{rename}\ (\mathsf{ext}\ \rho)\ N)$$
$$\mathsf{rename}\ \rho\ (M\langle\!\langle c \rangle\!\rangle) = (\mathsf{rename}\ \rho\ M)\langle\!\langle c \rangle\!\rangle$$

$\boxed{\mathsf{subst}\ \sigma\ M}$

$$\mathsf{subst}\ \rho\ (\texttt{case}\ L\ M\ N) = \texttt{case}\ (\mathsf{subst}\ \rho\ L)\ (\mathsf{subst}\ (\mathsf{exts}\ \rho)\ M)\ (\mathsf{subst}\ (\mathsf{exts}\ \rho)\ N)$$
$$\mathsf{subst}\ \rho\ (M\langle\!\langle c \rangle\!\rangle) = (\mathsf{subst}\ \rho\ M)\langle\!\langle c \rangle\!\rangle$$

$\boxed{M \longrightarrow N}$

$$\frac{}{V\langle c \rangle \longrightarrow V\langle\!\langle c \rangle\!\rangle}\ \mathsf{Inert}\ c \qquad (\texttt{wrap})$$

$$\frac{}{V\langle\!\langle c \rangle\!\rangle\ W \longrightarrow (V\ W\langle \mathsf{dom}\ c\ x \rangle)\langle \mathsf{cod}\ c\ x \rangle}\ x : \mathsf{Cross}\ c, \mathsf{Inert}\ c \qquad (\texttt{fun-cast-alt})$$

$$\frac{}{\texttt{fst}\ (V\langle\!\langle c \rangle\!\rangle) \longrightarrow (\texttt{fst}\ V)\langle \mathsf{fst}\ c\ x \rangle}\ x : \mathsf{Cross}\ c, \mathsf{Inert}\ c \qquad (\texttt{fst-cast-alt})$$

$$\frac{}{\texttt{snd}\ (V\langle\!\langle c \rangle\!\rangle) \longrightarrow (\texttt{snd}\ V)\langle \mathsf{snd}\ c\ x \rangle}\ x : \mathsf{Cross}\ c, \mathsf{Inert}\ c \qquad (\texttt{snd-cast-alt})$$

$$\frac{}{\texttt{case}\ (V\langle\!\langle c \rangle\!\rangle)\ M\ N \longrightarrow \texttt{case}\ V\ M'\ N'}\ \begin{array}{l} x : \mathsf{Cross}\ c, \mathsf{Inert}\ c \\ \text{where}\quad M' = (\mathsf{rename}\ (\mathsf{ext}\ \mathbb{S})\ M)[\texttt{`Z}\langle \mathsf{inl}\ c\ x \rangle] \\ \qquad\quad N' = (\mathsf{rename}\ (\mathsf{ext}\ \mathbb{S})\ N)[\texttt{`Z}\langle \mathsf{inr}\ c\ x \rangle] \end{array} \qquad (\texttt{case-cast-alt})$$

$$\frac{}{\texttt{case}\ (\texttt{inl}\ V)\ L\ M \longrightarrow L[V]} \qquad (\beta\texttt{-caseL-alt})$$

$$\frac{}{\texttt{case}\ (\texttt{inr}\ V)\ L\ M \longrightarrow M[V]} \qquad (\beta\texttt{-caseR-alt})$$

Fig. 9. The $CC'(\Rightarrow)$ Variant.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$\boxed{M \text{ safe for } \ell}$

$$\frac{\mathsf{CastBlameSafe}\ c\ \ell \qquad M \text{ safe for } \ell}{(M\langle c\rangle) \text{ safe for } \ell} \qquad \frac{}{(`x) \text{ safe for } \ell}$$

$$\frac{N \text{ safe for } \ell}{(\lambda N) \text{ safe for } \ell} \qquad \frac{L \text{ safe for } \ell \qquad M \text{ safe for } \ell}{(L\ M) \text{ safe for } \ell}$$

$$\frac{}{(\$k) \text{ safe for } \ell} \qquad \frac{M \text{ safe for } \ell \qquad N \text{ safe for } \ell}{(\mathtt{cons}\ M\ N) \text{ safe for } \ell}$$

$$\frac{L \text{ safe for } \ell \qquad M \text{ safe for } \ell \qquad N \text{ safe for } \ell}{(\mathtt{if}\ L\ M\ N) \text{ safe for } \ell}$$

$$\frac{M \text{ safe for } \ell}{(\mathtt{fst}\ M) \text{ safe for } \ell} \qquad \frac{M \text{ safe for } \ell}{(\mathtt{snd}\ M) \text{ safe for } \ell}$$

$$\frac{M \text{ safe for } \ell}{(\mathtt{inl}[B]\ M) \text{ safe for } \ell} \qquad \frac{M \text{ safe for } \ell}{(\mathtt{inr}[A]\ M) \text{ safe for } \ell}$$

$$\frac{L \text{ safe for } \ell \qquad M \text{ safe for } \ell \qquad N \text{ safe for } \ell}{(\mathtt{case}\ L\ M\ N) \text{ safe for } \ell}$$

$$\frac{\ell \neq \ell'}{(\mathtt{blame}\ \ell') \text{ safe for } \ell}$$

Fig. 10. The predicate that a term contains only blame-safe casts.

of applyCast can be any term, so in addition to ruling out $\mathtt{blame}\ \ell$ we also need to rule out terms that contain casts that are not blame-safe for ℓ. Thus, we define a predicate, written M safe for ℓ (Wadler & Findler, 2009), that holds if all the casts in M are blame-safe for ℓ, defined in Figure 10. Most importantly, $\mathtt{blame}\ \ell'$ is safe for ℓ only when $\ell \neq \ell'$. The other important case is the one for casts: $M\langle c\rangle$ is safe for ℓ if CastBlameSafe $c\ \ell$ and M is recursively safe for ℓ.

We also require that CastBlameSafe should be preserved under the operations on casts: dom, cod, fst, snd, inl, and inr.

We capture these requirements in two structures:

1. PreBlameSafe and
2. BlameSafe.

The first structure includes the CastBlameSafe predicate and the fields that preserve blame safety under the casts operations (Figure 11). The second structure includes the requirement that applyCast preserves blame safety.

The structure BlameSafe extends PreBlameSafe and CastStruct. It also includes the following field which, roughly speaking, requires that applying a blame-safe cast to a blame-safe value V results in a blame-safe term.

$$\begin{aligned}\mathsf{applyCast\text{-}pres\text{-}allsafe} : \forall \Gamma A B \ell.\ (V : \Gamma \vdash A) &\to \mathsf{Value}\ V \to (c : A \Rightarrow B)\\ &\to (a : \mathsf{Active}\ c) \to \mathsf{CastBlameSafe}\ c\ \ell\\ &\to V \text{ safe for } \ell\\ &\to (\mathsf{applyCast}\ V\ c\ a) \text{ safe for } \ell\end{aligned}$$

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

CastBlameSafe : $\forall AB.\, A \Rightarrow B \to \mathsf{Label} \to \mathsf{Set}$
Predicate for blame-safe casts, i.e. casts that never cause runtime errors.

domBlameSafe : $\forall A_1 A_2 B_1 B_2.\, (c : (A_1 \to A_2) \Rightarrow (B_1 \to B_2)) \to \mathsf{CastBlameSafe}\ c\ \ell \to (x : \mathsf{Cross}\ c) \to \mathsf{CastBlameSafe}\ (\mathsf{dom}\ c\ x)\ \ell$
Given a cross cast between function types that is blame-safe, domBlameSafe returns a proof that the cast between the domain types is blame-safe.

codBlameSafe : $\forall A_1 A_2 B_1 B_2.\, (c : (A_1 \to A_2) \Rightarrow (B_1 \to B_2)) \to \mathsf{CastBlameSafe}\ c\ \ell \to (x : \mathsf{Cross}\ c) \to \mathsf{CastBlameSafe}\ (\mathsf{cod}\ c\ x)\ \ell$
Similar to the above but for the codomain.

fstBlameSafe : $\forall A_1 A_2 B_1 B_2.\, (c : (A_1 \times A_2) \Rightarrow (B_1 \times B_2)) \to \mathsf{CastBlameSafe}\ c\ \ell \to (x : \mathsf{Cross}\ c) \to \mathsf{CastBlameSafe}\ (\mathsf{fst}\ c\ x)\ \ell$
Given a cross cast between product types that is blame-safe, fstBlameSafe returns a proof that the cast between the first components of the pair is blame-safe.

sndBlameSafe : $\forall A_1 A_2 B_1 B_2.\, (c : (A_1 \times A_2) \Rightarrow (B_1 \times B_2)) \to \mathsf{CastBlameSafe}\ c\ \ell \to (x : \mathsf{Cross}\ c) \to \mathsf{CastBlameSafe}\ (\mathsf{snd}\ c\ x)\ \ell$
Similar to the above but for the second component of the pair.

inlBlameSafe : $\forall A_1 A_2 B_1 B_2.\, (c : (A_1 + A_2) \Rightarrow (B_1 + B_2)) \to \mathsf{CastBlameSafe}\ c\ \ell \to (x : \mathsf{Cross}\ c) \to \mathsf{CastBlameSafe}\ (\mathsf{inl}\ c\ x)\ \ell$
Given a cross cast between sum types that is blame-safe, inlBlameSafe returns a proof that the cast for the first branch is blame-safe.

inrBlameSafe : $\forall A_1 A_2 B_1 B_2.\, (c : (A_1 + A_2) \Rightarrow (B_1 + B_2)) \to \mathsf{CastBlameSafe}\ c\ \ell \to (x : \mathsf{Cross}\ c) \to \mathsf{CastBlameSafe}\ (\mathsf{inr}\ c\ x)\ \ell$
Similar to the above but for the second component of the sum.

Fig. 11. PreBlameSafe extends PreCastStruct (Section 3.1).

We turn to the proof of blame safety. We first prove that "safe for" is preserved during reduction (Lemma 15). The proof depends on a number of technical lemmas about renaming and substitution. They are omitted here but can be found in the Agda formalization.

Lemma 15 (Preservation of Blame Safety). *If $M : \Gamma \vdash A$, $M' : \Gamma \vdash A$, M safe for ℓ, and $M \longrightarrow M'$, then M' safe for ℓ.*

Using Preservation of Blame Safety (Lemma 15), we prove the following Blame-Subtyping Theorem. The theorem says that if every cast with label ℓ is blame-safe in M, then these casts never fail and ℓ is guaranteed not to be blamed.

Theorem 16 (Blame-Subtyping Theorem). *If $M : \Gamma \vdash A$ and M safe for ℓ, then $\neg(M \longrightarrow^* \texttt{blame}\ \ell)$.*

Proof sketch. By induction on the reduction sequence and by inversion on the "safe for" predicate. □

3.11 Dynamic gradual guarantee

Recall that the dynamic gradual guarantee (Siek *et al.*, 2015b) states that changing type annotations in a program to be more precise should either result in the same observable

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

behavior or, in the event of a conflict between the new type annotation and some other part of the program, the program could produce a runtime cast error. On the other hand, changing the program to be less precise should always result in the same observable behavior.

Our proof of the dynamic gradual guarantee follows the same structure as that of Siek *et al.* (2015*b*). Their proof involves two main lemmas: (1) compilation from GTLC to its cast calculus preserves precision and (2) a more-precise program in the cast calculus simulates any less-precise version of itself. In this article, the proof that compilation preserves precision is in Section 4. Our proof of the simulation is in this section and is parameterized with respect to the cast representation, using two new structures, so that it can be applied to different cast calculi.

The second of those structures, CastStructWithPrecision, consists of five properties that must be proved for each cast representation. The proofs of these properties together account for approximately one half of the total effort of proving the dynamic gradual guarantee for a particular calculus. This places a higher burden on each cast representation compared to the generic proofs of type safety (Section 3.8), blame-subtyping (Section 3.10), and the space efficiency theorem (Section 6.5). So far we have proved that λB satisfies CastStructWithPrecision (Section 5.3.3) but we have not yet had the time for the proofs for the other cast calculi. An interesting open question is whether there exists a better structure that enables more of the proof of the dynamic gradual guarantee to be generic with respect to cast representations.

3.11.1 Precision structures and precision for CC′

The definition of term precision relies on a notion of precision for the cast representation, so precision for casts must be a parameter of the proof. In particular, we define PreCastStructWithPrecision as an extension of PreCastStruct in Figure 12. This structure includes three fields that define the precision relation between two casts and precision between a cast and a type (on the left or right). The record also includes four requirements on those relations that correspond to the precision rules in Figure 9 of Siek *et al.* (2015*b*): $\sqsubseteq \to$ lpit corresponds to the forward direction of the bottom right rule, while lpii$\to \sqsubseteq$, lpit$\to \sqsubseteq$, and lpti$\to \sqsubseteq$ are inversion properties about the top left, bottom right, and bottom left rules, respectively.

We define the precision relation between $CC'(\Rightarrow)$ terms in Figure 13. The rules $\texttt{Cast}\sqsubseteq^C$, $\texttt{CastL}\sqsubseteq^C$, and $\texttt{CastR}\sqsubseteq^C$ correspond to the precision rules for casts in Figure 9 of Siek *et al.* (2015*b*). The rule $\texttt{Wrap}\sqsubseteq^C$ corresponds to the rules p_wrap_wrap and p_inj in the Isabelle mechanization of Siek *et al.* (2015*b*). The side condition that $B = ?$ implies $B' = ?$ ensures that our $\texttt{Wrap}\sqsubseteq^C$ rule handles the same cases as p_inj (where both $B = ?$ and $B' = ?$) and p_wrap_wrap (where neither $B = ?$ nor $B' = ?$). The $\texttt{WrapL}\sqsubseteq^C$ rule corresponds to the rules p_wrap_r and p_injr.[2] The $\texttt{WrapR}\sqsubseteq^C$ rule corresponds to the rule p_wrap_l. The side condition $A \neq ?$ prevents overlap with the $\texttt{Wrap}\sqsubseteq^C$ rule which we explain as follows. If $A = ?$, then by Lemma 3 the term M would be an injection and therefore the $\texttt{Wrap}\sqsubseteq^C$ rule would apply. (For the same reason, there is no p_injl rule in

[2] The precision ordering in Siek *et al.* (2015*b*) is flipped with respect to our which explains the switching of left versus right.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$\langle\!\langle - \rangle\!\rangle \sqsubseteq \langle\!\langle - \rangle\!\rangle : \forall c\, c'.\, \mathsf{Inert}\, c \to \mathsf{Inert}\, c' \to \mathsf{Set}$
The precision relation between two inert casts.

$\langle\!\langle - \rangle\!\rangle \sqsubseteq - : \forall c.\, \mathsf{Inert}\, c \to \mathcal{T} \to \mathsf{Set}$
The precision relation between an inert cast and a type.

$- \sqsubseteq \langle\!\langle - \rangle\!\rangle : \forall c'.\, \mathcal{T} \to \mathsf{Inert}\, c' \to \mathsf{Set}$
The precision relation between a type and an inert cast.

$\sqsubseteq\!\to\!\mathsf{Ipit}$:

$$\forall A B A'.\, (c : A \Rightarrow B) \to (i : \mathsf{Inert}\, c) \to A \sqsubseteq A' \to B \sqsubseteq A' \to \langle\!\langle i \rangle\!\rangle \sqsubseteq A'$$

If the source and target of an inert cast are less precise than another type, then so is the inert cast.

$\mathsf{Ipii}\!\to\!\sqsubseteq$:

$$\forall A B A' B'.\, (c : A \Rightarrow B) \to (c' : A' \Rightarrow B') \to (i : \mathsf{Inert}\, c) \to (i' : \mathsf{Inert}\, c') \to \langle\!\langle i \rangle\!\rangle \sqsubseteq \langle\!\langle i' \rangle\!\rangle \to (A \sqsubseteq A') \times (B \sqsubseteq B')$$

If an inert cast is less precise than another, then its source and target are too.

$\mathsf{Ipit}\!\to\!\sqsubseteq$:

$$\forall A A' B.\, (c : A \Rightarrow B) \to (i : \mathsf{Inert}\, c) \to \langle\!\langle i \rangle\!\rangle \sqsubseteq A' \to (A \sqsubseteq A') \times (B \sqsubseteq A')$$

If an inert cast is less precise than a type, then so are its source and target.

$\mathsf{Ipti}\!\to\!\sqsubseteq$:

$$\forall A A' B'.\, (c' : A' \Rightarrow B') \to (i' : \mathsf{Inert}\, c') \to A \sqsubseteq \langle\!\langle i' \rangle\!\rangle \to (A \sqsubseteq A') \times (A \sqsubseteq B')$$

If a type is less precise than an inert cast, then it is less precise than the source and target.

Fig. 12. PreCastStructWithPrecision extends PreCastStruct.

the mechanization of Siek *et al*. (2015*b*).) The term `blame` ℓ is treated as more precise than any term, just as in Siek *et al*. (2015*b*). The rest of the rules are equivalent to the precision rules for GTLC (Figure 4).

There are five technical lemmas of Siek *et al*. (2015*b*) used in the simulation proof that directly involve the cast representation, so those too become structure fields. We extend CastSruct to create the structure CastStructWithPrecision in Figure 14

3.11.2 Proof of the simulation

This section is parameterized by CastStructWithPrecision and contains two lemmas that lead up to the proof of the simulation. The first lemma states that a term that is less precise than a value will catch up by reducing to a value.

Lemma 17 (Catchup to Value). *Suppose* $M : \Gamma \vdash A$ *and* $V' : \Gamma' \vdash A'$. *If* $M \sqsubseteq^C V'$, *then* $M \longrightarrow^* V$ *and* $V \sqsubseteq^C V'$ *for some* V.

Proof sketch. The proof is by induction on the precision relation $M \sqsubseteq^C V'$. All the cases are trivial, using induction hypotheses about subterms, except the one case for CastL. Suppose $M \equiv N\langle c \rangle$. By the induction hypothesis for N, we have $N \longrightarrow^* V$ for some V and

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$\boxed{\sigma \sqsubseteq^s \sigma'}$

$$\frac{M \sqsubseteq^C M'}{\mathsf{substZero}\ M \sqsubseteq^s \mathsf{substZero}\ M'} \qquad \frac{\sigma \sqsubseteq^s \sigma'}{\mathsf{exts}\ \sigma \sqsubseteq^s \mathsf{exts}\ \sigma'}$$

$\boxed{M \sqsubseteq^C M'}$

$$\frac{}{\$k \sqsubseteq^C \$k} \qquad \frac{}{`x \sqsubseteq^C `x}$$

$$\frac{A \sqsubseteq A' \quad N \sqsubseteq^C N'}{\lambda\, N \sqsubseteq^C \lambda\, N'} N : \Gamma \cdot A \vdash B, N' : \Gamma' \cdot A' \vdash B'$$

$$\frac{L \sqsubseteq^C L' \quad M \sqsubseteq^C M'}{L\ M \sqsubseteq^C L'\ M'} \qquad \frac{L \sqsubseteq^C L' \quad M \sqsubseteq^C M' \quad N \sqsubseteq^C N'}{\mathtt{if}\ L\ M\ N \sqsubseteq^C \mathtt{if}\ L'\ M'\ N'}$$

$$\frac{M \sqsubseteq^C M' \quad N \sqsubseteq^C N'}{\mathtt{cons}\ M\ N \sqsubseteq^C \mathtt{cons}\ M'\ N'} \qquad \frac{M \sqsubseteq^C M'}{\pi_1\ M \sqsubseteq^C \pi_1\ M'} \qquad \frac{M \sqsubseteq^C M'}{\pi_2\ M \sqsubseteq^C \pi_2\ M'}$$

$$\frac{B \sqsubseteq B' \quad M \sqsubseteq^C M'}{\mathtt{inl}[B]\ M \sqsubseteq^C \mathtt{inl}[B']\ M'} \qquad \frac{A \sqsubseteq A' \quad M \sqsubseteq^C M'}{\mathtt{inr}[A]\ M \sqsubseteq^C \mathtt{inr}[A']\ M'}$$

$$\frac{A_1 \sqsubseteq A_1' \quad A_2 \sqsubseteq A_2' \quad L \sqsubseteq^C L' \quad M \sqsubseteq^C M' \quad N \sqsubseteq^C N'}{\mathtt{case}\ L\ M\ N \sqsubseteq^C \mathtt{case}\ L'\ M'\ N'} \quad L : \Gamma \vdash A_1 + A_2, L' : \Gamma' \vdash A_1' + A_2'$$

$$\mathtt{Cast}\sqsubseteq^C : \frac{A \sqsubseteq A' \quad B \sqsubseteq B' \quad M \sqsubseteq^C M'}{M\langle c\rangle \sqsubseteq^C M'\langle c'\rangle} c : A \Rightarrow B, c' : A' \Rightarrow B'$$

$$\mathtt{CastL}\sqsubseteq^C : \frac{A \sqsubseteq A' \quad B \sqsubseteq A' \quad M \sqsubseteq^C M'}{M\langle c\rangle \sqsubseteq^C M'} c : A \Rightarrow B, M' : \Gamma' \vdash A'$$

$$\mathtt{CastR}\sqsubseteq^C : \frac{A \sqsubseteq A' \quad A \sqsubseteq B' \quad M \sqsubseteq^C M'}{M \sqsubseteq^C M'\langle c'\rangle} M : \Gamma \vdash A, c' : A' \Rightarrow B'$$

$$\mathtt{Wrap}\sqsubseteq^C : \frac{\langle\!\langle i\rangle\!\rangle \sqsubseteq \langle\!\langle i'\rangle\!\rangle \quad M \sqsubseteq^C M'}{M\langle\!\langle i\rangle\!\rangle \sqsubseteq^C M'\langle\!\langle i'\rangle\!\rangle} i : A \Rightarrow B, i' : A' \Rightarrow B', (B = ?) \to (B' = ?)$$

$$\mathtt{WrapL}\sqsubseteq^C : \frac{\langle\!\langle i\rangle\!\rangle \sqsubseteq A' \quad M \sqsubseteq^C M'}{M\langle\!\langle i\rangle\!\rangle \sqsubseteq^C M'} M' : \Gamma' \vdash A'$$

$$\mathtt{WrapR}\sqsubseteq^C : \frac{A \sqsubseteq \langle\!\langle i'\rangle\!\rangle \quad M \sqsubseteq^C M'}{M \sqsubseteq^C M'\langle\!\langle i'\rangle\!\rangle} M : \Gamma \vdash A, A \neq ?$$

$$\frac{A \sqsubseteq A'}{M \sqsubseteq^C \mathtt{blame}\ \ell} M : \Gamma \vdash A, \mathtt{blame}\ \ell : \Gamma' \vdash A'$$

Fig. 13. Precision between $CC'(\Rightarrow)$ terms and between substitutions.

$V \sqsubseteq^C V'$. For $V\langle c\rangle$, we case on whether cast c is inert or active: if inert, then $V\langle c\rangle \longrightarrow V\langle\!\langle i\rangle\!\rangle$ where $i : \mathsf{Inert}\ c$ by rule `wrap`; if active, then it is proved by the field applyCast-catchup of CastStructWithPrecision (Figure 14). □

Next, we prove that substitution preserves precision.

Lemma 18 (Substitution Preserves $\sqsubseteq^C$). *If* $\sigma \sqsubseteq^s \sigma'$ *and* $N \sqsubseteq^C N'$, *then* $\mathsf{subst}\ \sigma\ N \sqsubseteq^C \mathsf{subst}\ \sigma'\ N'$.

Proof sketch. The proof is by induction on the precision relation $N \sqsubseteq^C N'$. □

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

applyCast-catchup :

$$\begin{aligned}\forall \Gamma\Gamma' AA'B.\,(V : \Gamma \vdash A) &\to (V' : \Gamma' \vdash A') \to (c : A \Rightarrow B)\\ &\to (a : \mathsf{Active}\ c) \to \mathsf{Value}\ V \to \mathsf{Value}\ V'\\ &\to A \sqsubseteq A' \to B \sqsubseteq A' \to V \sqsubseteq^C V'\\ &\to \exists W.(\mathsf{Value}\ W) \times (\mathsf{applyCast}\ V\ c\ a \longrightarrow^* W) \times (W \sqsubseteq^C V')\end{aligned}$$

sim-cast :

$$\begin{aligned}\forall AA'BB'.\,(V : \emptyset \vdash A) &\to (V' : \emptyset \vdash A') \to (c : A \Rightarrow B) \to (c' : A' \Rightarrow B')\\ &\to \mathsf{Value}\ V \to \mathsf{Value}\ V' \to (a' : \mathsf{Active}\ c')\\ &\to A \sqsubseteq A' \to B \sqsubseteq B' \to V \sqsubseteq^C V'\\ &\to \exists N.(V\langle c\rangle \longrightarrow^* N) \times (N \sqsubseteq^C \mathsf{applyCast}\ V'\ c'\ a')\end{aligned}$$

sim-wrap :

$$\begin{aligned}\forall AA'BB'.\,(V : \emptyset \vdash A) &\to (V' : \emptyset \vdash A') \to (c : A \Rightarrow B) \to (c' : A' \Rightarrow B')\\ &\to \mathsf{Value}\ V \to \mathsf{Value}\ V' \to (i' : \mathsf{Inert}\ c')\\ &\to A \sqsubseteq A' \to B \sqsubseteq B' \to V \sqsubseteq^C V'\\ &\to \exists N.(V\langle c\rangle \longrightarrow^* N) \times (N \sqsubseteq^C V'\langle\!\langle i'\rangle\!\rangle)\end{aligned}$$

castr-cast :

$$\begin{aligned}\forall AA'B.\,(V : \emptyset \vdash A) &\to (V' : \emptyset \vdash A') \to (c' : A' \Rightarrow B')\\ &\to \mathsf{Value}\ V \to \mathsf{Value}\ V' \to (a' : \mathsf{Active}\ c')\\ &\to A \sqsubseteq A' \to A \sqsubseteq B' \to V \sqsubseteq^C V'\\ &\to V \sqsubseteq^C \mathsf{applyCast}\ V'\ c'\ a'\end{aligned}$$

castr-wrap :

$$\begin{aligned}\forall AA'B.\,(V : \emptyset \vdash A) &\to (V' : \emptyset \vdash A') \to (c' : A' \Rightarrow B')\\ &\to \mathsf{Value}\ V \to \mathsf{Value}\ V' \to (i' : \mathsf{Inert}\ c')\\ &\to A \sqsubseteq A' \to A \sqsubseteq B' \to V \sqsubseteq^C V'\\ &\to V \sqsubseteq^C V'\langle\!\langle i'\rangle\!\rangle\end{aligned}$$

Fig. 14. The CastStructWithPrecision structure extends CastStruct.

We come to the proof of the main lemma that a program simulates any more-precise version of itself.

Lemma 19 (Simulation of More Precise Programs). *Suppose* $M_1 : \emptyset \vdash A$ *and* $M_1' : \emptyset \vdash A'$ *and* $M_2' : \emptyset \vdash A'$. *If* $M_1 \sqsubseteq^C M_1'$ *and* $M_1' \longrightarrow M_2'$, *then* $M_1 \longrightarrow^* M_2$ *and* $M_2 \sqsubseteq^C M_2'$ *for some* M_2.

Proof sketch. The proof is by induction on the precision relation $M_1 \sqsubseteq^C M_1'$. The full proof is in Agda; we briefly describe a few cases here.

Application ($M_1 \equiv L\,M$ and $M_1' \equiv L'\,M'$). We case on the reduction $L'\,M' \longrightarrow M_2'$, which yields seven subcases:

Subcase ξ, where frame $F = (\square\ \_)$. By the induction hypothesis about the precision between L and L', there exists L_2 such that $L \longrightarrow^* L_2$ and L_2 satisfies the precision

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

relation. Since multi-step reduction is a congruence, there exists $M_2 \equiv L_2\, M$ that satisfies $M_1 \longrightarrow^* M_2$ and $M_2 \sqsubseteq^C M'_2$.

Subcase ξ, where frame $F = (\_\ \Box)$. By Proposition 17, there exists L_2 such that: 1) L_2 is a value; 2) $L \longrightarrow^* L_2$; 3) $L_2 \sqsubseteq^C L'$. By the induction hypothesis about M and M', there exists N_2 such that $M \longrightarrow^* N_2$ and N_2 satisfies the precision relation. Since multi-step reduction is a congruence and is transitive, there exists $M_2 \equiv L_2\, N_2$ that satisfies $M_1 \longrightarrow^* M_2$ and $M_2 \sqsubseteq^C M'_2$.

Subcase ξ-`blame`, where frame $F = (\Box\ \_)$. $L\, M$ itself satisfies the precision relation since blame is more precise than any term, so there exists $M_2 \equiv L\, M$.

Subcase ξ-`blame`, where frame $F = (\_\ \Box)$. Same as the previous case.

Subcase β. Using Proposition 17 twice, $L\, M \longrightarrow^* V\, W$, where V and W are both values and $V \sqsubseteq^C L'$, $W \sqsubseteq^C M'$. We know that $L' \equiv \lambda\, N'$ for some N' due to β. By induction on the precision relation $V \sqsubseteq^C \lambda\, N'$ and inversion on $\mathsf{Value}\ V$, it further generates two sub subcases:

Sub-subcase: V is `Vλ` Suppose $V \equiv \lambda\, N$, then there exists $M_2 \equiv N[W]$ that satisfies both the reduction (from $V\ W$ by β) and the precision.

Sub-subcase: V is `Vwrap` Suppose $V \equiv V_1\langle\!\langle i \rangle\!\rangle$ for some value V_1 and $i : \mathsf{Inert}\ c$:

$$V_1\langle\!\langle i \rangle\!\rangle\ W \longrightarrow (V_1\ W\langle \mathsf{dom}\ c\rangle)\langle \mathsf{cod}\ c\rangle, \text{ by } \texttt{fun-cast}.$$

By using Proposition 17 and congruence of reduction:

$$(V_1\ W\langle \mathsf{dom}\ c\rangle)\langle \mathsf{cod}\ c\rangle \longrightarrow^* V_1\ W_1\langle \mathsf{cod}\ c\rangle$$

for some value W_1. Note that V_1, and W_1 are now both values, about which by induction hypothesis there exists N such that $V_1\ W_1 \longrightarrow^* N$ and N satisfies the precision relation.

Subcase δ. Similar to the previous case, by repeatedly using Proposition 17 ("catch-up") and the induction hypothesis.

Subcase `fun−cast`. Similar to the previous case, by repeatedly using Proposition 17 ("catch-up") and the induction hypothesis.

Cast ($M_1 \equiv M\langle c\rangle$ and $M'_1 \equiv M'\langle c'\rangle$). We case on the reduction $M'\langle c'\rangle \longrightarrow M'_2$, which yields four subcases:

Subcase ξ. By the induction hypothesis about subterms M and M' and that multi-step reduction is a congruence.

Subcase ξ-`blame`. $M\langle c\rangle$ itself satisfies the precision relation since blame is more precise than any term.

Subcase `cast`. By Proposition 17 and that M' is value, there exists value V such that $M \longrightarrow^* V$ and $V \sqsubseteq^C M'$. Then this case is directly proved by the field sim-cast of CastStructWithPrecision (Figure 14).

Subcase `wrap`. Similar to the previous case, first use Proposition 17 and then use field sim-wrap of CastStructWithPrecision (Figure 14).

Similarly, fields castr-cast and castr-wrap of CastStructWithPrecision are used in the case for `castr`, which is omitted since the idea is the same as the `cast` case described above.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$$\boxed{\begin{array}{l}\mathcal{C}[\![-]\!] : \forall \Gamma\, A.\, (M : \texttt{Term}) \to \Gamma \vdash_G M : A \to \Gamma \vdash A \\ \mathcal{C}'[\![-]\!] : \forall \Gamma\, A.\, (M : \texttt{Term}) \to \Gamma \vdash_G M : A \to \Gamma \vdash' A\end{array}}$$

$$\begin{aligned}
\mathcal{C}[\![\$k]\!]\ (\vdash \texttt{lit} = k \\
\mathcal{C}[\![x]\!]\ (\vdash \texttt{var} = x \\
\mathcal{C}[\![\lambda[A]\ M]\!]\ (\vdash \texttt{lam}\ \vdash M) &= \lambda\, (\mathcal{C}[\![M]\!]\ \vdash M) \\
\mathcal{C}[\![(L\ M)_\ell]\!]\ (\vdash \texttt{app}\ \vdash L\ \vdash M) &= (\mathcal{C}[\![L]\!]\ \vdash L)\langle c\rangle\ (\mathcal{C}[\![M]\!]\ \vdash M)\langle d\rangle \\
&\quad \text{where} \\
&\quad \Gamma \vdash_G L : A, \Gamma \vdash_G M : B, A \triangleright A_1 \to A_2, \\
&\quad c = (\!|A \Rightarrow (A_1 \to A_2)|\!)^\ell, d = (\!|B \Rightarrow A_1|\!)^\ell \\
\mathcal{C}[\![\texttt{if}_\ell\ L\ M\ N]\!]\ (\vdash \texttt{if}\ \vdash L\ \vdash M\ \vdash N) &= \texttt{if}\ ((\mathcal{C}[\![L]\!]\ \vdash L)\langle c\rangle) \\
&\qquad ((\mathcal{C}[\![M]\!]\ \vdash M)\langle d_1\rangle)\ ((\mathcal{C}[\![N]\!]\ \vdash N)\langle d_2\rangle) \\
&\quad \text{where} \\
&\quad \Gamma \vdash_G L : A, \Gamma \vdash_G M : B_1, \Gamma \vdash_G N : B_2, \\
&\quad c = (\!|A \Rightarrow \texttt{Bool}|\!)^\ell, \\
&\quad d_1 = (\!|B_1 \Rightarrow B_1 \sqcup B_2|\!)^\ell, d_2 = (\!|B_2 \Rightarrow B_1 \sqcup B_2|\!)^\ell \\
\mathcal{C}[\![\texttt{cons}\ M\ N]\!]\ (\vdash \texttt{cons}\ \vdash M\ \vdash N) &= \texttt{cons}\ (\mathcal{C}[\![M]\!]\ \vdash M)\ (\mathcal{C}[\![N]\!]\ \vdash N) \\
\mathcal{C}[\![\pi_i^\ell\ M]\!]\ (\vdash \pi_i\ \vdash M) &= \pi_i\ ((\mathcal{C}[\![M]\!]\ \vdash M)\langle c\rangle) \\
&\quad \text{where} \\
&\quad \Gamma \vdash_G M : A, A \triangleright A_1 \times A_2, c = (\!|A \Rightarrow A_1 \times A_2|\!)^\ell \\
\mathcal{C}[\![\texttt{inl}[B]\ M]\!]\ (\vdash \texttt{inl}\ \vdash M) &= \texttt{inl}\ (\mathcal{C}[\![M]\!]\ \vdash M) \\
\mathcal{C}[\![\texttt{inr}[A]\ M]\!]\ (\vdash \texttt{inr}\ \vdash M) &= \texttt{inr}\ (\mathcal{C}[\![M]\!]\ \vdash M) \\
\mathcal{C}[\![\texttt{case}_\ell[B_1, C_1]\ L\ M\ N]\!]\ (\vdash \texttt{case}\ \vdash L\ \vdash M\ \vdash N) &= \texttt{case}\ (\mathcal{C}[\![L]\!]\ \vdash L)\langle c\rangle \\
&\qquad (\lambda(\mathcal{C}[\![M]\!]\ \vdash M)\langle d_1\rangle)\ (\lambda(\mathcal{C}[\![N]\!]\ \vdash N)\langle d_2\rangle) \\
&\quad \text{where} \\
&\quad \Gamma \vdash_G L : A, \Gamma \cdot B_1 \vdash_G M : B_2, \Gamma \cdot C_1 \vdash_G N : C_2 \\
&\quad c = (\!|A \Rightarrow B_1 + C_1|\!)^\ell \\
&\quad d_1 = (\!|B_2 \Rightarrow B_2 \sqcup C_2|\!)^\ell, d_2 = (\!|C_2 \Rightarrow B_2 \sqcup C_2|\!)^\ell \\
\mathcal{C}'[\![\texttt{case}_\ell[B_1, C_1]\ L\ M\ N]\!]\ (\vdash \texttt{case}\ \vdash L\ \vdash M\ \vdash N) &= \texttt{case}\ (\mathcal{C}[\![L]\!]\ \vdash L)\langle c\rangle \\
&\qquad (\mathcal{C}[\![M]\!]\ \vdash M)\langle d_1\rangle\ (\mathcal{C}[\![N]\!]\ \vdash N)\langle d_2\rangle
\end{aligned}$$

Fig. 15. Compilation from GTLC to $CC(\Rightarrow)$ and $CC'(\Rightarrow)$.

Wrap ($M_1 \equiv M\langle\!\langle i\rangle\!\rangle$ and $M_1' \equiv M'\langle i'\rangle$). We case on the reduction $M'\langle\!\langle i'\rangle\!\rangle \longrightarrow M_2'$, which yields two subcases:

Subcase ξ. By the induction hypothesis about subterms M and M' and that multi-step reduction is a congruence.

Subcase ξ-`blame`. $M\langle\!\langle i\rangle\!\rangle$ itself satisfies the precision relation since blame is more precise than any term. □

4 Compilation of GTLC to $CC(\Rightarrow)$ and $CC(\Rightarrow)'$

This section is parameterized by the cast representation $\Rightarrow$ and by a cast constructor, written $(\!|A \Rightarrow B|\!)^\ell$, that builds a cast from a source type A, target type B, blame label ℓ, and (implicitly) a proof that A and B are consistent.

The compilation functions $\mathcal{C}[\![-]\!]$ and $\mathcal{C}'[\![-]\!]$ are defined in Figure 15 and map a well-typed term of the GTLC to a well-typed term of the Parameterized Cast Calculus $CC(\Rightarrow)$ or $CC'(\Rightarrow)$, respectively. The two functions are the same except in the equation for `case`, so we only show that equation for $\mathcal{C}'[\![-]\!]$. The Agda type signatures of the compilation functions ensure that they are type preserving.

Lemma 20 (Compilation Preserves Types).

1. *If* $\Gamma \vdash_G M : A$, *then* $\mathcal{C}[\![M]\!] : \Gamma \vdash A$.
2. *If* $\Gamma \vdash_G M : A$, *then* $\mathcal{C}'[\![M]\!] : \Gamma \vdash' A$.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

The compilation functions also preserve precision, which is an important lemma in the proof of the dynamic gradual guarantee. This lemma is parameterized over the PreCastStructWithPrecision structure (Figure 12).

Lemma 21 (Compilation Preserves Precision). *Suppose* $\Gamma \vdash_G M : A$ *and* $\Gamma' \vdash_G M' : A'$. *If* $\Gamma \sqsubseteq \Gamma'$ *and* $M \sqsubseteq^G M'$, *then* $C'[\![M]\!] \sqsubseteq^C C'[\![M']\!]$ *and* $A \sqsubseteq A'$.

5 A half-dozen cast calculi

We begin to reap the benefits of creating the Parameterized Cast Calculus by instantiating it with six different implementations of CastStruct to produce six cast calculi.

5.1 Partially Eager "D" casts with Active cross casts (EDA)

The cast calculus defined in this section corresponds to the original one of Siek & Taha (2006*a*), although their presentation used a big-step semantics instead of a reduction semantics and did not include blame tracking. In the nomenclature of Siek *et al*. (2009), this calculus is partially eager and uses the "D" blame tracking strategy. As we shall see shortly, this calculus categorizes cross casts as active, so we refer to this cast calculus as the partially eager "D" cast calculus with active cross casts (EDA).

We define the EDA calculus and prove that it is type-safe and blame-safe by instantiating the meta-theory of the Parameterized Cast Calculus. This requires that we define instances of the structures PreCastStruct, CastStruct, PreBlameSafe, and BlameSafe.

We begin by defining the cast representation type $\Rightarrow$ as a data type with a single constructor also named $\Rightarrow$ that takes two types, a blame label, and a proof of consistency:

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

$$A \Rightarrow^\ell B : \frac{}{A \Rightarrow B}\ A \sim B$$

The cast constructor is defined trivially as follows:

$$(\!|A \Rightarrow B|\!)^\ell = (A \Rightarrow^\ell B)$$

(Recall that the cast constructor is used during compilation in Section 4.)

5.1.1 Reduction semantics and type safety

We categorize the casts whose target type is ? as inert casts.

$\boxed{\mathsf{Inert}\ c}$

$$\frac{}{\mathsf{Inert}\ (A \Rightarrow^\ell ?)}\ A \neq ?$$

We categorize casts between function, pair, and sum types as cross casts.

$\boxed{\mathsf{Cross}\ c}$

$$\mathtt{Cross}\otimes : \frac{}{\mathsf{Cross}\ (A \otimes B \Rightarrow^\ell C \otimes D)}\ \otimes \in \{\rightarrow, \times, +\}$$

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

We categorize the identity, projection, and cross casts as active.

$\boxed{\mathsf{Active}\ c}$

$$\texttt{ActId} : \frac{}{\mathsf{Active}\ (a \Rightarrow^{\ell} a)} \qquad \texttt{ActProj} : \frac{}{\mathsf{Active}\ (? \Rightarrow^{\ell} B)}\ B \neq ?$$

$$\texttt{ActCross} : \frac{\mathsf{Cross}\ c}{\mathsf{Active}\ c}$$

Lemma 22. *For any types A and B, $c : A \Rightarrow B$ is either an active or inert cast.*

Proof. The cast c must be of the form $A \Rightarrow^{\ell} B$ where $A \sim B$. We proceed by cases on $A \sim B$.

Case $A \sim ?$ If $A \equiv ?$, then c is active by `ActId`. Otherwise c is inert.
Case $? \sim B$ If $B \equiv ?$, then c is active by `ActId`. Otherwise c is active by `ActProj`.
Case $b \sim b$ c is active by `ActId`.
Case $A \otimes B \sim A' \otimes B'$ where $\otimes \in \{\rightarrow, \times, +\}$. c is active by `ActCross` and `Cross`$\otimes$. □

Next we show that inert casts into non-atomic types are cross casts.

Lemma 23. *If $c : A \Rightarrow (B \otimes C)$ and* Inert *c, then* Cross *c and $A \equiv D \otimes E$ for some D and E.*

Proof. There are no inert casts whose target type is $B \otimes C$ (it must be ?), so this lemma is vacuously true. □

Continuing on the topic of cross casts, we define dom, cod, etc. as follows. (The second parameter x is the evidence that the first parameter is a cross cast.)

$$\begin{aligned}
\mathsf{dom}\ (A \rightarrow B \Rightarrow^{\ell} C \rightarrow D)\ x &= C \Rightarrow^{\ell} A \\
\mathsf{cod}\ (A \rightarrow B \Rightarrow^{\ell} C \rightarrow D)\ x &= B \Rightarrow^{\ell} D \\
\mathsf{fst}\ (A \times B \Rightarrow^{\ell} C \times D)\ x &= A \Rightarrow^{\ell} C \\
\mathsf{snd}\ (A \times B \Rightarrow^{\ell} C \times D)\ x &= B \Rightarrow^{\ell} D \\
\mathsf{inl}\ (A + B \Rightarrow^{\ell} C + D)\ x &= A \Rightarrow^{\ell} C \\
\mathsf{inr}\ (A + B \Rightarrow^{\ell} C + D)\ x &= B \Rightarrow^{\ell} D
\end{aligned}$$

We check that a cast to a base type is not inert.

Lemma 24. *A cast $c : A \Rightarrow b$ is not inert.*

Proof. This is easy to verify because $b \neq ?$. □

Proposition 25. *The EDA calculus is an instance of the* PreCastStruct *structure.*

We import and instantiate the definitions and lemmas from Section 3.2 (values and frames) and Section 3.3 (eta-like reduction rules).

Next we define the applyCast function by cases on the proof that the cast is active. The case below for `ActProj` relies on Lemma 3 to know that the term is of the form

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$$\frac{}{A <: ?} \qquad \frac{}{b <: b} \qquad \frac{C <: A \quad B <: D}{A \to B <: C \to D}$$

$$\frac{A <: C \quad B <: D}{A \times B <: C \times D} \qquad \frac{A <: C \quad B <: D}{A + B <: C + D}$$

Fig. 16. Subtyping for "D" blame tracking.

$M\langle A \Rightarrow^{\ell_1} ?\rangle$.

$$\mathsf{applyCast} : \forall \Gamma AB.\, (M : \Gamma \vdash A) \to \mathsf{Value}\, M \to (c : A \Rightarrow B) \to \mathsf{Active}\, c \to \Gamma \vdash B$$

$$\begin{array}{llllll} \mathsf{applyCast} & M & v & (A \Rightarrow^{\ell} A) & \texttt{ActId} & = M \\ \mathsf{applyCast} & M\langle A \Rightarrow^{\ell_1} ?\rangle & v & (? \Rightarrow^{\ell_2} B) & \texttt{ActProj} & = \begin{cases} M\langle A \Rightarrow^{\ell_2} B\rangle & \text{if } A \sim B \\ \texttt{blame}\, \ell_2 & \text{otherwise} \end{cases} \\ \mathsf{applyCast} & M & v & c & \texttt{Act}\otimes & = \mathsf{eta}\otimes\, M\, c\, \texttt{Cross}\otimes \end{array}$$

Proposition 26. *The EDA calculus is an instance of the* CastStruct *structure.*

We instantiate from $CC(\Rightarrow)$ the reduction semantics (Section 3.6) and proof of type safety (Section 3.8) to obtain the following definition and results.

Definition 27 (Reduction for EDA). *The reduction relation $M \longrightarrow N$ for the EDA calculus is the reduction relation of $CC(\Rightarrow)$ instantiated with EDA's instance of* CastStruct.

Corollary 28 (Preservation for EDA). *If $M : \Gamma \vdash A$ and $M \longrightarrow M'$, then $M' : \Gamma \vdash A$.*

Corollary 29 (Progress for EDA). *If $M : \emptyset \vdash A$, then*

1. *$M \longrightarrow M'$ for some M',*
2. *$\mathsf{Value}\, M$, or*
3. *$M \equiv \texttt{blame}\, \ell$.*

Let EDA′ be the variant of EDA obtained by instantiating CC' instead of CC.

5.1.2 Blame-subtyping

Because the EDA′ calculus uses the "D" blame-tracking strategy, the subtyping relation that corresponds to safe casts is the one in Figure 16 where the unknown type ? is the top of the subtyping order.

We define the CastBlameSafe predicate as follows:

$$\frac{A <: B}{\mathsf{CastBlameSafe}\, (A \Rightarrow^{\ell} B)\, \ell} \qquad \frac{\ell \neq \ell'}{\mathsf{CastBlameSafe}\, (A \Rightarrow^{\ell'} B)\, \ell}$$

Lemma 30 (Blame Safety of Cast Operators). *Suppose* $\mathsf{CastBlameSafe}\, c\, \ell$ *and c is a cross cast, that is, $x : \mathsf{Cross}\, c$.*

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

- *If* $x = \texttt{Cross}\,\rightarrow$, *then* CastBlameSafe (dom $c\,x$) ℓ *and* CastBlameSafe (cod $c\,x$) ℓ.
- *If* $x = \texttt{Cross}\times$, *then* CastBlameSafe (fst $c\,x$) ℓ *and* CastBlameSafe (snd $c\,x$) ℓ.
- *If* $x = \texttt{Cross}+$, *then* CastBlameSafe (inl $c\,x$) ℓ *and* CastBlameSafe (inr $c\,x$) ℓ.

Proposition 31. *EDA′ is an instance of the* PreBlameSafe *structure.*

Lemma 32 (applyCast preserves blame safety in EDA′). *If* CastBlameSafe $c\,\ell$ *and* a : Active c *and* V *safe for* ℓ *and* v : Value V, *then* (applyCast $V\,v\,c\,a$) *safe for* ℓ.

Proof. The proof is by cases on a (the cast being active and if a cross cast, cases on the three kinds) and then by cases on CastBlameSafe $c\,\ell$, except that if the cast is an identity cast, then there is no need for casing on CastBlameSafe $c\,\ell$. So there are nine cases to check, but they are all straightforward. □

Proposition 33. *EDA′ is an instance of the* BlameSafe *structure.*

We instantiate Theorem 16 with this BlameSafe instance to obtain the Blame-Subtyping Theorem for EDA′.

Corollary 34 (Blame-Subtyping Theorem for EDA′). *If* $M : \Gamma \vdash A$ *and* M *safe for* ℓ, *then* $\neg(M \longrightarrow^* \texttt{blame}\ \ell)$.

5.2 Partially Eager "D" casts with Inert cross casts (EDI)

Many cast calculi (Wadler & Findler, 2007, 2009; Siek *et al.*, 2009; Siek & Wadler, 2010; Siek *et al.*, 2015a) categorize terms of the form:

$$V\langle A \rightarrow B \Rightarrow C \rightarrow D\rangle$$

as values and then define the reduction rule:

$$V\langle A \rightarrow B \Rightarrow C \rightarrow D\rangle\ W \longrightarrow (V\ (W\langle C \Rightarrow A\rangle))\langle B \Rightarrow D\rangle$$

That is, these calculi categorize cross casts as inert instead of active. In this section, we take this approach for functions, pairs, and sums, but keep everything else the same as in the previous section. We conjecture that this cast calculus is equivalent in observable behavior to EDA. We refer to this calculus as EDI, where the I stands for "inert."

So the cast representation type is again made of a source type, target type, blame label, and consistency proof.

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

$$A \Rightarrow^{\ell} B : \frac{}{A \Rightarrow B}\, A \sim B$$

The cast constructor is defined as follows:

$$(\!|A \Rightarrow B|\!)^{\ell} = (A \Rightarrow^{\ell} B)$$

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

5.2.1 Reduction semantics and type safety

Again, we categorize casts between function, pair, and sum types as cross casts.

$\boxed{\mathsf{Cross}\, c}$

$$\texttt{Cross}\otimes : \frac{\otimes \in \{\rightarrow, \times, +\}}{\mathsf{Cross}\,(A \otimes B \Rightarrow^{\ell} C \otimes D)}$$

But for the inert casts, we include the cross casts this time.

$\boxed{\mathsf{Inert}\, c}$

$$\texttt{InInj} : \frac{A \neq ?}{\mathsf{Inert}\,(A \Rightarrow^{\ell} ?)} \qquad \texttt{InCross} : \frac{\mathsf{Cross}\, c}{\mathsf{Inert}\, c}$$

The active casts include just the identity casts and projections.

$\boxed{\mathsf{Active}\, c}$

$$\texttt{ActId} : \frac{}{\mathsf{Active}\,(a \Rightarrow^{\ell} a)} \qquad \texttt{ActProj} : \frac{B \neq ?}{\mathsf{Active}\,(? \Rightarrow^{\ell} B)}$$

Lemma 35. *For any types A and B, $c : A \Rightarrow B$ is either an active or inert cast.*

Proof. The proof is similar to that of Lemma 22, except the cross casts are categorized as inert instead of active. □

Lemma 36. *If $c : A \Rightarrow (B \otimes C)$ and* Inert *c, then* Cross *c and $A \equiv D \otimes E$ for some D and E.*

Proof. We proceed by cases on Inert c.

Case `InInj`**:** The target type is ?, not $B \otimes C$, so we have a contradiction.
Case `InCross`**:** We have Cross c. By cases on Cross c, we have $A \equiv D \otimes E$ for some D and E. □

The definitions of dom, cod, etc. are exactly the same as in Section 5.1.

Lemma 37. *A cast $c : A \Rightarrow b$ is not inert.*

Proof. The inert casts in this section have a target type of either ? or a non-atomic type $A_1 \otimes A_2$, but not a base type. □

Proposition 38. *The EDI calculus is an instance of the* PreCastStruct *structure.*

Again, we define the applyCast function by cases on the proof that the cast is active, but this time there is one less case to consider (the cross casts). The cases for `ActId` and `ActProj` are the same as in Section 5.1.

$$\mathsf{applyCast} : \forall \Gamma AB.\, (M : \Gamma \vdash A) \rightarrow \mathsf{Value}\, M \rightarrow (c : A \Rightarrow B) \rightarrow \mathsf{Active}\, c \rightarrow \Gamma \vdash B$$

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$$\mathsf{applyCast}\ \ M\ \ v\ \ (a \Rightarrow^{\ell} a)\ \ \mathtt{ActId} \ = \ M$$

$$\mathsf{applyCast}\ \ M\langle A \Rightarrow^{\ell_1} ?\rangle\ \ v\ \ (? \Rightarrow^{\ell_2} B)\ \ \mathtt{ActProj} \ = \ \begin{cases} M\langle A \Rightarrow^{\ell_2} B\rangle & \text{if } A \sim B \\ \mathtt{blame}\ \ell_2 & \text{otherwise} \end{cases}$$

Proposition 39. *The EDI calculus is an instance of the* CastStruct *structure.*

We instantiate from $CC(\Rightarrow)$ the reduction semantics (Section 3.6) and proof of type safety (Section 3.8) for EDI to obtain the following definition and results.

Definition 40. (Reduction for EDI). *The reduction relation $M \longrightarrow N$ for the EDI calculus is the reduction relation of $CC(\Rightarrow)$ instantiated with EDI's instance of* CastStruct.

Corollary 41 (Preservation for EDI). *If $M : \Gamma \vdash A$ and $M \longrightarrow M'$, then $M' : \Gamma \vdash A$.*

Corollary 42 (Progress for EDI). *If $M : \emptyset \vdash A$, then*

1. *$M \longrightarrow M'$ for some M',*
2. *$\mathsf{Value}\, M$, or*
3. *$M \equiv \mathtt{blame}\ \ell$.*

Let EDI′ be the variant of EDI obtained by instantiating CC' instead of CC.

5.2.2 Blame-subtyping

We define the CastBlameSafe predicate for EDI′ exactly the same way as for EDA′, using the subtyping relation of Figure 16.

Because the cast operators for EDI are defined exactly the same as for EDA, Lemma 30 also applies to EDI, that is, the cast operators such as dom preserve blame safety.

Proposition 43. *EDI′ is an instance of the* PreBlameSafe *structure.*

Lemma 44 (applyCast preserves blame safety in EDI′). *If* $\mathsf{CastBlameSafe}\, c\, \ell$ *and* $a : \mathsf{Active}\, c$ *and V safe for ℓ and $v : \mathsf{Value}\, V$, then* $(\mathsf{applyCast}\ V\ v\ c\ a)$ *safe for ℓ.*

Proof. The proof of this lemma is much shorter than the corresponding one for EDA′ because there are fewer casts categorized as active. So there are just three straightforward cases to check. □

Proposition 45. *EDI′ is an instance of the* BlameSafe *structure.*

We instantiate Theorem 16 with this BlameSafe instance to obtain the Blame-Subtyping Theorem for EDI′.

Corollary 46 (Blame-Subtyping Theorem for EDI′). *If $M : \Gamma \vdash A$ and M safe for ℓ, then* $\neg(M \longrightarrow^{*} \mathtt{blame}\ \ell)$.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

5.3 The λB Blame Calculus

This section generates the λB variant (Siek *et al.*, 2015a) of the Blame Calculus (Wadler & Findler, 2009) as an instance of the Parameterized Cast Calculus. It also extends λB in Siek *et al.* (2015*a*) with product types and sum types. We explore treating cross casts on products and sums both as active and inert casts in Section 5.3.1. Compared to the previous sections, the main difference in λB is that all injections and projections factor through *ground types*, defined as follows:

$$\text{Ground Types} \quad G, H ::= b \mid ? \to ? \mid ? \times ? \mid ? + ?$$

The cast representation type consists of a source type, target type, blame label, and consistency proof.

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

$$A \Rightarrow^{\ell} B : \frac{}{A \Rightarrow B} A \sim B$$

The cast constructor is defined as follows:

$$(\!|A \Rightarrow B|\!)^{\ell} = (A \Rightarrow^{\ell} B)$$

5.3.1 Reduction semantics and type safety

We categorize casts between function, pair, and sum types as cross casts.

$\boxed{\mathsf{Cross}\, c}$

$$\texttt{Cross}\otimes : \frac{}{\mathsf{Cross}\,(A \otimes B \Rightarrow^{\ell} C \otimes D)} \otimes \in \{\to, \times, +\}$$

Regarding inert casts in λB, an injection from ground type is inert and a cast between function types is inert. As for casts between product types and sum types, there are two choices, either to make them inert or to make them active. This yields two variants of λB: Variant 1 where all cross casts are inert and Variant 2 where only function casts are inert.

$\boxed{\mathsf{Inert}\, c.\ \text{Variant 1}}$

$$\texttt{InInj} : \frac{}{\mathsf{Inert}\,(G \Rightarrow^{\ell} ?)} \qquad \texttt{InCross} : \frac{\mathsf{Cross}\, c}{\mathsf{Inert}\, c}$$

$\boxed{\mathsf{Inert}\, c.\ \text{Variant 2}}$

$$\texttt{InInj} : \frac{}{\mathsf{Inert}\,(G \Rightarrow^{\ell} ?)} \qquad \texttt{InFun} : \frac{}{\mathsf{Inert}\,(A \to B \Rightarrow^{\ell} C \to D)}$$

The active casts in both λB variations include injections from non-ground type, projections, and identity casts. In Variant 2, we categorize only function casts as active, so compared to Variant 1 we have an additional rule Act⊗ for casts between product types and sum types being active.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

Active c. Variant 1

$$\texttt{ActId} : \frac{}{\mathsf{Active}\,(a \Rightarrow^{\ell} a)} \qquad \texttt{ActInj} : \frac{}{\mathsf{Active}\,(A \Rightarrow^{\ell} ?)}\ A \not\equiv ?,\ \nexists G.\, A \equiv G$$

$$\texttt{ActProj} : \frac{}{\mathsf{Active}\,(? \Rightarrow^{\ell} B)}\ B \not\equiv ?$$

Active c. Variant 2

$$\texttt{ActId} : \frac{}{\mathsf{Active}\,(a \Rightarrow^{\ell} a)} \qquad \texttt{ActInj} : \frac{}{\mathsf{Active}\,(A \Rightarrow^{\ell} ?)}\ A \not\equiv ?,\ \nexists G.\, A \equiv G$$

$$\texttt{ActProj} : \frac{}{\mathsf{Active}\,(? \Rightarrow^{\ell} B)}\ B \not\equiv ?$$

$$\texttt{Act}\otimes : \frac{}{\mathsf{Active}\,(A \otimes B \Rightarrow^{\ell} C \otimes D)} \otimes \in \{\times, +\}$$

Lemma 47. *For any types A and B, $c : A \Rightarrow B$ is either an active or inert cast.*

Lemma 48. *If $c : A \Rightarrow (B \otimes C)$ and* Inert c*, then* Cross c *and $A \equiv D \otimes E$ for some D and E.*

Lemma 49. *A cast $c : A \Rightarrow b$ is not inert.*

Proposition 50. *Both variants of* λB *are instances of the* PreCastStruct *structure.*

We define the following partial function named gnd (short for "ground"). It is defined on all types except for ?.

gnd A

$$\begin{aligned} \mathsf{gnd}\, b &= b \\ \mathsf{gnd}\, A \otimes B &= ? \otimes ? \qquad\qquad \text{for } \otimes \in \{\rightarrow, \times, +\} \end{aligned}$$

Also, we use the following shorthand for a sequence of two casts:

$$M\langle A \Rightarrow^{\ell_1} B \Rightarrow^{\ell_2} C\rangle = M\langle A \Rightarrow^{\ell_1} B\rangle\langle B \Rightarrow^{\ell_2} C\rangle$$

The following is the definition of applyCast for λB Variant 1:

$$\mathsf{applyCast} : \forall \Gamma AB.\, (M : \Gamma \vdash A) \rightarrow \mathsf{Value}\, M \rightarrow (c : A \Rightarrow B) \rightarrow \mathsf{Active}\, c \rightarrow \Gamma \vdash B$$

$$\begin{array}{llllll} \mathsf{applyCast} & M & v & (a \Rightarrow^{\ell} a) & \texttt{ActId} & = M \\ \mathsf{applyCast} & M & v & (A \Rightarrow^{\ell} ?) & \texttt{ActInj} & = M\langle A \Rightarrow^{\ell} \mathsf{gnd}\, A \Rightarrow^{\ell} ?\rangle \\ \mathsf{applyCast} & M\langle G \Rightarrow^{\ell_1} ?\rangle & v & (? \Rightarrow^{\ell_2} B) & \texttt{ActProj} & = M' \end{array}$$

where

$$M' = \begin{cases} M & \text{if } B = \mathsf{gnd}\, B = G \\ \texttt{blame}\, \ell_2 & \text{if } B = \mathsf{gnd}\, B \neq G \\ M\langle G \Rightarrow^{\ell_1} ? \Rightarrow^{\ell_2} \mathsf{gnd}\, B \Rightarrow^{\ell_2} B\rangle & \text{otherwise} \end{cases}$$

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

The definition of applyCast for λB Variant 2 has two additional cases, for $\texttt{Act}\times$ and $\texttt{Act}+$, respectively:

$$\mathsf{applyCast} : \forall \Gamma AB.\, (M : \Gamma \vdash A) \to \mathsf{Value}\, M \to (c : A \Rightarrow B) \to \mathsf{Active}\, c \to \Gamma \vdash B$$

$$\begin{array}{llllll}
\mathsf{applyCast} & M & v & (a \Rightarrow^{\ell} a) & \texttt{ActId} & = M \\
\mathsf{applyCast} & M & v & (A \Rightarrow^{\ell} ?) & \texttt{ActInj} & = M\langle A \Rightarrow^{\ell} \mathsf{gnd}\, A \Rightarrow^{\ell} ?\rangle \\
\mathsf{applyCast} & M & v & c & \texttt{Act}\times & = \mathsf{eta}\times\, M\, c\, \texttt{Cross}\times \\
\mathsf{applyCast} & M & v & c & \texttt{Act}+ & = \mathsf{eta}+\, M\, c\, \texttt{Cross}+ \\
\mathsf{applyCast} & M\langle G \Rightarrow^{\ell_1} ?\rangle & v & (? \Rightarrow^{\ell_2} B) & \texttt{ActProj} & = M'
\end{array}$$

where

$$M' = \begin{cases} M & \text{if } B = \mathsf{gnd}\, B = G \\ \texttt{blame}\, \ell_2 & \text{if } B = \mathsf{gnd}\, B \neq G \\ M\langle G \Rightarrow^{\ell_1} ? \Rightarrow^{\ell_2} \mathsf{gnd}\, B \Rightarrow^{\ell_2} B\rangle & \text{otherwise} \end{cases}$$

Proposition 51. *Both variants of* λB *are instances of the* CastStruct *structure.*

We import and instantiate the reduction semantics and proof of type safety from Sections 3.6 and 3.8 to obtain the following definition and results.

Definition 52 (Reduction for λB). *The reduction relation $M \longrightarrow N$ for* λB *is the reduction relation of CC(⇒) instantiated with* λB*'s instance of* CastStruct.

Corollary 53 (Preservation for λB). *If $M : \Gamma \vdash A$ and $M \longrightarrow M'$, then $M' : \Gamma \vdash A$.*

Corollary 54 (Progress for λB). *If $M : \emptyset \vdash A$, then*

1. *$M \longrightarrow M'$ for some M',*
2. *$\mathsf{Value}\, M$, or*
3. *$M \equiv \texttt{blame}\, \ell$.*

Let λB′ be the variant of λB obtained by instantiating CC' instead of CC.

5.3.2 Blame-subtyping

λB′ uses the "UD" blame-tracking strategy (Wadler & Findler, 2009; Siek *et al.*, 2009), so the subtyping relation that corresponds to safe casts is the one in Figure 17. In this subtyping relation, a type A is a subtype of ? only if it is a subtype of a ground type. For example, $\texttt{Int} \to \texttt{Int} \not<: ?$ because $\texttt{Int} \to \texttt{Int} \not<: ? \to ?$ because $? \not<: \texttt{Int}$.

The CastBlameSafe predicate is defined as it was for EDA in Section 5.1.2, except using the subtyping relation of Figure 17.

The cast operators for λB′ are defined exactly the same as for EDA, so Lemma 30 also applies to λB′, that is, the cast operators such as dom preserve blame safety.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$$\frac{}{? <: ?} \qquad \frac{}{b <: b} \qquad \frac{A <: G}{A <: ?} \qquad \frac{C <: A \quad B <: D}{A \to B <: C \to D}$$

$$\frac{A <: C \quad B <: D}{A \times B <: C \times D} \qquad \frac{A <: C \quad B <: D}{A + B <: C + D}$$

Fig. 17. Subtyping for "UD" blame tracking.

Proposition 55. $\lambda B'$ *is an instance of the* PreBlameSafe *structure.*

Lemma 56 (applyCast preserves blame safety in both variants of $\lambda B'$).
If CastBlameSafe $c\ \ell$ *and* a : Active c *and* V *safe for* ℓ *and* v : Value V, *then* (applyCast $V\ v\ c\ a$) *safe for* ℓ.

Proposition 57 *Both variants of* $\lambda B'$ *are an instance of the* BlameSafe *structure.*

We instantiate Theorem 16 with this BlameSafe instance to obtain the Blame-Subtyping Theorem for both variants of $\lambda B'$.

Corollary 58 (Blame-Subtyping Theorem for both variants of $\lambda B'$).
If $M : \Gamma \vdash A$ *and* M *safe for* ℓ, *then* $\neg(M \longrightarrow^* \texttt{blame}\ \ell)$.

5.3.3 Dynamic gradual guarantee

We now turn to proving the dynamic gradual guarantee for both variants of λB. This involving defining precision on casts and then proving the lemmas required to show that λB is an instance of CastStructWithPrecision. We then instantiate the generic Lemma 19 to obtain the required simulation result and prove the main theorem.

We define the precision relations between two casts and between a cast and a type, corresponding to the first three fields in PreCastStructWithPrecision (Figure 12).

$\boxed{\langle\!\langle i \rangle\!\rangle \sqsubseteq \langle\!\langle i' \rangle\!\rangle}$. Variant 1

$$\frac{i : \mathsf{Inert}\ (G \Rightarrow^{\ell} ?) \qquad i' : \mathsf{Inert}\ (G \Rightarrow^{\ell'} ?)}{\langle\!\langle i \rangle\!\rangle \sqsubseteq \langle\!\langle i' \rangle\!\rangle} \qquad \frac{\begin{array}{c} A \to B \sqsubseteq A' \to B' \qquad C \to D \sqsubseteq C' \to D' \\ i : \mathsf{Inert}\ (A \to B \Rightarrow^{\ell} C \to D) \\ i' : \mathsf{Inert}\ (A' \to B' \Rightarrow^{\ell'} C' \to D') \end{array}}{\langle\!\langle i \rangle\!\rangle \sqsubseteq \langle\!\langle i' \rangle\!\rangle}$$

$$\frac{\begin{array}{c} A \times B \sqsubseteq A' \times B' \qquad C \times D \sqsubseteq C' \times D' \\ i : \mathsf{Inert}\ (A \times B \Rightarrow^{\ell} C \times D) \\ i' : \mathsf{Inert}\ (A' \times B' \Rightarrow^{\ell'} C' \times D') \end{array}}{\langle\!\langle i \rangle\!\rangle \sqsubseteq \langle\!\langle i' \rangle\!\rangle} \qquad \frac{\begin{array}{c} A + B \sqsubseteq A' + B' \qquad C + D \sqsubseteq C' + D' \\ i : \mathsf{Inert}\ (A + B \Rightarrow^{\ell} C + D) \\ i' : \mathsf{Inert}\ (A' + B' \Rightarrow^{\ell'} C' + D') \end{array}}{\langle\!\langle i \rangle\!\rangle \sqsubseteq \langle\!\langle i' \rangle\!\rangle}$$

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$\langle\!\langle i \rangle\!\rangle \sqsubseteq A'$. Variant 1

$$\frac{i : \mathsf{Inert}\,(G \Rightarrow^{\ell} ?) \qquad G \sqsubseteq A'}{\langle\!\langle i \rangle\!\rangle \sqsubseteq A'} \qquad \frac{A \to B \sqsubseteq A' \to B' \qquad C \to D \sqsubseteq A' \to B' \qquad i : \mathsf{Inert}\,(A \to B \Rightarrow^{\ell} C \to D)}{\langle\!\langle i \rangle\!\rangle \sqsubseteq A' \to B'}$$

$$\frac{A \times B \sqsubseteq A' \times B' \qquad C \times D \sqsubseteq A' \times B' \qquad i : \mathsf{Inert}\,(A \times B \Rightarrow^{\ell} C \times D)}{\langle\!\langle i \rangle\!\rangle \sqsubseteq A' \times B'} \qquad \frac{A + B \sqsubseteq A' + B' \qquad C + D \sqsubseteq A' + B' \qquad i : \mathsf{Inert}\,(A + B \Rightarrow^{\ell} C + D)}{\langle\!\langle i \rangle\!\rangle \sqsubseteq A' + B'}$$

$A \sqsubseteq \langle\!\langle i' \rangle\!\rangle$. Variant 1

$$\frac{A \to B \sqsubseteq A' \to B' \qquad A \to B \sqsubseteq C' \to D' \qquad i' : \mathsf{Inert}\,(A' \to B' \Rightarrow^{\ell} C' \to D')}{A \to B \sqsubseteq \langle\!\langle i' \rangle\!\rangle} \qquad \frac{A \times B \sqsubseteq A' \times B' \qquad A \times B \sqsubseteq C' \times D' \qquad i' : \mathsf{Inert}\,(A' \times B' \Rightarrow^{\ell} C' \times D')}{\langle\!\langle A \times B \rangle\!\rangle \sqsubseteq i'}$$

$$\frac{A + B \sqsubseteq A' + B' \qquad A + B \sqsubseteq C' + D' \qquad i' : \mathsf{Inert}\,(A' + B' \Rightarrow^{\ell} C' + D')}{\langle\!\langle A + B \rangle\!\rangle \sqsubseteq i'}$$

The precision relations for Variant 2 have fewer cases compared to Variant 1, since the casts between product types and sum types are active:

$\langle\!\langle i \rangle\!\rangle \sqsubseteq \langle\!\langle i' \rangle\!\rangle$. Variant 2

$$\frac{i : \mathsf{Inert}\,(G \Rightarrow^{\ell} ?) \qquad i' : \mathsf{Inert}\,(G \Rightarrow^{\ell'} ?)}{\langle\!\langle i \rangle\!\rangle \sqsubseteq \langle\!\langle i' \rangle\!\rangle} \qquad \frac{A \to B \sqsubseteq A' \to B' \qquad C \to D \sqsubseteq C' \to D' \qquad i : \mathsf{Inert}\,(A \to B \Rightarrow^{\ell} C \to D) \qquad i' : \mathsf{Inert}\,(A' \to B' \Rightarrow^{\ell'} C' \to D')}{\langle\!\langle i \rangle\!\rangle \sqsubseteq \langle\!\langle i' \rangle\!\rangle}$$

$\langle\!\langle i \rangle\!\rangle \sqsubseteq A'$. Variant 2

$$\frac{i : \mathsf{Inert}\,(G \Rightarrow^{\ell} ?) \qquad G \sqsubseteq A'}{\langle\!\langle i \rangle\!\rangle \sqsubseteq A'} \qquad \frac{A \to B \sqsubseteq A' \to B' \qquad C \to D \sqsubseteq A' \to B' \qquad i : \mathsf{Inert}\,(A \to B \Rightarrow^{\ell} C \to D)}{\langle\!\langle i \rangle\!\rangle \sqsubseteq A' \to B'}$$

$A \sqsubseteq \langle\!\langle i' \rangle\!\rangle$. Variant 2

$$\frac{A \to B \sqsubseteq A' \to B' \qquad A \to B \sqsubseteq C' \to D' \qquad i' : \mathsf{Inert}\,(A' \to B' \Rightarrow^{\ell} C' \to D')}{A \to B \sqsubseteq \langle\!\langle i' \rangle\!\rangle}$$

Then, we instantiate and prove the four lemmas that correspond to the last four fields in PreCastStructWithPrecision (Figure 12). They are forward direction and inversion lemmas about the precision relations defined above between casts and between a cast and a type.

Lemma 59 (Type Precision Implies Cast-Type Precision). *Suppose* $c : A \Rightarrow B$ *and* $i : \mathsf{Inert}\,c$. *If* $A \sqsubseteq A'$, $B \sqsubseteq A'$, *then* $\langle\!\langle i \rangle\!\rangle \sqsubseteq A'$.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

Lemma 60 (Cast-Cast Precision Implies Type Precision). *Suppose* $c : A \Rightarrow B$, $c' : A' \Rightarrow B'$, $i : \mathsf{Inert}\, c$, $i' : \mathsf{Inert}\, c'$. *If* $\langle\!\langle i \rangle\!\rangle \sqsubseteq \langle\!\langle i' \rangle\!\rangle$, *then* $A \sqsubseteq A'$ *and* $B \sqsubseteq B'$.

Lemma 61 (Cast-Type Precision Implies Type Precision). *Suppose* $c : A \Rightarrow B$ *and* $i : \mathsf{Inert}\, c$. *If* $\langle\!\langle i \rangle\!\rangle \sqsubseteq A'$, *then* $A \sqsubseteq A'$ *and* $B \sqsubseteq A'$.

Lemma 62 (Type-Cast Precision Implies Type Precision). *Suppose* $c' : A' \Rightarrow B'$ *and* $i' : \mathsf{Inert}\, c'$. *If* $A \sqsubseteq \langle\!\langle i' \rangle\!\rangle$, *then* $A \sqsubseteq A'$ *and* $A \sqsubseteq B'$.

We instantiate PreCastStructWithPrecision using the definitions of precision and their related lemmas (Lemmas 59, 60, 61, and 62).

Proposition 63 $\lambda B'$ *is an instance of the* PreCastStructWithPrecision *structure.*

We instantiate Lemma 21 to prove that compilation of the GTLC into $\lambda B'$ preserves the precision relation.

Corollary 64 (Compilation into $\lambda B'$ Preserves Precision). *Suppose* $\Gamma \vdash_G M : A$ *and* $\Gamma' \vdash_G M' : A'$. *If* $\Gamma \sqsubseteq \Gamma'$ *and* $M \sqsubseteq^G M'$, *then* $\mathcal{C}'[\![M]\!] \sqsubseteq^C \mathcal{C}'[\![M']\!]$ *and* $A \sqsubseteq A'$.

Lemma 65 (applyCast Catches Up to the Right). *Suppose* $V' : \Gamma' \vdash A'$, $c : A \Rightarrow B$, *and* $a : \mathsf{Active}\, c$. *If* $A \sqsubseteq A'$, $B \sqsubseteq A'$, *and* $V \sqsubseteq^C V'$, *then* $\mathsf{applyCast}\, V\, c\, a \longrightarrow^* W$ *and* $W \sqsubseteq^C V'$ *for some value* W.

Proof sketch. We briefly describe the proof for Variant 1, since the proof methodology is the same for both variants.

By induction on the premise $\mathsf{Active}\, c$, it generates three cases:

`ActId`. Since c is identity cast, $W \equiv V$ satisfies the reduction and $W \sqsubseteq^C V'$.

`ActInj`. We follow the branch structure of applyCast and proceed.

`ActProj`. We follow the branch structure of applyCast and case on whether the target type B of the projection is ground. If B is ground, then the proof is straightforward by inversion on the canonical form of the projected value. Otherwise, if the B is not ground, applyCast expands the cast by routing through a ground type, from where we proceed using the induction hypothesis. □

Lemma 66 (Simulation Between Cast and applyCast). *Suppose* $c : A \Rightarrow B$, $c' : A' \Rightarrow B'$, *and* $a' : \mathsf{Active}\, c'$. *If* $A \sqsubseteq A'$, $B \sqsubseteq B'$, *and* $V \sqsubseteq^C V'$, *then* $V\langle c \rangle \longrightarrow^* N$ *and* $N \sqsubseteq^C \mathsf{applyCast}\, V'\, c'\, a'$ *for some* N.

Proof sketch. We case on $\mathsf{Active}\, c'$ and generate three cases that are all straightforward. □

Lemma 67 (Simulation Between Cast and Wrap). *Suppose* $c : A \Rightarrow B$, $c' : A' \Rightarrow B'$, *and* $i' : \mathsf{Inert}\, c'$. *If* $A \sqsubseteq A'$, $B \sqsubseteq B'$, *and* $V \sqsubseteq^C V'$, *then* $V\langle c \rangle \longrightarrow^* N$ *and* $N \sqsubseteq^C V'\langle\!\langle i' \rangle\!\rangle$ *for some* N.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

Proof sketch. We case on the premise Inert c':

`InInj`. In this case, by inversion on $A \sqsubseteq A'$ and $B \sqsubseteq B'$, A can be either ? or ground. If A is ?, an identity cast $? \Rightarrow ?$ is active so we use rule `cast` and proceed; otherwise, the cast is inert so we use rule `wrap` and proceed.

`InCross`. Similar to the `InInj` case, by inversion on the two type precision relations. □

Lemma 68 (Simulation Between Value and applyCast). *Suppose* $V : \emptyset \vdash A$, $c' : A' \Rightarrow B'$, *and* $a' : \mathsf{Active}\, c'$. *If* $A \sqsubseteq A'$, $A \sqsubseteq B'$, *and* $V \sqsubseteq^C V'$, *then* $V \sqsubseteq^C \mathsf{applyCast}\, V'\, c'\, a'$.

Proof sketch. Straightforward. By case analysis on $a' : \mathsf{Active}\, c'$. □

Lemma 69 (Simulation Between Value and Wrap). *Suppose* $V : \emptyset \vdash A$, $c' : A' \Rightarrow B'$, *and* $i' : \mathsf{Inert}\, c'$. *If* $A \sqsubseteq A'$, $A \sqsubseteq B'$, *and* $V \sqsubseteq^C V'$, *then* $V \sqsubseteq^C V' \langle\!\langle i' \rangle\!\rangle$.

Proof sketch. Straightforward. By case analysis on $i' : \mathsf{Inert}\, c'$ and by inversion on the type precision relations. □

We prove that both variants of $\lambda B'$ are instances of CastStructWithPrecision using Lemmas 65, 66, 67, 68, and 69.

Proposition 70 *Both variants of* $\lambda B'$ *are an instance of the* CastStructWithPrecision *structure.*

We instantiate Lemma 19 (Simulation of More Precise Programs) with the $\lambda B'$ instances of CastStructWithPrecision (Proposition 70) to obtain the main lemma of the dynamic gradual guarantee for both variants of $\lambda B'$.

Corollary 71 (Simulation of More Precise Programs for $\lambda B'$). *Suppose* $M_1 : \emptyset \vdash A$ *and* $M_1' : \emptyset \vdash A'$ *and* $M_2' : \emptyset \vdash A'$. *If* $M_1 \sqsubseteq^C M_1'$ *and* $M_1' \longrightarrow M_2'$, *then* $M_1 \longrightarrow^* M_2$ *and* $M_2 \sqsubseteq^C M_2'$ *for some* M_2.

We prove the dynamic gradual guarantee for $\lambda B'$ following the reasoning of Siek *et al.* (2015*b*). We give the full proof here.

Theorem 72 (Dynamic Gradual Guarantee for both variants of $\lambda B'$).
Suppose $M \sqsubseteq^G N$ *and* $\Gamma \vdash_G M : A$ *and* $\Gamma \vdash_G N : B$.

1. *If* $\mathcal{C}[\![N]\!] \longrightarrow^* W$, *then* $\mathcal{C}[\![M]\!] \longrightarrow^* V$ *and* $V \sqsubseteq^C W$.
2. *If* $\mathcal{C}[\![N]\!]$ *diverges, so does* $\mathcal{C}[\![M]\!]$.
3. *If* $\mathcal{C}[\![M]\!] \longrightarrow^* V$, *then either* $\mathcal{C}[\![N]\!] \longrightarrow^* W$ *and* $V \sqsubseteq^C W$, *or* $\mathcal{C}[\![N]\!] \longrightarrow^* \mathtt{blame}\, \ell$ *for some* ℓ.
4. *If* $\mathcal{C}[\![M]\!]$ *diverges then either* $\mathcal{C}[\![N]\!]$ *diverges or* $\mathcal{C}[\![N]\!] \longrightarrow^* \mathtt{blame}\, \ell$ *for some* ℓ.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

Proof.

1. By Corollary 64, we have $\mathcal{C}[\![M]\!] \sqsubseteq^C \mathcal{C}[\![N]\!]$. Then by induction on $\mathcal{C}[\![N]\!] \longrightarrow^* W$ and Corollary 71, we have $\mathcal{C}[\![M]\!] \longrightarrow^* M'$ and $M' \sqsubseteq^C W$ for some M'. Finally, by Lemma 17 (instantiated for $\lambda\mathsf{B}'$), we have $M' \longrightarrow^* V$ and $V \sqsubseteq^C W$.
2. By Corollary 64, we have $\mathcal{C}[\![M]\!] \sqsubseteq^C \mathcal{C}[\![N]\!]$. Then by Corollary 71, $\mathcal{C}[\![M]\!]$ also diverges.
3. Because $\lambda\mathsf{B}'$ is type safe (Corollaries 53 and 54), we have the following cases.
   - Case $\mathcal{C}[\![N]\!] \longrightarrow^* W$ for some W. Then by Part 1 of this theorem and because reduction is deterministic, $V \sqsubseteq^C W$.
   - Case $\mathcal{C}[\![N]\!] \longrightarrow^* \mathtt{blame}\ \ell$ for some ℓ. We immediately conclude.
   - Case $\mathcal{C}[\![N]\!]$ diverges. Then by Part 1, $\mathcal{C}[\![M]\!]$ also diverges, but that contradicts the assumption that $\mathcal{C}[\![M]\!] \longrightarrow^* V$.
4. Again because $\lambda\mathsf{B}'$ is type safe (Corollaries 53 and 54), we have the following cases.
   - Case $\mathcal{C}[\![N]\!] \longrightarrow^* W$ for some W. Then by Part 1, $\mathcal{C}[\![M]\!] \longrightarrow^* V$ for some V but that contradicts the assumption that $\mathcal{C}[\![M]\!]$ diverges.
   - Case $\mathcal{C}[\![N]\!] \longrightarrow^* \mathtt{blame}\ \ell$ for some ℓ. We immediately conclude.
   - Case $\mathcal{C}[\![N]\!]$ diverges. We immediately conclude. □

5.4 Partially Eager "D" Coercions (EDC)

The next three cast calculi use cast representations based on the Coercion Calculus of Henglein (1994). We start with one that provides the same behavior as the cast calculus of Siek & Taha (2006*a*), that is, partially eager casts with active cross casts (Section 5.1). We use the abbreviation EDC for this cast calculus.

We define coercions as follows, omitting sequence coercions because they are not necessary in this calculus.

$\boxed{c, d : A \Rightarrow B}$

$$\mathsf{id} : \frac{}{a \Rightarrow a} \qquad A! : \frac{A \neq ?}{A \Rightarrow ?} \qquad B?^{\ell} : \frac{B \neq ?}{? \Rightarrow B}$$

$$- \to - : \frac{C \Rightarrow A \qquad B \Rightarrow D}{(A \to B) \Rightarrow (C \to D)} \qquad - \otimes - : \frac{A \Rightarrow C \qquad B \Rightarrow D}{(A \otimes B) \Rightarrow (C \otimes D)} \otimes \in \{\times, +\}$$

The cast constructor is defined by applying the coerce function (defined later in this section) to the implicit proof of consistency between A and B and the blame label ℓ:

$$(\!|A \Rightarrow B|\!)^{\ell}\ \{p : A \sim B\} = \mathsf{coerce}\, p\, \ell$$

5.4.1 Reduction semantics and type safety

Injections are categorized as inert casts.

$\boxed{\mathsf{Inert}\ c}$

$$\frac{}{\mathsf{Inert}\ A!}$$

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

The coercions between function, pair, and sum types are categorized as cross casts.

Cross c

$$\mathsf{Cross}\otimes : \frac{}{\mathsf{Cross}\,(c \otimes d)} \otimes \in \{\rightarrow, \times, +\}$$

We categorize the identity, projection, and cross casts as active.

Active c

$$\texttt{ActId} : \frac{}{\mathsf{Active}\,\mathsf{id}} \qquad \texttt{ActProj} : \frac{}{\mathsf{Active}\,A?^{\ell}} \qquad \texttt{ActCross} : \frac{\mathsf{Cross}\,c}{\mathsf{Active}\,c}$$

Lemma 73. *For any types A and B, $c : A \Rightarrow B$ is either an active or inert cast.*

Lemma 74. *If $c : A \Rightarrow (B \otimes C)$ and* Inert *c, then* Cross *c and $A \equiv D \otimes E$ for some D and E.*

The cast operators dom, cod, etc. are defined as follows:

$$\begin{aligned}
\mathsf{dom}\,(c \rightarrow d)\,x &= c \\
\mathsf{cod}\,(c \rightarrow d)\,x &= d \\
\mathsf{fst}\,(c \times d)\,x &= c \\
\mathsf{snd}\,(c \times d)\,x &= d \\
\mathsf{inl}\,(c + d)\,x &= c \\
\mathsf{inr}\,(c + d)\,x &= d
\end{aligned}$$

Lemma 75. *A cast $c : A \Rightarrow b$ is not inert.*

Proposition 76 *The EDC calculus is instance of the* PreCastStruct *structure.*

To help define the applyCast function, we define an auxiliary function named coerce for converting two consistent types and a blame label into a coercion. (The coerce function is also necessary for compiling from the GTLC to this calculus.)

$$\mathsf{coerce} : \forall A\,B.\, A \sim B \rightarrow \mathsf{Label} \rightarrow A \Rightarrow B$$

$$\mathsf{coerce}\ \mathsf{UnkL}{\sim}[B]\ \ell = \begin{cases} \mathsf{id} & \text{if } B \equiv ? \\ B?^{\ell} & B \not\equiv ? \end{cases}$$

$$\mathsf{coerce}\ \mathsf{UnkR}{\sim}[A]\ \ell = \begin{cases} \mathsf{id} & \text{if } A \equiv ? \\ A! & A \not\equiv ? \end{cases}$$

$$\begin{aligned}
\mathsf{coerce}\ \texttt{Base}{\sim}[b]\ \ell &= \mathsf{id} \\
\mathsf{coerce}\ (\texttt{Fun}{\sim}\ d_1\ d_2)\ \ell &= (\mathsf{coerce}\ d_1\ \overline{\ell}) \rightarrow (\mathsf{coerce}\ d_2\ \ell) \\
\mathsf{coerce}\ (\texttt{Pair}{\sim}\ d_1\ d_2)\ \ell &= (\mathsf{coerce}\ d_1\ \ell) \times (\mathsf{coerce}\ d_2\ \ell) \\
\mathsf{coerce}\ (\texttt{Sum}{\sim}\ d_1\ d_2)\ \ell &= (\mathsf{coerce}\ d_1\ \ell) + (\mathsf{coerce}\ d_2\ \ell)
\end{aligned}$$

The structure of coercions is quite similar to that of the active casts but a bit more convenient to work with, so we define the applyCast function by cases on the coercion.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$$\frac{}{\mathsf{CastBlameSafe}\ \mathsf{id}\ \ell} \qquad \frac{}{\mathsf{CastBlameSafe}\ A!\ \ell} \qquad \frac{\ell \neq \ell'}{\mathsf{CastBlameSafe}\ B?^{\ell'}\ \ell}$$

$$\frac{\mathsf{CastBlameSafe}\ c\ \ell \qquad \mathsf{CastBlameSafe}\ d\ \ell}{\mathsf{CastBlameSafe}\ (c \otimes d)\ \ell} \otimes \in \{\rightarrow, \times, +\}$$

Fig. 18. Definition of the CastBlameSafe predicate for EDC′.

We omit the case for injection because that coercion is not active.

$$\mathsf{applyCast} : \forall \Gamma AB.\ (M : \Gamma \vdash A) \rightarrow \mathsf{Value}\ M \rightarrow (c : A \Rightarrow B) \rightarrow \mathsf{Active}\ c \rightarrow \Gamma \vdash B$$

$$\begin{array}{llllll} \mathsf{applyCast} & M & v & \mathsf{id} & a & = & M \\ \mathsf{applyCast} & M\langle A! \rangle & v & B?^{\ell} & a & = & \begin{cases} M\langle \mathsf{coerce}\ ab\ \ell \rangle & \text{if } ab : A \sim B \\ \mathtt{blame}\ \ell & \text{otherwise} \end{cases} \\ \mathsf{applyCast} & M & v & c \otimes d & a & = & \mathsf{eta}\otimes\ M\ c\ \mathtt{Cross}\otimes \end{array}$$

Proposition 77. *The EDC calculus is an instance of the* CastStruct *structure.*

We import and instantiate the reduction semantics and proof of type safety from Sections 3.6 and 3.8 to obtain the following definition and results.

Definition 78 (Reduction for EDC). *The reduction relation $M \longrightarrow N$ for the EDC calculus is the reduction relation of CC($\Rightarrow$) instantiated with EDC's instance of* CastStruct.

Corollary 79 (Preservation for EDC). *If $M : \Gamma \vdash A$ and $M \longrightarrow M'$, then $M' : \Gamma \vdash A$.*

Corollary 80 (Progress for EDC). *If $M : \emptyset \vdash A$, then*

1. *$M \longrightarrow M'$ for some M',*
2. *Value M, or*
3. *$M \equiv \mathtt{blame}\ \ell$.*

Let EDC′ be the variant of EDC obtained by instantiating *CC′* instead of *CC*.

5.4.2 Blame-subtyping

The CastBlameSafe predicate for EDC′ is defined in Figure 18.

Lemma 81 (Blame Safety of Cast Operators). *Suppose* CastBlameSafe $c\ \ell$ *and c is a cross cast, that is,* Cross c.

- *If $x = \mathtt{Cross}\rightarrow$, then* CastBlameSafe (dom $c\ x$) ℓ *and* CastBlameSafe (cod $c\ x$) ℓ.
- *If $x = \mathtt{Cross}\times$, then* CastBlameSafe (fst $c\ x$) ℓ *and* CastBlameSafe (snd $c\ x$) ℓ.
- *If $x = \mathtt{Cross}+$, then* CastBlameSafe (inl $c\ x$) ℓ *and* CastBlameSafe (inr $c\ x$) ℓ.

Proof. By inversion on Cross c and the CastBlameSafe predicate. □

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

Proposition 82. *EDC′ is an instance of the* PreBlameSafe *structure.*

Lemma 83 (coerce is blame safe). *Suppose* $\ell \neq \ell'$. *If* $ab : A \sim B$, *then* CastBlameSafe (coerce $ab\ \ell'$) ℓ.

Lemma 84 (applyCast preserves blame safety in EDC′). *If* CastBlameSafe $c\ \ell$ *and* a : Active c *and* V *safe for* ℓ *and* v : Value V, *then* (applyCast $V\ v\ c\ a$) *safe for* ℓ.

Proof. The proof is by cases on Active c and inversion on CastBlameSafe $c\ \ell$, using Lemma 83 in the case for projection. □

Proposition 85. *EDC′ is an instance of the* BlameSafe *structure.*

We instantiate Theorem 16 with this BlameSafe instance to obtain the Blame-Subtyping Theorem for EDC′.

Corollary 86 (Blame-Subtyping Theorem for EDC′). *If* $M : \Gamma \vdash A$ *and* M *safe for* ℓ, *then* $\neg(M \longrightarrow^* \texttt{blame}\ \ell)$.

5.5 Lazy "D" Coercions (LDC)

The Lazy "D" Coercions sometimes catch inconsistencies later in the execution of a program compared to the partially eager and eager calculi. For example, the following term does not immediately trigger a cast failure:

$$(\lambda\text{`Z})\langle \texttt{Nat} \to \texttt{Nat} \Rightarrow \texttt{Bool} \to \texttt{Bool} \rangle$$

It would instead trigger an error if/when it is applied to an argument. In this respect, Lazy "D" Coercions are similar to λB (Section 5.3) and λC (Section 5.6). The lazy behavior of the Lazy "D" Coercions is acheived by changing the definition of applyCast to use shallow consistency instead of consistency (via the definition of the cast constructor $(\!|A \Rightarrow B|\!)^\ell$), which only inspects the head constructors of the source and target type. However, Lazy "D" Coercions differs from λB and λC with respect to how it performs blame tracking. In particular, a cast from any type to the unknown type ? is a safe cast with the "D" blame-tracking strategy.

The Lazy "D" Coercions (Siek *et al.*, 2009) are syntactically similar to the coercions of Section 5.4 except that include a failure coercion, written $\bot^\ell$.

$\boxed{c, d : A \Rightarrow B}$

$$\bot^\ell : \frac{}{A \Rightarrow B} \qquad \mathsf{id} : \frac{}{a \Rightarrow a} \qquad A! : \frac{A \neq ?}{A \Rightarrow ?} \qquad B?^\ell : \frac{B \neq ?}{? \Rightarrow B}$$

$$- \to - : \frac{C \Rightarrow A \quad B \Rightarrow D}{(A \to B) \Rightarrow (C \to D)} \qquad - \otimes - : \frac{A \Rightarrow C \quad B \Rightarrow D}{(A \otimes B) \Rightarrow (C \otimes D)} \otimes \in \{\times, +\}$$

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

The cast constructor is defined as follows, using the coerce function defined later in this section:

$$(\!|A \Rightarrow B|\!)^{\ell} = \begin{cases} \mathsf{coerce}\, d\, \ell & \text{if } d : A \smile B \\ \bot^{\ell} & \text{otherwise} \end{cases}$$

5.5.1 Reduction semantics and type safety

Injections are categorized as inert casts.

$\boxed{\mathsf{Inert}\ c}$

$$\frac{}{\mathsf{Inert}\ A!}$$

The coercions between function, pair, and sum types are categorized as cross casts.

$\boxed{\mathsf{Cross}\ c}$

$$\mathsf{Cross}\otimes : \frac{}{\mathsf{Cross}\ c \otimes d} \otimes \in \{\rightarrow, \times, +\}$$

In addition to the identity and projection coercions and the cross casts, the failure coercions are also active.

$\boxed{\mathsf{Active}\ c}$

$$\texttt{ActId} : \frac{}{\mathsf{Active\ id}} \qquad \texttt{ActProj} : \frac{}{\mathsf{Active}\ A?^{\ell}} \qquad \texttt{ActCross} : \frac{\mathsf{Cross}\ c}{\mathsf{Active}\ c}$$

$$\texttt{ActFail} : \frac{}{\mathsf{Active}\ \bot^{\ell}}$$

Lemma 87. *For any types A and B, $c : A \Rightarrow B$ is either an active or inert cast.*

Lemma 88. *If $c : A \Rightarrow (B \otimes C)$ and* Inert *c, then* Cross *c and $A \equiv D \otimes E$ for some D and E.*

The definition of cast operators such as dom are the same as in Section 5.4.

Lemma 89. *A cast $c : A \Rightarrow b$ is not inert.*

Proposition 90. *The LDC calculus is an instance of the* PreCastStruct *structure.*

We define shallow consistency, written $A \smile B$, as follows:

$$\texttt{UnkL}\smile[B] : ? \smile B \qquad \texttt{UnkR}\smile[B] : A \smile ? \qquad \texttt{Base}\smile[b] : b \smile b$$

$$\otimes\smile[A, B, C, D] : (A \otimes B) \smile (C \otimes D)$$

The coerce function differs from that of the EDC calculus (Section 5.4) in that we only require the source and target types to be shallowly consistent. The coerce function is mutually defined with the function $(\!|A \Rightarrow B|\!)^{\ell}$ that was presented earlier in this section:

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$$\mathsf{coerce} : \forall A\ B.\ A \smile B \to \mathsf{Label} \to A \Rightarrow B$$

$$\mathsf{coerce}\ \mathsf{UnkL}\smile[B]\ \ell = \begin{cases} \mathsf{id} & \text{if } B \equiv ? \\ B?^{\ell} & B \not\equiv ? \end{cases}$$

$$\mathsf{coerce}\ \mathsf{UnkR}\smile[A]\ \ell = \begin{cases} \mathsf{id} & \text{if } A \equiv ? \\ A! & A \not\equiv ? \end{cases}$$

$$\mathsf{coerce}\ \mathsf{Base}\smile[b]\ \ell = \mathsf{id}$$

$$\mathsf{coerce}\ (\mathtt{Fun}\smile[A,B,C,D])\ \ell = (\!|C \Rightarrow A|\!)^{\overline{\ell}} \to (\!|B \Rightarrow D|\!)^{\ell}$$

$$\mathsf{coerce}\ (\mathtt{Pair}\smile[A,B,C,D])\ \ell = (\!|A \Rightarrow C|\!)^{\ell} \times (\!|B \Rightarrow D|\!)^{\ell}$$

$$\mathsf{coerce}\ (\mathtt{Sum}\smile[A,B,C,D])\ \ell = (\!|A \Rightarrow C|\!)^{\ell} + (\!|B \Rightarrow D|\!)^{\ell}$$

The definition of applyCast is similar to the one for EDC (Section 5.4) except that there is an additional case for $\bot^{\ell}$ and the projection case checks for shallow consistency instead of consistency, using $(\!|A \Rightarrow B|\!)^{\ell}$:

$$\mathsf{applyCast} : \forall \Gamma A B.\ (M : \Gamma \vdash A) \to \mathsf{Value}\ M \to (c : A \Rightarrow B) \to \mathsf{Active}\ c \to \Gamma \vdash B$$

$$\begin{array}{llllll} \mathsf{applyCast} & M & v & \mathsf{id} & a & = & M \\ \mathsf{applyCast} & M\langle A! \rangle & v & B?^{\ell} & a & = & M\langle (\!|A \Rightarrow B|\!)^{\ell} \rangle \\ \mathsf{applyCast} & M & v & c \otimes d & a & = & \mathtt{eta}\otimes\ M\ c\ \mathtt{Cross}\otimes \\ \mathsf{applyCast} & M & v & \bot^{\ell} & a & = & \mathtt{blame}\ \ell \end{array}$$

Proposition 91. *The LDC calculus is an instance of the* CastStruct *structure.*

We import and instantiate the reduction semantics and proof of type safety from Sections 3.6 and 3.8 to obtain the following definition and results.

Definition 92 (Reduction for LDC). *The reduction relation $M \longrightarrow N$ for the LDC calculus is the reduction relation of CC($\Rightarrow$) instantiated with LDC's instance of* CastStruct.

Corollary 93 (Preservation for LDC). *If $M : \Gamma \vdash A$ and $M \longrightarrow M'$, then $M' : \Gamma \vdash A$.*

Corollary 94 (Progress for LDC). *If $M : \emptyset \vdash A$, then*

1. *$M \longrightarrow M'$ for some M',*
2. *Value M, or*
3. *$M \equiv \mathtt{blame}\ \ell$.*

Let LDC′ be the variant of LDC obtained by instantiating CC' instead of CC.

5.5.2 Blame-subtyping

The CastBlameSafe predicate for LDC′ is the same as for EDC′ (Section 5.4) except that there is an additional rule for failure coercions:

$$\frac{\ell \neq \ell'}{\mathsf{CastBlameSafe}\ \bot^{\ell'}\ \ell}$$

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

Lemma 95 (Blame Safety of Cast Operators). *Suppose* CastBlameSafe $c\ \ell$ *and* c *is a cross cast, that is,* x : Cross c.

- *If* $x = \texttt{Cross}\to$, *then* CastBlameSafe (dom $c\ x$) ℓ *and* CastBlameSafe (cod $c\ x$) ℓ.
- *If* $x = \texttt{Cross}\times$, *then* CastBlameSafe (fst $c\ x$) ℓ *and* CastBlameSafe (snd $c\ x$) ℓ.
- *If* $x = \texttt{Cross}+$, *then* CastBlameSafe (inl $c\ x$) ℓ *and* CastBlameSafe (inr $c\ x$) ℓ.

Proposition 96. *LDC′ is an instance of the* PreBlameSafe *structure.*

Lemma 97. (coerce is blame safe). *Suppose* $\ell \neq \ell'$.

1. CastBlameSafe $(\!|A \Rightarrow B|\!)^{\ell'}\ \ell$.
2. *If* $ab : A \smile B$, *then* CastBlameSafe (coerce $ab\ \ell'$) ℓ.

Lemma 98. (applyCast preserves blame safety in LDC′). *If* CastBlameSafe $c\ \ell$ *and* a : Active c *and* V *safe for* ℓ *and* v : Value V, *then* (applyCast $V\ v\ c\ a$) *safe for* ℓ.

Proof. The proof is by cases on Active c and inversion on CastBlameSafe $c\ \ell$, using Lemma 97 in the case for projection. □

Proposition 99. *LDC′ is an instance of the* BlameSafe *structure.*

We instantiate Theorem 16 with this BlameSafe instance to obtain the Blame-Subtyping Theorem for LDC′.

Corollary 100 (Blame-Subtyping Theorem for LDC′). *If* $M : \Gamma \vdash A$ *and* M *safe for* ℓ, *then* $\neg(M \longrightarrow^* \texttt{blame}\ \ell)$.

5.6 The λC *Coercion Calculus*

This section instantiates the Parametric Cast Calculus to obtain the λC calculus of Siek *et al.* (2015*a*). Again we represent casts as coercions, but this time we must include the notion of sequencing of two coercions, written $c\,;d$, to enable the factoring of casts through the ground type. As part of this factoring, injections and projections are restricted to ground types. We omit the failure coercion because it is not necessary for λC.

$$\boxed{c, d : A \Rightarrow B}$$

$$\mathsf{id} : \frac{}{a \Rightarrow a} \qquad G! : \frac{}{G \Rightarrow ?} \qquad H?^{\ell} : \frac{}{? \Rightarrow H} \qquad - \,;- : \frac{A \Rightarrow B \qquad B \Rightarrow C}{A \Rightarrow C}$$

$$- \to - : \frac{C \Rightarrow A \qquad B \Rightarrow D}{(A \to B) \Rightarrow (C \to D)} \qquad - \otimes - : \frac{A \Rightarrow C \qquad B \Rightarrow D}{(A \otimes B) \Rightarrow (C \otimes D)} \otimes \in \{\times, +\}$$

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

The cast constructor is defined as follows:

$$
\begin{aligned}
(\!|A \Rightarrow ?|\!)^{\ell} &= (\!|A \Rightarrow G|\!)^{\ell} \,;(\!|G \Rightarrow ?|\!)^{\ell} \\
(\!|? \Rightarrow A|\!)^{\ell} &= (\!|? \Rightarrow G|\!)^{\ell} \,;(\!|G \Rightarrow A|\!)^{\ell} \\
(\!|G \Rightarrow ?|\!)^{\ell} &= G! \\
(\!|? \Rightarrow H|\!)^{\ell} &= H?^{\ell} \\
(\!|b \Rightarrow b|\!)^{\ell} &= \mathsf{id} \\
(\!|A \to B \Rightarrow A' \to B'|\!)^{\ell} &= (\!|A' \Rightarrow A|\!)^{\ell} \to (\!|B \Rightarrow B'|\!)^{\ell} \\
(\!|A \times B \Rightarrow A' \times B'|\!)^{\ell} &= (\!|A' \Rightarrow A|\!)^{\ell} \times (\!|B \Rightarrow B'|\!)^{\ell} \\
(\!|A + B \Rightarrow A' + B'|\!)^{\ell} &= (\!|A' \Rightarrow A|\!)^{\ell} + (\!|B \Rightarrow B'|\!)^{\ell}
\end{aligned}
$$

5.6.1 Reduction semantics and type safety

The coercions between function, pair, and sum types are categorized as cross casts. We do not categorize sequence coercions as cross casts, which, for example, simplifies the definition of the dom and cod functions.

Cross c

$$\texttt{Cross}\otimes : \frac{}{\mathsf{Cross}\; c \otimes d} \otimes \in \{\to, \times, +\}$$

Injections and function coercions are categorized as inert casts.

Inert c

$$\texttt{InInj} : \frac{}{\mathsf{Inert}\; G!} \qquad \texttt{InFun} : \frac{}{\mathsf{Inert}\; c \to d}$$

The active casts in λC include identity casts, projections, and sequences. The λC calculus did not include pairs and sums (Siek *et al.*, 2015a), but here we choose to categorize casts between pairs and sums as active casts, as we did for Variant 2 of λB in Section 5.3.

Active c

$$\texttt{ActId} : \frac{}{\mathsf{Active}\; \mathsf{id}} \qquad \texttt{ActProj} : \frac{}{\mathsf{Active}\; H?^{\ell}} \qquad \texttt{ActSeq} : \frac{}{\mathsf{Active}\; (c\,;d)}$$

$$\texttt{Act}\otimes : \frac{}{\mathsf{Active}\; (c \otimes d)} \otimes \in \{\times, +\}$$

Lemma 101. *For any types A and B, $c : A \Rightarrow B$ is either an active or inert cast.*

Lemma 102. *If $c : A \Rightarrow (B \otimes C)$ and* Inert c, *then* Cross c *and $A \equiv D \otimes E$ for some D and E.*

The definition of the functions such as dom are the usual ones, but note that the x parameter plays an important role in this definition. We did not categorize sequence casts as cross casts, so the following functions can omit the cases for $(c\,;d)$:

$$
\begin{aligned}
\mathsf{dom}\; (c \to d)\; \texttt{Cross}\to &= c \\
\mathsf{cod}\; (c \to d)\; \texttt{Cross}\to &= d
\end{aligned}
$$

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$$\mathsf{fst}\,(c \times d)\,\texttt{Cross}\times = c$$
$$\mathsf{snd}\,(c \times d)\,\texttt{Cross}\times = d$$
$$\mathsf{inl}\,(c + d)\,\texttt{Cross}+ = c$$
$$\mathsf{inr}\,(c + d)\,\texttt{Cross}+ = d$$

Lemma 103. *A cast $c : A \Rightarrow b$ is not inert.*

Proposition 104. *The* λC *Calculus is an instance of the* PreCastStruct *structure.*

We define the applyCast function for λC as follows:

$$\mathsf{applyCast} : \forall \Gamma AB.\ (M : \Gamma \vdash A) \to \mathsf{Value}\,M \to (c : A \Rightarrow B) \to \mathsf{Active}\,c \to \Gamma \vdash B$$

$$\begin{array}{llllll}
\mathsf{applyCast} & M & v & \mathsf{id} & a & = & M \\
\mathsf{applyCast} & M\langle G! \rangle & v & H?^{\ell} & a & = & \begin{cases} M & \text{if } G \equiv H \\ \texttt{blame}\,\ell & \text{otherwise} \end{cases} \\
\mathsf{applyCast} & M & v & c\,;d & a & = & M\langle c \rangle \langle d \rangle \\
\mathsf{applyCast} & M & v & c \times d & a & = & \mathsf{eta}\times\, M\, c\, \texttt{Cross}\times \\
\mathsf{applyCast} & M & v & c + d & a & = & \mathsf{eta}+\, M\, c\, \texttt{Cross}+
\end{array}$$

Proposition 105. *The* λC *calculus is an instance of the* CastStruct *structure.*

We import and instantiate the reduction semantics and proof of type safety from Sections 3.6 and 3.8 to obtain the following definition and results.

Definition 106 (Reduction for λC). *The reduction relation $M \longrightarrow N$ for* λC *is the reduction relation of CC(⇒) instantiated with* λC*'s instance of* CastStruct.

Corollary 107 (Preservation for λC). *If $M : \Gamma \vdash A$ and $M \longrightarrow M'$, then $M' : \Gamma \vdash A$.*

Corollary 108 (Progress for λC). *If $M : \emptyset \vdash A$, then*

1. *$M \longrightarrow M'$ for some M',*
2. *$\mathsf{Value}\,M$, or*
3. *$M \equiv \texttt{blame}\,\ell$.*

Let λC′ be the variant of λC obtained by instantiating CC' instead of CC.

5.6.2 Blame-subtyping

The CastBlameSafe predicate for λC′ is the same as the one for EDC′ (Figure 18) except for the additional rule for sequence coercions:

$$\frac{\mathsf{CastBlameSafe}\,c\,\ell \qquad \mathsf{CastBlameSafe}\,d\,\ell}{\mathsf{CastBlameSafe}\,(c; d)\,\ell}$$

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

Lemma 109 (Blame Safety of Cast Operators). *Suppose* CastBlameSafe $c\ \ell$ *and* c *is a cross cast, that is,* x : Cross c.

- *If* $x = \texttt{Cross}\to$, *then* CastBlameSafe (dom $c\ x$) ℓ *and* CastBlameSafe (cod $c\ x$) ℓ.
- *If* $x = \texttt{Cross}\times$, *then* CastBlameSafe (fst $c\ x$) ℓ *and* CastBlameSafe (snd $c\ x$) ℓ.
- *If* $x = \texttt{Cross}+$, *then* CastBlameSafe (inl $c\ x$) ℓ *and* CastBlameSafe (inr $c\ x$) ℓ.

Proposition 110. $\lambda\mathsf{C}'$ *is an instance of the* PreBlameSafe *structure.*

Lemma 111 (applyCast preserves blame safety in $\lambda\mathsf{C}'$). *If* CastBlameSafe $c\ \ell$ *and* a : Active c *and* V *safe for* ℓ *and* v : Value V, *then* (applyCast $V\ v\ c\ a$) *safe for* ℓ.

Proof. The proof is by cases on Active c and inversion on CastBlameSafe $c\ \ell$. □

Proposition 112. $\lambda\mathsf{C}'$ *is an instance of the* BlameSafe *structure.*

We instantiate Theorem 16 with this BlameSafe instance to obtain the Blame-Subtyping Theorem for $\lambda\mathsf{C}'$.

Corollary 113 (Blame-Subtyping Theorem for $\lambda\mathsf{C}'$). *If* $M : \Gamma \vdash A$ *and* M *safe for* ℓ, *then* $\neg(M \longrightarrow^* \texttt{blame}\ \ell)$.

6 Space-Efficient Parameterized Cast Calculus

The cast calculi in Section 5 all suffer from a space efficiency problem (Herman *et al.*, 2010). When a cast is applied to a higher-order value such as a function, these calculi either wrap it inside another function or wrap the cast itself around the value. Either way, the value grows larger. If a value goes through many casts, it can grow larger in an unbounded fashion. This phenomenon causes significant space and time overheads in real programs, for example, changing the worse-cast time complexity of quicksort from $O(n^2)$ to $O(n^3)$ (Takikawa *et al.*, 2016; Kuhlenschmidt *et al.*, 2019).

Herman *et al.* (2010) proposed solving this problem by replacing casts with the coercions of Henglein (1994). Any sequence of coercions can normalize to just three coercions, thereby providing a space-efficient representation. Siek *et al.* (2015*a*) define an algorithm for efficiently normalizing coercions and use it to define the $\lambda\mathsf{S}$ calculus.

Siek & Wadler (2010) propose another approach that compresses a sequence of casts into two casts where the middle type is the least upper bound with respect to precision. The AGT methodology uses a similar representation (Garcia *et al.*, 2016) and was proved space-efficient (Toro & Tanter, 2020; Bañados Schwerter *et al.*, 2021).

In this section, we develop a space-efficient version of the Parameterized Cast Calculus, which we name $SC(\Rightarrow)$. As a sanity check, prove that $SC(\Rightarrow)$ is type-safe, but more importantly, we prove that $SC(\Rightarrow)$ is indeed space-efficient provided that the cast representation is an instance of the structures defined later in this section. In Section 7, we instantiate $SC(\Rightarrow)$ two different ways to reproduce the $\lambda\mathsf{S}$ calculus and to define a new calculus that

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

more directly maps to a compact bit-level encoding. We then instantiate the meta-theory for $SC(\Rightarrow)$ to produce proofs of type safety and space efficiency for both of these calculi.

6.1 Space-efficient values

This subsection is parameterized over the PreCastStruct structure.

To prepare for the definition of space-efficient cast calculi, we define a notion of value that may be wrapped in at most one cast. We accomplish this by stratifying the non-cast values, that is the simple values S, from the values V that may be wrapped in a cast.

$\boxed{S : (\Gamma \vdash A) \to \mathsf{Set}}$

$$\mathtt{S\lambda} : \frac{}{\mathsf{Simple}\,(\lambda M)} \qquad \mathtt{Sconst} : \frac{}{\mathsf{Simple}\,(\$k)} \qquad \mathtt{Spair} : \frac{\mathsf{Value}\,M \qquad \mathsf{Value}\,N}{\mathsf{Simple}\,(\mathtt{cons}\,M\,N)}$$

$$\mathtt{Sinl} : \frac{\mathsf{Value}\,M}{\mathsf{Simple}\,(\mathtt{inl}[B]\,M)} \qquad \mathtt{Sinr} : \frac{\mathsf{Value}\,M}{\mathsf{Simple}\,(\mathtt{inr}[A]\,M)}$$

$\boxed{V : (\Gamma \vdash A) \to \mathsf{Set}}$

$$\mathtt{Vsimp} : \frac{\mathsf{Simple}\,M}{\mathsf{Value}\,M} \qquad \mathtt{Vcast} : \frac{\mathsf{Simple}\,M}{\mathsf{Value}\,(M\langle c\rangle)}\,\mathsf{Inert}\,c$$

Lemma 114. *If* Simple M *and* $M : \Gamma \vdash A$, *then* $A \not\equiv ?$.

Lemma 115 (Canonical Form for type ?). *If* $M : \Gamma \vdash ?$ *and* Value M, *then* $M \equiv M'\langle c\rangle$ *where* $M' : \Gamma \vdash A$, $c : A \Rightarrow ?$, Inert c, *and* $A \not\equiv ?$.

Lemma 116. *If* Simple M *and* $M : \Gamma \vdash b$, *then* $M \equiv k$ *for some* $k : [\![b]\!]$

6.2 The ComposableCasts *structure*

The ComposableCasts structure extends PreCastStruct with two more fields, one for applying a cast to a value (like CastStruct) and one for composing two casts into a single, equivalent cast, for the purposes of achieving space efficiency.[3] It would seem reasonable to have this structure extend CastStruct instead of PreCastStruct, but the problem is that the notion of value is different. Here, we use the definition of Value from Section 6.1.

The two fields of the ComposableCasts are

$$\begin{aligned}&\mathsf{applyCast} : \forall \Gamma AB.\,(M : \Gamma \vdash A) \to \mathsf{SimpleValue}\,M \to (c : A \Rightarrow B)\\ &\qquad\qquad \to \mathsf{Active}\,c \to \Gamma \vdash B\\ &- \fatsemi - : \forall ABC.\,A \Rightarrow B \to B \Rightarrow C \to A \Rightarrow C\end{aligned}$$

[3] The ComposableCasts structure is named EfficientCastStruct in the Agda development.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$$\boxed{\Gamma \vdash A \rightarrowtail B}$$

$$(\Box\ -) : \frac{\Gamma \vdash A}{\Gamma \vdash (A \to B) \rightarrowtail B} \qquad (-\ \Box) : \frac{M : \Gamma \vdash (A \to B)}{\Gamma \vdash A \rightarrowtail B}\,\mathsf{Value}\, M$$

$$\mathtt{if}\,\Box\ -\ -\ : \frac{\Gamma \vdash A \quad \Gamma \vdash A}{\Gamma \vdash \mathtt{Bool} \rightarrowtail A}$$

$$\mathtt{cons}\ -\ \Box : \frac{M : \Gamma \vdash A}{\Gamma \vdash B \rightarrowtail A \times B}\,\mathsf{Value}\, M \qquad \mathtt{cons}\ \Box\ - : \frac{\Gamma \vdash B}{\Gamma \vdash A \rightarrowtail A \times B}$$

$$\pi_i\,\Box : \frac{}{\Gamma \vdash A_1 \times A_2 \rightarrowtail A_i}$$

$$\mathtt{inl}[B]\,\Box : \frac{}{\Gamma \vdash A \rightarrowtail A \times B} \qquad \mathtt{inr}[A]\,\Box : \frac{}{\Gamma \vdash B \rightarrowtail A \times B}$$

$$\mathtt{case}\,\Box\ -\ -\ : \frac{\Gamma \vdash A \rightarrowtail C \quad \Gamma \vdash B \rightarrowtail C}{\Gamma \vdash A + B \rightarrowtail C}$$

$$\boxed{plug : \forall \Gamma AB.\, (\Gamma \vdash A) \to (\Gamma \vdash A \rightarrowtail B) \to (\Gamma \vdash B)}$$

$$\begin{aligned}
plug\ L\ (\Box\ M) &= (L\ M)\\
plug\ M\ (L\ \Box) &= (L\ M)\\
plug\ L\ (\mathtt{if}\,\Box\,M\,N) &= \mathtt{if}\,L\,M\,N\\
plug\ N (\mathtt{cons}\,M\,\Box) &= \mathtt{cons}\,M\,N\\
plug\ M (\mathtt{cons}\,\Box\,N) &= \mathtt{cons}\,M\,N\\
plug\ M (\pi_i\,\Box) &= \pi_i\,M\\
plug\ M (\mathtt{inl}[B]\,\Box) &= \mathtt{inl}[B]\,M\\
plug\ M (\mathtt{inr}[A]\,\Box) &= \mathtt{inr}[A]\,M\\
plug\ L (\mathtt{case}\,\Box\,M\,N) &= \mathtt{case}\,L\,M\,N
\end{aligned}$$

Fig. 19. Frames of $SC(\Rightarrow)$.

6.3 Reduction semantics of $SC(\Rightarrow)$

This section is parameterized by ComposableCasts and defines the Space-Efficient Parameterized Cast Calculus, written $SC(\Rightarrow)$. The syntax is the same as that of the Parameterized Cast Calculus (Figure 5).

The frames of $SC(\Rightarrow)$ and the *plug* function are defined in Figure 19. The definitions are quite similar to those of the Parameterized Cast Calculus (Figure 6), with the notable omission of a frame for casts, which are handled by special congruence rules.

A space-efficient reduction semantics must include a reduction rule for compressing adjacent casts to prevent the growth of long sequences of them. In particular, the (`compose`) rule compresses two adjacent casts into a single cast:

$$M\langle c\rangle\langle d\rangle \longrightarrow M\langle c \fatsemi d\rangle$$

Furthermore, the semantics must ensure that this reduction rule is triggered frequently enough to prevent long sequences from forming. To date, the way to accomplish this in a reduction semantics has been to define evaluation contexts in a subtle way, with two mutual definitions (Herman *et al.*, 2007, 2010; Siek & Wadler, 2010; Siek *et al.*, 2015a). Here, we take a different approach that we believe is simpler to understand and that fits into using frames to control evaluation order. The idea is to parameterize the reduction relation according to whether a reduction rule can fire in any context or only in non-cast contexts, that is, when the immediately enclosing term is not a cast. We define reduction context RedCtx as follows. (It is isomorphic to the Booleans but with more specific names.)

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$\boxed{ctx : \mathsf{RedCtx}}$

$$\texttt{Any} : \frac{}{\mathsf{RedCtx}} \qquad \texttt{NonCast} : \frac{}{\mathsf{RedCtx}}$$

So the reduction relation takes the form:

$$ctx \vdash M \longrightarrow N$$

To prevent reducing under a sequence of two or more casts, the congruence rule for casts (ξ-cast) requires a non-cast context. Further, the inner reduction must be OK with a cast context (i.e. Any context). The congruence rule (ξ) for all other language features can fire in any context, and the inner reduction can require either any context or non-cast contexts. The rule for composing two casts can only fire in a non-cast context, which enforces an outside-in strategy for compressing sequences of casts. For the same reason, the rule for applying a cast to a value can only fire in a non-cast context. All other reduction rules can fire in any context. The reduction semantics for $SC(\Rightarrow)$ is defined in Figure 20.

6.4 Type safety of SC(⇒)

Our terms are intrinsically typed, so the fact that Agda checked the definition in Figure 20 gives us Preservation.

Theorem 117 (Preservation). *If* $M : \Gamma \vdash A$ *and* $M \longrightarrow M'$, *then* $M' : \Gamma \vdash A$.

Next, we prove Progress. First we define the following predicate for identifying when a term is a cast and prove a lemma about switching from NonCast to Any when the redex is not a cast.

$\boxed{\mathsf{IsCast}\ M}$

$$\frac{}{\mathsf{IsCast}\ (M\langle c\rangle)}$$

Lemma 118. *If* $\mathsf{NonCast} \vdash M \longrightarrow M'$, *then* $\mathsf{IsCast}\ M$.

Theorem 119 (Progress). *If* $M : \emptyset \vdash A$, *then*

1. $ctx \vdash M \longrightarrow M'$ *for some* M' *and ctx,*
2. $\mathsf{Value}\ M$, *or*
3. $M \equiv \texttt{blame}\ \ell$.

Proof. The proof is quite similar to that of Theorem 14 except in the case for casts, so we explain just that case here.

Case $M\langle c\rangle$ The induction hypothesis for M yields three subcases.

Subcase $ctx \vdash M \longrightarrow M'$**.** Suppose $ctx = \texttt{Any}$. By rule (ξ-cast), we conclude that

$$\texttt{Any} \vdash M\langle c\rangle \longrightarrow M'\langle c\rangle$$

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$\boxed{ctx \vdash M \longrightarrow N}$

$$\frac{ctx \vdash M \longrightarrow M'}{\texttt{Any} \vdash plug\, M\, F \longrightarrow plug\, M'\, F} \quad (\xi)$$

$$\frac{\texttt{Any} \vdash M \longrightarrow M'}{\texttt{NonCast} \vdash M\langle c\rangle \longrightarrow M'\langle c\rangle} \quad (\xi\texttt{-cast})$$

$$\frac{}{\texttt{Any} \vdash plug\,(\texttt{blame}\,\ell)\, F \longrightarrow \texttt{blame}\,\ell} \quad (\xi\texttt{-blame})$$

$$\frac{}{\texttt{NonCast} \vdash (\texttt{blame}\,\ell)\langle c\rangle \longrightarrow \texttt{blame}\,\ell} \quad (\xi\texttt{-cast-blame})$$

$$\frac{}{\texttt{NonCast} \vdash S\langle c\rangle \longrightarrow \mathrm{applyCast}\, S\, c\, a}\; a : \mathrm{Active}\, c \quad (\texttt{cast})$$

$$\frac{}{\texttt{NonCast} \vdash M\langle c\rangle\langle d\rangle \longrightarrow M\langle c \fatsemi d\rangle} \quad (\texttt{compose})$$

$$\frac{}{\texttt{Any} \vdash V\langle c\rangle\, W \longrightarrow (V\, W\langle \mathrm{dom}\, c\, x\rangle)\langle \mathrm{cod}\, c\, x\rangle}\; x : \mathrm{Cross}\, c \quad (\texttt{fun-cast})$$

$$\frac{}{\texttt{Any} \vdash \texttt{fst}\,(V\langle c\rangle) \longrightarrow (\texttt{fst}\, V)\langle \mathsf{fst}\, c\, x\rangle}\; x : \mathrm{Cross}\, c \quad (\texttt{fst-cast})$$

$$\frac{}{\texttt{Any} \vdash \texttt{snd}\,(V\langle c\rangle) \longrightarrow (\texttt{snd}\, V)\langle \mathsf{snd}\, c\, x\rangle}\; x : \mathrm{Cross}\, c \quad (\texttt{snd-cast})$$

$$\frac{}{\texttt{Any} \vdash \texttt{case}\,(V\langle c\rangle)\, W_1\, W_2 \longrightarrow \texttt{case}\, V\, W_1'\, W_2'} \quad (\texttt{case-cast})$$

$$x : \mathrm{Cross}\, c$$
$$\text{where} \quad W_1' = \lambda(\mathsf{rename}\,\mathsf{S}\, W_1)\,(\mathsf{Z}\langle \mathsf{inl}\, c\, x\rangle)$$
$$W_2' = \lambda(\mathsf{rename}\,\mathsf{S}\, W_2)\,(\mathsf{Z}\langle \mathsf{inr}\, c\, x\rangle)$$

$$\frac{}{\texttt{Any} \vdash (\lambda M)\, V \longrightarrow M[V]} \quad (\beta)$$

$$\frac{}{\texttt{Any} \vdash \texttt{if}\,\$\texttt{true}\, M\, N \longrightarrow M} \quad (\beta\texttt{-true})$$

$$\frac{}{\texttt{Any} \vdash \texttt{if}\,\$\texttt{false}\, M\, N \longrightarrow N} \quad (\beta\texttt{-false})$$

$$\frac{}{\texttt{Any} \vdash \texttt{fst}\,(\texttt{cons}\, V\, W) \longrightarrow V} \quad (\beta\texttt{-fst})$$

$$\frac{}{\texttt{Any} \vdash \texttt{snd}\,(\texttt{cons}\, V\, W) \longrightarrow W} \quad (\beta\texttt{-snd})$$

$$\frac{}{\texttt{Any} \vdash \texttt{case}\,(\texttt{inl}\, V)\, L\, M \longrightarrow L\, V} \quad (\beta\texttt{-caseL})$$

$$\frac{}{\texttt{Any} \vdash \texttt{case}\,(\texttt{inr}\, V)\, L\, M \longrightarrow M\, V} \quad (\beta\texttt{-caseR})$$

$$\frac{}{\texttt{Any} \vdash k\, k' \longrightarrow [\![k]\!]\,[\![k']\!]} \quad (\delta)$$

Fig. 20. Reduction for the Space-Efficient Parameterized Cast Calculus $SC(\Rightarrow)$.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

On the other hand, suppose $ctx = \texttt{NonCast}$. By Lemma 118, $\mathsf{IsCast}\, M$, so we have $M \equiv M_1\langle d\rangle$. By rule (`compose`), we conclude that

$$\texttt{NonCast} \vdash M_1\langle d\rangle\langle c\rangle \longrightarrow M_1\langle d \fatsemi c\rangle$$

Subcase $M \equiv \texttt{blame}\ \ell$**.** By rule ($\xi$-`cast-blame`), we conclude that

$$\texttt{NonCast} \vdash (\texttt{blame}\ \ell)\langle c\rangle \longrightarrow \texttt{blame}\ \ell$$

Subcase $\mathsf{Value}\, M$**.** Here, we use the ActiveOrInert field of the PreCastStruct on the cast c. Suppose c is active, so we have $a : \mathsf{Active}\, c$. By rule (`cast`), using $\mathsf{Value}\, M$, we conclude that

$$\texttt{NonCast} \vdash M\langle c\rangle \longrightarrow \mathsf{applyCast}\, M\ c\ a$$

Suppose c is inert. From $\mathsf{Value}\, M$, we know that M is either a simple value or a cast. If M is a simple value, then we conclude that $\mathsf{Value}\, M\langle c\rangle$. Otherwise, $M \equiv M_1\langle d\rangle$ and we conclude by rule (`compose`):

$$\texttt{NonCast} \vdash M_1\langle d\rangle\langle c\rangle \longrightarrow M_1\langle d \fatsemi c\rangle$$

□

6.5 Space efficiency of SC(⇒)

We follow the space efficiency proof of Herman *et al*. (2010) but refactor it into two parts: (1) generic lemmas about the reduction of $SC(\Rightarrow)$ that appear in this section and (2) lemmas about specific casts and coercions, which appear in Section 7. We clarify a misleading statement by Herman *et al*. (2010) and fill in details needed to mechanize the proof in Agda.

The theorem we aim to prove is that, during execution, the program's size is bounded above by the program's idealized size multiplied by a constant. The idealized size does not include any of the casts. The following are excerpts from the definitions of real-size and ideal-size :

$$\ldots$$
$$\mathsf{real\text{-}size}(M\langle c\rangle) = \mathsf{size}(c) + \mathsf{real\text{-}size}(M)$$
$$\ldots$$
$$\mathsf{ideal\text{-}size}(M\langle c\rangle) = \mathsf{ideal\text{-}size}(M)$$

We shall prove that the size of every cast is bounded above by a constant, so we can simplify some of the technical development by using the following alternative definition of size that uses 1 as the size of each cast:

$$\mathsf{size}(M\langle c\rangle) = 1 + \mathsf{size}(M)$$

Regarding reduction of $SC(\Rightarrow)$, the key property is that the reduction rules prevent the accumulation of long sequences of adjacent casts. In their proof, Herman *et al*. (2010) state that there is never a coercion adjacent to another coercion.

> "During evaluation, the [E-CCAST] rule prevents nesting of adjacent coercions in any term in the evaluation context, redex, or store. Thus the number of coercions in the program state is proportional to the size of the program state."

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$$\boxed{n \mid b \vdash M \text{ ok}}$$

$$\texttt{SCast1}: \frac{n \mid \mathit{false} \vdash M \text{ ok} \quad n \leq 2}{n+1 \mid \mathit{false} \vdash M\langle c \rangle \text{ ok}} \qquad \texttt{SCast2}: \frac{n \mid \mathit{true} \vdash M \text{ ok} \quad n \leq 1}{n+1 \mid \mathit{true} \vdash M\langle c \rangle \text{ ok}}$$

$$\texttt{SVar}: \frac{}{1 \mid b \vdash \text{`}x \text{ ok}} \qquad \frac{n \mid \mathit{true} \vdash N \text{ ok}}{0 \mid b \vdash \lambda N \text{ ok}} \qquad \frac{n \mid b \vdash L \text{ ok} \quad m \mid b \vdash M \text{ ok}}{0 \mid b \vdash L\ M \text{ ok}}$$

$$\frac{}{0 \mid b \vdash \$k \text{ ok}} \qquad \frac{n \mid b \vdash L \text{ ok} \quad m \mid \mathit{true} \vdash M \text{ ok} \quad k \mid \mathit{true} \vdash N \text{ ok}}{0 \mid b \vdash \texttt{if}_\ell \text{ ok}\, L\, M\, N}$$

$$\frac{n \mid b \vdash M \text{ ok} \quad m \mid b \vdash N \text{ ok}}{0 \mid b \vdash \texttt{cons}\, M\, N \text{ ok}} \qquad \frac{n \mid b \vdash M \text{ ok}}{0 \mid b \vdash \texttt{fst}\, M \text{ ok}} \qquad \frac{n \mid b \vdash M \text{ ok}}{0 \mid b \vdash \texttt{snd}\, M \text{ ok}}$$

$$\frac{n \mid b \vdash M \text{ ok}}{0 \mid b \vdash \texttt{inl}[B]\, M \text{ ok}} \qquad \frac{n \mid b \vdash M \text{ ok}}{0 \mid b \vdash \texttt{inr}[B]\, M \text{ ok}}$$

$$\frac{n \mid b \vdash L \text{ ok} \quad m \mid \mathit{true} \vdash M \text{ ok} \quad k \mid \mathit{true} \vdash N \text{ ok}}{0 \mid b \vdash \texttt{case}\, L\, M\, N \text{ ok}} \qquad \frac{}{0 \mid b \vdash \texttt{blame}\, \ell \text{ ok}}$$

Fig. 21. The Size Predicate limits the number of adjacent casts.

Of course, for the rule [E-CCAST] to apply in the first place, there must be two adjacent coercions. So perhaps we could amend the statement of Herman *et al*. (2010) to instead say that there are never more than two. However, even that would be technically incorrect. Consider the following example that begins with three separated coercions but a β-reduction brings them together:

$$((\lambda\ \text{`}0\langle \texttt{Int?} \rangle)\,(1\langle \texttt{Int!} \rangle))\langle \texttt{Int!} \rangle \longrightarrow \$1\langle \texttt{Int!} \rangle \langle \texttt{Int?}^{\ell} \rangle \langle \texttt{Int!} \rangle$$

This turns out to be the worst-case scenario. In the following, we prove that the [E-CCAST] rule, that is, the (`compose`) rule in this article, together with rules about the order of evaluation, prevent nesting of more than three adjacent coercions.

We define the Size Predicate on terms in Figure 21 which only includes terms with no more than three adjacent coercions. The judgment is written $n \mid b \vdash M$ ok where M is a term, n counts the number of cast application expressions at the top of the term, and b indicates with *true* or *false* whether this term is in a delayed context, that is, inside a λ-abstraction or a branch of a conditional expression. The above example with three coercions satisfies the predicate when outside of a delayed context:

$$3 \mid \mathit{false} \vdash \$1\langle \texttt{Int!} \rangle \langle \texttt{Int?}^{\ell} \rangle \langle \texttt{Int!} \rangle \text{ ok}$$

The rule `SCast1` for cast application expressions adds one to the count of adjacent casts and makes sure that the count does not exceed three.

For terms in a delayed context, the rule (SCAST2) restricts the number of adjacent casts to two instead of three. To see why, consider the next example in which there are three adjacent casts inside the λ-abstraction and a β-reduction yields a term with four adjacent casts:

$$((\lambda\ \$8\langle \texttt{Int!} \rangle \langle \texttt{Int?}^{\ell_1} \rangle \langle \texttt{Int!} \rangle)\ 1)\langle \texttt{Int?}^{\ell_2} \rangle \longrightarrow \$8\langle \texttt{Int!} \rangle \langle \texttt{Int?}^{\ell_1} \rangle \langle \texttt{Int!} \rangle \langle \texttt{Int?}^{\ell_2} \rangle$$

The rule `SVar` starts the count at one even though a variable is obviously not a cast application. The reason is that a value substituted for a variable may have one cast at the

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

top. If we did not count variables as one, then a variable could be surrounded by two casts inside of a λ-abstraction which could reduce to a term with four adjacent casts as in the following example:

$$((\lambda\, `0\langle \mathtt{Int}?^{\ell_1}\rangle\langle \mathtt{Int}!\rangle)\,(1\langle \mathtt{Int}!\rangle))\langle \mathtt{Int}?^{\ell_2}\rangle \longrightarrow 1\langle \mathtt{Int}!\rangle\langle \mathtt{Int}?^{\ell_1}\rangle\langle \mathtt{Int}!\rangle\langle \mathtt{Int}?^{\ell_2}\rangle$$

We turn to the proof of the space consumption theorem, starting with the necessary lemmas.

The Size Predicate guarantees that the number of adjacent casts is less than or equal to 3.

Lemma 120 (Maximum of 3 Adjacent Casts). *If* $n \mid b \vdash M$ ok*, then* $n \leq 3$.

The Size Predicate guarantees that a term's size is bounded above by its ideal size multiplied by a constant, plus 3.

Lemma 121 (Size Predicate and Ideal Size).
If $n \mid b \vdash M$ ok *then* $\mathsf{size}(M) \leq 10 \cdot \mathit{ideal\text{-}size}(M) + 3$.

The compilation of source programs (GTCL) to the cast calculus produces terms that satisfy the Size Predicate.

Lemma 122 (Cast Insertion Size).
If $\Gamma \vdash_G M : A$ *then* $n \mid b \vdash \mathcal{C}[\![M]\!]$ ok *for some* $n \leq 1$.

Reduction preserves the Size Predicate. The proof of this lemma involves a number of technical lemmas about substitution, evaluation contexts, and values, which can be found in the Agda formalization.

Lemma 123 (Size Preservation). *If* $M : \Gamma \vdash A$ *and* $M' : \Gamma \vdash A$ *and* $n \mid \mathit{false} \vdash M$ ok *and* $M \longrightarrow M'$*, then* $m \mid \mathit{false} \vdash M'$ ok *for some* m.

The next piece needed for the space efficiency theorem is to place a bound on the size of the casts. We require their size to be bounded by their height, as in Herman *et al*. (2010), so it remains to show that the heights of all the coercions does not grow during execution. To prove this, we must place further requirements on the specific cast calculi, which we formulate as a structure named CastHeight in Figure 22 that extends the ComposableCasts structure in Section 6.

We define the *cast height* of a term, written c-height(M), to be the maximum of the heights of the all the casts in term M. The cast height of a term monotonically decreases under reduction.

Lemma 124 (Preserve Height). *If* $M : \Gamma \vdash A$*,* $M' : \Gamma \vdash A$*, and* $\mathit{ctx} \vdash M \longrightarrow M'$*, then* $\mathsf{c\text{-}height}(M') \leq \mathsf{c\text{-}height}(M)$.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$$\mathsf{height} : \forall AB.\ (c : A \Rightarrow B) \to \mathbb{N}$$

$$\mathsf{size} : \forall AB.\ (c : A \Rightarrow B) \to \mathbb{N}$$

$$\mathsf{compose\text{-}height} : \begin{array}{l} \forall ABC.\ (c : A \Rightarrow B) \to (d : B \Rightarrow C) \\ \to \mathsf{height}(c\ ;\ d) \leq \max(\mathsf{height}(c), \mathsf{height}(d)) \end{array}$$

$$\mathsf{applyCastSize} : \begin{array}{l} \forall \Gamma ABn.\ (M : \Gamma \vdash A) \to (c : A \Rightarrow B) \\ \to n \mid \mathit{false} \vdash M\ \mathsf{ok} \to (v : \mathsf{Value}\ M) \\ \to \exists m.\ m \mid \mathit{false} \vdash (\mathsf{applyCast}\ M\ v\ c)\ \mathsf{ok} \times m \leq 2 + n \end{array}$$

$$\mathsf{applyCastHeight} : \begin{array}{l} \forall \Gamma ABV.\ (v : \mathsf{Value}\ V) \\ \to \mathsf{height}(\mathsf{applyCast}\ V\ v\ c) \leq \max(\mathsf{height}(V), \mathsf{height}(c)) \end{array}$$

$$\mathsf{dom\text{-}height} : \begin{array}{l} (c : A \to B \Rightarrow C \to D) \to (x : \mathsf{Cross}\ c) \\ \to \mathsf{height}(\mathsf{dom}\ c\ x) \leq \mathsf{height}(c) \end{array}$$

$$\mathsf{cod\text{-}height} : \begin{array}{l} (c : A \to B \Rightarrow C \to D) \to (x : \mathsf{Cross}\ c) \\ \to \mathsf{height}(\mathsf{cod}\ c\ x) \leq \mathsf{height}(c) \end{array}$$

$$\mathsf{fst\text{-}height} : \begin{array}{l} (c : A \times B \Rightarrow C \times D) \to (x : \mathsf{Cross}\ c) \\ \to \mathsf{height}(\mathsf{fst}\ c\ x) \leq \mathsf{height}(c) \end{array}$$

$$\mathsf{snd\text{-}height} : \begin{array}{l} (c : A \times B \Rightarrow C \times D) \to (x : \mathsf{Cross}\ c) \\ \to \mathsf{height}(\mathsf{snd}\ c\ x) \leq \mathsf{height}(c) \end{array}$$

$$\mathsf{inl\text{-}height} : \begin{array}{l} (c : A + B \Rightarrow C + D) \to (x : \mathsf{Cross}\ c) \\ \to \mathsf{height}(\mathsf{inl}\ c\ x) \leq \mathsf{height}(c) \end{array}$$

$$\mathsf{inr\text{-}height} : \begin{array}{l} (c : A + B \Rightarrow C + D) \to (x : \mathsf{Cross}\ c) \\ \to \mathsf{height}(\mathsf{inr}\ c\ x) \leq \mathsf{height}(c) \end{array}$$

$$\mathsf{size\text{-}height} : \exists k_1 k_2.\ \forall AB.\ (c : A \Rightarrow B) \to \mathsf{size}(c) + k_1 \leq k_2 * 2^{\mathsf{height}(c)}$$

Fig. 22. CastHeight extends ComposableCasts (Section 6.2).

The size of any cast c is bounded above by $k_2 \cdot 2^{\mathsf{height}(c)}$ (according to the size-height member of CastHeight), so the real-size of a term is bounded above by $k_2 \cdot 2^{\mathsf{c\text{-}height}(M)}$ multiplied by its size.

Lemma 125 (Real Size Bounded by Size).
If $M : \Gamma \vdash A$, *then* $\mathsf{real\text{-}size}(M) \leq k_2 \cdot 2^{\mathsf{c\text{-}height}(M)} \cdot \mathsf{size}(M)$.

With the above lemmas in place, we proceed with the main theorem. The size of the program (including the casts) is bounded above by its ideal size (not including casts) multiplied by a constant.

Theorem 126 (Space Consumption). *If* $M : \Gamma \vdash A$, *then there exists* c *such that for any* $M' : \Gamma \vdash A$ *where* $\mathit{ctx} \vdash \mathcal{C}[\![M]\!] \longrightarrow^* M'$, *we have* $\mathsf{real\text{-}size}(M') \leq c \cdot \mathit{ideal\text{-}size}(M')$.

Proof. (The formal proof of this theorem is in the Agda development. Here we give a proof that is less formal but easier to read.) By Lemma 122, there is an n such that

$$n \mid \mathit{false} \vdash \mathcal{C}[\![M]\!]\ \mathsf{ok}$$

Using Lemma 123 and an induction on the reduction sequence $\mathit{ctx} \vdash \mathcal{C}[\![M]\!] \longrightarrow^* M'$, there is an m such that

$$m \mid \mathit{false} \vdash M\ \mathsf{ok}$$

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

We establish the conclusion of the theorem by the following inequational reasoning:

$$\begin{aligned}
\mathsf{real\text{-}size}(M') &\le k_2 \cdot 2^{\mathsf{c\text{-}height}(M')} \cdot \mathsf{size}(M') && \text{by Lemma 125}\\
&\le (k_2 \cdot 2^{\mathsf{c\text{-}height}(M')}) \cdot (3 + 10 \cdot \mathsf{ideal\text{-}size}(M')) && \text{by Lemma 121}\\
&\le 3k_2 \cdot 2^{\mathsf{c\text{-}height}(M')} + 10k_2 \cdot 2^{\mathsf{c\text{-}height}(M')} \cdot \mathsf{ideal\text{-}size}(M')\\
&\le 3k_2 \cdot 2^{\mathsf{c\text{-}height}(C[\![M]\!])} + 10k_2 \cdot 2^{\mathsf{c\text{-}height}(C[\![M]\!])} \cdot \mathsf{ideal\text{-}size}(M') && \text{by Lemma 124}\\
&\le 13k_2 \cdot 2^{\mathsf{c\text{-}height}(C[\![M]\!])} \cdot \mathsf{ideal\text{-}size}(M')
\end{aligned}$$

So we choose $13k_2 \cdot 2^{\mathsf{c\text{-}height}(C[\![M]\!])}$ as the witness for c. □

We observe that the space consumption theorem of Herman *et al*. (2010) (as well as the above theorem) has a limitation in that it does not prevent a cast calculus that uses the eta cast reduction rules, as discussed in Section 3.3, from consuming an unbounded amount of space. To capture the space used by the eta cast rules, one would need to mark the terms introduced by the eta cast rules so that they can be excluded from the ideal-size of a term. We do not pursue that direction at this time, but note that the calculi that we introduce in the next section do not use eta cast rules.

7 Space-efficient cast calculi

We instantiate the Efficient Parameterized Calculus $SC(\Rightarrow)$ with two different instances of the ComposableCasts and CastHeight to obtain definitions and proofs of type safety and space efficiency for two cast calculi: λS of Siek *et al*. (2015*a*) and the hypercoercions of Lu *et al*. (2020).

7.1 λS

The cast representation in λS are coercions in a particular canonical form, with a three-part grammar consisting of top-level coercions, intermediate coercions, and ground coercions, defined in Figure 23. A top-level coercion is an identity cast, a projection followed by an intermediate coercion, or just an intermediate coercion. An intermediate coercion is a ground coercion followed by an injection, just a ground coercion, or a failure coercion. A ground coercion is an identity on base type or a cross cast between function, pair, or sum types.

The cast constructor is also defined in Figure 23.

7.1.1 Reduction semantics and type safety

Casts between function, pair, and sum types are categorized as cross casts.

$\boxed{\mathsf{Cross}\ c}$

$$\mathtt{Cross}\otimes : \frac{}{\mathsf{Cross}\ (c \otimes d)} \quad \otimes \in \{\to, \times, +\}$$

The inert casts include casts between function types, injections, and the failure coercion.

$\boxed{\mathsf{Inert}\ c}$

$$\frac{}{\mathsf{Inert}\ c \to d} \qquad \frac{}{\mathsf{Inert}\ g; G!} \qquad \frac{}{\mathsf{Inert}\ \bot^{\ell}}$$

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$\boxed{c, d : A \Rightarrow B}$

$$\mathsf{id} : \frac{}{? \Rightarrow ?} \qquad (H?^{\ell}; -) : \frac{H \Rightarrow_i B}{? \Rightarrow B} \qquad - : \frac{A \Rightarrow_i B}{A \Rightarrow B}$$

$\boxed{i : A \Rightarrow_i B}$

$$-; G! : \frac{A \Rightarrow_g G}{A \Rightarrow_i ?} \qquad - : \frac{A \Rightarrow_g B}{A \Rightarrow_i B} \qquad \bot^{\ell} : \frac{}{A \Rightarrow_i B}$$

$\boxed{g, h : A \Rightarrow_g B}$

$$\mathsf{id} : \frac{}{b \Rightarrow_g b} \qquad - \to - : \frac{C \Rightarrow A \quad B \Rightarrow D}{A \to B \Rightarrow_g C \to D}$$

$$- \times - : \frac{A \Rightarrow C \quad B \Rightarrow D}{A \times B \Rightarrow_g C \times D} \qquad - + - : \frac{A \Rightarrow C \quad B \Rightarrow D}{A + B \Rightarrow_g C + D}$$

$\boxed{(\!|A \Rightarrow B|\!)^{\ell} = c, (\!|A \Rightarrow G|\!)^{\ell} = g, (\!|H \Rightarrow A|\!)^{\ell} = g}$

$$\begin{aligned}
(\!|? \Rightarrow ?|\!)^{\ell} &= \mathsf{id} \\
(\!|A \Rightarrow ?|\!)^{\ell} &= (\!|A \Rightarrow G|\!)^{\ell}; G! && \text{where } G = \mathsf{gnd}\ A, A \neq ? \\
(\!|? \Rightarrow A|\!)^{\ell} &= H?^{\ell}; (\!|H \Rightarrow A|\!)^{\ell} && \text{where } H = \mathsf{gnd}\ A, A \neq ? \\
(\!|b \Rightarrow b|\!)^{\ell} &= \mathsf{id} \\
(\!|A \to B \Rightarrow A' \to B'|\!)^{\ell} &= (\!|A' \Rightarrow A|\!)^{\ell} \to (\!|B \Rightarrow B'|\!)^{\ell} \\
(\!|A \times B \Rightarrow A' \times B'|\!)^{\ell} &= (\!|A' \Rightarrow A|\!)^{\ell} \times (\!|B \Rightarrow B'|\!)^{\ell} \\
(\!|A + B \Rightarrow A' + B'|\!)^{\ell} &= (\!|A' \Rightarrow A|\!)^{\ell} + (\!|B \Rightarrow B'|\!)^{\ell}
\end{aligned}$$

Fig. 23. Coercions of λS and its cast constructor.

There are five kinds of active coercions: the identity on ?, projections, failures, cross casts on pairs and sums, and identity on base types.

$\boxed{\mathsf{Active}\ c}$

$$\mathtt{Aid} : \frac{}{\mathsf{Active}\ \mathsf{id}} \qquad \mathtt{Aproj} : \frac{}{\mathsf{Active}\ (G?^{\ell}; i)} \qquad \mathtt{Afail} : \frac{}{\mathsf{Active}\ \bot^{\ell}}$$

$$\mathtt{A}\otimes : \frac{\otimes \in \{\times, +\}}{\mathsf{Active}\ (c \otimes d)} \qquad \mathtt{Abase} : \frac{}{\mathsf{Active}\ \mathsf{id}}$$

Lemma 127. *For any types A and B, $c : A \Rightarrow B$ is either an active or inert cast.*

Lemma 128. *If $c : A \Rightarrow (B \otimes C)$ and $\mathsf{Inert}\ c$, then $\mathsf{Cross}\ c$ and $A \equiv D \otimes E$ for some D and E.*

The definition of dom, etc., for λS is given below:

$$\begin{aligned}
\mathsf{dom}\ (c \to d)\ \mathtt{Cross}{\to} &= c \\
\mathsf{cod}\ (c \to d)\ \mathtt{Cross}{\to} &= d \\
\mathsf{fst}\ (c \times d)\ \mathtt{Cross}{\times} &= c \\
\mathsf{snd}\ (c \times d)\ \mathtt{Cross}{\times} &= d \\
\mathsf{inl}\ (c + d)\ \mathtt{Cross}{+} &= c \\
\mathsf{inr}\ (c + d)\ \mathtt{Cross}{+} &= d
\end{aligned}$$

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

Lemma 129. *A cast $c : A \Rightarrow b$ is not inert.*

Proposition 130. λS *is an instance of the* PreCastStruct *structure.*

To support space efficiency, we define a composition operator for the coercions of λS. The operator uses two auxiliary versions of the operator for intermediate and ground coercions. The operator that composes an intermediate coercion with a coercion always yields an intermediate coercion. The operator that composes two ground coercions always returns a ground coercion. Agda does not automatically prove termination for this set of mutually recursive functions, so we manually prove termination, using the sum of the sizes of the two coercions as the measure.

$\boxed{c \fatsemi d}$ $\boxed{i \fatsemi d}$ $\boxed{g \fatsemi h}$

$$
\begin{aligned}
\mathsf{id} \fatsemi d &= d \\
(G?^{\ell}; i) \fatsemi d &= G?^{\ell}; (i \fatsemi d)
\end{aligned}
$$

$$
\begin{aligned}
(g; G!) \fatsemi \mathsf{id} &= g; G! \\
g \fatsemi (h; H!) &= (g \fatsemi h); H! \\
(g; G!) \fatsemi (G?^{\ell}; i) &= g \fatsemi i \\
(g; G!) \fatsemi (H?^{\ell}; i) &= \bot^{\ell} \qquad \text{if } G \neq H \\
\bot^{\ell} \fatsemi d &= \bot^{\ell} \\
g \fatsemi \bot^{\ell} &= \bot^{\ell}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{id} \fatsemi \mathsf{id} &= \mathsf{id} \\
(c_1 \to d_1) \fatsemi (c_2 \to d_2) &= (c_2 \fatsemi c_1) \to (d_1 \fatsemi d_2) \\
(c_1 \times d_1) \fatsemi (c_2 \times d_2) &= (c_1 \fatsemi c_2) \times (d_1 \fatsemi d_2) \\
(c_1 + d_1) \fatsemi (c_2 + d_2) &= (c_1 \fatsemi c_2) + (d_1 \fatsemi d_2)
\end{aligned}
$$

We define applyCast for λS by cases on the coercion.

$$\mathsf{applyCast} : \forall \Gamma AB.\, (M : \Gamma \vdash A) \to \mathsf{SimpleValue}\, M \to (c : A \Rightarrow B) \to \mathsf{Active}\, c \to \Gamma \vdash B$$

$$
\begin{array}{llllll}
\mathsf{applyCast} & M & v & \mathsf{id} & a & = M \\
\mathsf{applyCast} & M & v & \bot^{\ell} & a & = \mathtt{blame}\ \ell \\
\mathsf{applyCast} & M\langle c \rangle & v & (G?^{\ell}; i) & a & = M\langle c \fatsemi (G?^{\ell}; i) \rangle \\
\mathsf{applyCast} & (\mathtt{cons}\ V_1\ V_2) & v & c \times d & a & = \mathtt{cons}\, (V_1\langle c \rangle)\, (V_2\langle d \rangle) \\
\mathsf{applyCast} & (\mathtt{inl}\ V) & v & c + d & a & = \mathtt{inl}\, (V\langle c \rangle) \\
\mathsf{applyCast} & (\mathtt{inr}\ V) & v & c + d & a & = \mathtt{inr}\, (V\langle d \rangle)
\end{array}
$$

Proposition 131. λS *is an instance of the* ComposableCasts *structure.*

We import and instantiate the reduction semantics and proof of type safety from Section 6 to obtain the following definitions and results for λS.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

Definition 132 (Reduction). *The reduction relation* $ctx \vdash M \longrightarrow_S N$ *of* λS *is the reduction relation of* $SC(\Rightarrow)$ *instantiated with* λS*'s instance of the* ComposableCasts *structure.*

Corollary 133 (Preservation for λS). *If* $M : \Gamma \vdash A$ *and* $M \longrightarrow_S M'$*, then* $M' : \Gamma \vdash A$.

Corollary 134 (Progress for λS). *If* $M : \emptyset \vdash A$*, then*

1. $M \longrightarrow_S M'$ *for some* M',
2. Value M*, or*
3. $M \equiv \texttt{blame}\ \ell$.

7.1.2 Space efficiency

Next, we establish that λS is an instance of the CastHeight structure so that we can apply Theorem 126 (Space Consumption) to obtain space efficiency for λS.

We define the height of a coercion as follows:

$$\begin{aligned}
\mathsf{height}(\mathsf{id}) &= 0 \\
\mathsf{height}(\bot^\ell) &= 0 \\
\mathsf{height}(G?^\ell; i) &= \mathsf{height}(i) \\
\mathsf{height}(g; G!) &= \mathsf{height}(g) \\
\mathsf{height}(c \to d) &= 1 + \max(\mathsf{height}(c), \mathsf{height}(d)) \\
\mathsf{height}(c \times d) &= 1 + \max(\mathsf{height}(c), \mathsf{height}(d)) \\
\mathsf{height}(c + d) &= 1 + \max(\mathsf{height}(c), \mathsf{height}(d))
\end{aligned}$$

The size of a coercion is given by the following definition:

$$\begin{aligned}
\mathsf{size}(\mathsf{id}) &= 0 \\
\mathsf{size}(\bot^\ell) &= 0 \\
\mathsf{size}(G?^\ell; i) &= 2 + \mathsf{size}(i) \\
\mathsf{size}(g; G!) &= 2 + \mathsf{size}(g) \\
\mathsf{size}(c \to d) &= 1 + \mathsf{size}(c) + \mathsf{size}(d) \\
\mathsf{size}(c \times d) &= 1 + \mathsf{size}(c) + \mathsf{size}(d) \\
\mathsf{size}(c + d) &= 1 + \mathsf{size}(c) + \mathsf{size}(d)
\end{aligned}$$

The cast height of the result of applyCast applied to a simple value S and coercion c is less than the max of the cast height of S and the height of c.

Lemma 135. $\mathsf{c\text{-}height}(\mathsf{applyCast}\ S\ c) \leq \max(\mathsf{height}(S), \mathsf{height}(c))$

The dom, cod, fst, snd, inl, and inr operators on coercions all return coercions of equal or lesser height than their input.

Lemma 136.

1. $\mathsf{height}(\mathsf{dom}\ c\ x) \leq \mathsf{height}(c)$
2. $\mathsf{height}(\mathsf{cod}\ c\ x)) \leq \mathsf{height}(c)$

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

3. $\mathsf{height}(\mathsf{fst}\, c\, x) \leq \mathsf{height}(c)$
4. $\mathsf{height}(\mathsf{snd}\, c\, x) \leq \mathsf{height}(c)$
5. $\mathsf{height}(\mathsf{inl}\, c\, x) \leq \mathsf{height}(c)$
6. $\mathsf{height}(\mathsf{inr}\, c\, x) \leq \mathsf{height}(c)$

The size of a coercion c is bounded by $9 \cdot 2^{\mathsf{height}(c)}$. We prove this by simultaneously proving the following three facts about the three kinds of coercions.

Lemma 137.

1. $\mathsf{size}(c) + 5 \leq 9 \cdot 2^{\mathsf{height}(c)}$
2. $\mathsf{size}(i) + 7 \leq 9 \cdot 2^{\mathsf{height}(i)}$
3. $\mathsf{size}(g) + 9 \leq 9 \cdot 2^{\mathsf{height}(g)}$

The above lemmas and definitions establish the following.

Proposition 138. $\lambda\mathsf{S}$ *is an instance of the* CastHeight *structure.*

We apply Theorem 139 (Space Consumption) to obtain the following result for $\lambda\mathsf{S}$.

Corollary 139 (Space Consumption for $\lambda\mathsf{S}$). *If* $M : \Gamma \vdash A$, *then there exists* c *such that for any* $M' : \Gamma \vdash A$ *where* $ctx \vdash \mathcal{C}[\![M]\!] \longrightarrow_S^* M'$, *we have* $\mathsf{real\text{-}size}(M') \leq c \cdot \mathit{ideal\text{-}size}(M')$.

7.2 Hypercoercions

This section develops an alternative formulation of $\lambda\mathsf{S}$ using a new representation, called hypercoercions and written $\lambda\mathsf{H}$, that more directly maps to a compact bit-level encoding. We presented hypercoercions at the Workshop on Gradual Typing (Lu *et al.*, 2020). Hypercoercions are inspired by the supercoercions of Garcia (2013).

The idea behind hypercoercions is to choose a canonical representation in which a coercion always has three parts, a beginning p, middle m, and end i. The beginning part p may be a projection or an identity. The middle part m is a cross cast or an identity cast at base type. The end part i may be an injection, failure, or identity. The definition of hypercoercions, given in Figure 24, factors the definition into these three parts.

The cast constructor is also defined in Figure 24.

7.2.1 Reduction semantics and type safety

A hypercoercion whose middle is a cast between function, pair, or sum types, is a cross cast, provided the hypercoercion begins and ends with the identity.

$\boxed{\mathsf{Cross}\, c}$

$$\mathtt{Cross}\otimes : \frac{}{\mathsf{Cross}\,(\mathsf{id}; c \otimes d; \mathsf{id})} \otimes \in \{\rightarrow, \times, +\}$$

A hypercoercion that begins with identity and ends with an injection to ? is inert. A hypercoercion that begins and ends with identity, but whose middle is a cast between function types, is also insert.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$\boxed{c, d : A \Rightarrow B}\ \boxed{p : A \Rightarrow_p B}\ \boxed{m : A \Rightarrow_m B}\ \boxed{i : A \Rightarrow_i B}$

$$\mathsf{id} : \frac{}{? \Rightarrow ?} \qquad (-;-;-) : \frac{A \Rightarrow_p B \quad B \Rightarrow_m C \quad C \Rightarrow_i D}{A \Rightarrow D}$$

$$\mathsf{id} : \frac{}{A \Rightarrow_p A} \qquad H?^\ell : \frac{}{? \Rightarrow_p H}$$

$$\mathsf{id} : \frac{}{a \Rightarrow_m a} \qquad - \to - : \frac{C \Rightarrow A \quad B \Rightarrow D}{A \to B \Rightarrow_m C \to D}$$

$$- \times - : \frac{A \Rightarrow C \quad B \Rightarrow D}{A \times B \Rightarrow_m C \times D} \qquad - + - : \frac{A \Rightarrow C \quad B \Rightarrow D}{A + B \Rightarrow_m C + D}$$

$$\mathsf{id} : \frac{}{A \Rightarrow_i A} \qquad G! : \frac{}{G \Rightarrow_i ?} \qquad \bot^\ell : \frac{}{A \Rightarrow_i B}$$

$\boxed{(\!|A \Rightarrow B|\!)^\ell = c, (\!|A \otimes B \Rightarrow A' \otimes B'|\!)^\ell_m = m}$

$$\begin{aligned}
(\!|? \Rightarrow ?|\!)^\ell &= \mathsf{id} \\
(\!|A \Rightarrow ?|\!)^\ell &= \mathsf{id}; (\!|A \Rightarrow G|\!)^\ell_m; G! && \text{where } G = \mathsf{gnd}\ A \\
(\!|? \Rightarrow A|\!)^\ell &= H?^\ell; (\!|H \Rightarrow A|\!)^\ell_m; \mathsf{id} && \text{where } H = \mathsf{gnd}\ A \\
(\!|b \Rightarrow b|\!)^\ell &= \mathsf{id}; \mathsf{id}; \mathsf{id} \\
(\!|A \otimes B \Rightarrow A' \otimes B'|\!)^\ell &= \mathsf{id}; (\!|A \otimes B \Rightarrow A' \otimes B'|\!)^\ell_m; \mathsf{id} \\
(\!|A \to B \Rightarrow A' \to B'|\!)^\ell_m &= (\!|A' \Rightarrow A|\!)^\ell \to (\!|B \Rightarrow B'|\!)^\ell \\
(\!|A \times B \Rightarrow A' \times B'|\!)^\ell_m &= (\!|A \Rightarrow A'|\!)^\ell \times (\!|B \Rightarrow B'|\!)^\ell \\
(\!|A + B \Rightarrow A' + B'|\!)^\ell_m &= (\!|A \Rightarrow A'|\!)^\ell + (\!|B \Rightarrow B'|\!)^\ell
\end{aligned}$$

Fig. 24. Hypercoercions.

$\boxed{\mathsf{Inert}\ c}$

$$\frac{}{\mathsf{Inert}\ (\mathsf{id}; m; G!)} \qquad \frac{}{\mathsf{Inert}\ (\mathsf{id}; c \to d; \mathsf{id})}$$

There are four kinds of active hypercoercions. The identity hypercoercion is active, as is a hypercoercion that begins with a projection from ? or ends with a failure coercion. Furthermore, if the hypercoercion begins and ends with identity, and the middle is either the identity or a cast between pair or sum types, then it is active.

$\boxed{\mathsf{Active}\ c}$

$$\mathtt{Aid} : \frac{}{\mathsf{Active}\ \mathsf{id}} \qquad \mathtt{Aproj} : \frac{}{\mathsf{Active}\ (H?^\ell; m; i)} \qquad \mathtt{Afail} : \frac{}{\mathsf{Active}\ (p; m; \bot^\ell)}$$

$$\mathtt{A}\otimes : \frac{\otimes \in \{\times, +\}}{\mathsf{Active}\ (\mathsf{id}; c \otimes d; \mathsf{id})} \qquad \mathtt{Abase} : \frac{}{\mathsf{Active}\ (\mathsf{id}; \mathsf{id}; \mathsf{id})}$$

Lemma 140. *For any types A and B, $c : A \Rightarrow B$ is either an active or inert cast.*

Lemma 141. *If $c : A \Rightarrow (B \otimes C)$ and* Inert *c, then* Cross *c and $A \equiv D \otimes E$ for some D and E.*

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$\boxed{c \fatsemi d}$ $\boxed{m_1 \fatsemi m_2}$

$$
\begin{aligned}
c \fatsemi \mathsf{id} &= c \\
\mathsf{id} \fatsemi (p; m; i) &= p; m; i \\
(p_1; m_1; \mathsf{id}) \fatsemi (\mathsf{id}; m_2; i_2) &= p; (m_1 \fatsemi m_2); i_2 \\
(p_1; m_1; G!) \fatsemi (G?^{\ell}; m_2; i_2) &= p_1; (m_1 \fatsemi m_2); i_2 \\
(p_1; m_1; G!) \fatsemi (H?^{\ell}; m_2; i_2) &= p_1; m_1; \bot^{\ell} && \text{if } G \neq H \\
(p_1; m_1; \bot^{\ell}) \fatsemi (p_2; m_2; i_2) &= p_1; m_1; \bot^{\ell}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{id} \fatsemi \mathsf{id} &= \mathsf{id} \\
(c_1 \to d_1) \fatsemi (c_2 \to d_2) &= (c_2 \fatsemi c_1) \to (d_1 \fatsemi d_2) \\
(c_1 \times d_1) \fatsemi (c_2 \times d_2) &= (c_1 \fatsemi c_2) \times (d_1 \fatsemi d_2) \\
(c_1 + d_1) \fatsemi (c_2 + d_2) &= (c_1 \fatsemi c_2) + (d_1 \fatsemi d_2)
\end{aligned}
$$

Fig. 25. Composition of Hypercoercions.

The definition of dom, etc. for hypercoercions is given below:

$$
\begin{aligned}
\mathsf{dom}\,(\mathsf{id}; c \to d; \mathsf{id})\,\texttt{Cross}{\to} &= c \\
\mathsf{cod}\,(\mathsf{id}; c \to d; \mathsf{id})\,\texttt{Cross}{\to} &= d \\
\mathsf{fst}\,(\mathsf{id}; c \times d; \mathsf{id})\,\texttt{Cross}{\times} &= c \\
\mathsf{snd}\,(\mathsf{id}; c \times d; \mathsf{id})\,\texttt{Cross}{\times} &= d \\
\mathsf{inl}\,(\mathsf{id}; c + d; \mathsf{id})\,\texttt{Cross}{+} &= c \\
\mathsf{inr}\,(\mathsf{id}; c + d; \mathsf{id})\,\texttt{Cross}{+} &= d
\end{aligned}
$$

Lemma 142. *A cast $c : A \Rightarrow b$ is not inert.*

Proposition 143. *Hypercoercions are an instance of the* PreCastStruct *structure.*

To support space efficiency, we define the composition operator on hypercoercions in Figure 25. For the most part, this operator compares the end of the first coercion with the beginning of the second. For example, if the end of the first coercion is an injection $G!$ and the beginning of the second is the corresponding projection $G?^{\ell}$, then the injection and projection are discarded and the resulting hypercoercion is formed using the beginning of the first, the composition of the middles, and the end of the second. The composition operator handles identity coercions and failure coercions as special cases. An identity coercion composed on the left or right is discarded. A failure coercion on the left causes the coercion on the right to be discarded. The composition operator is a mutually recursive with the definition of composition on the middle parts. Thankfully, Agda's termination checker approves of this definition even though the contravariance in function coercions means that it is not technically structurally recursive.

We define the applyCast function for hypercoercions by cases on Active c.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$$\mathsf{applyCast} : \forall \Gamma AB.\, (M : \Gamma \vdash A) \to \mathsf{SimpleValue}\, M \to (c : A \Rightarrow B) \to \mathsf{Active}\, c \to \Gamma \vdash B$$

$$\begin{array}{llllll}
\mathsf{applyCast} & M & v & \mathsf{id} & \mathtt{Aid} & = M \\
\mathsf{applyCast} & M\langle c \rangle & v & (H?^{\ell}; m; i) & \mathtt{Aproj} & = M\langle c \fatsemi (H?^{\ell}; m; i)\rangle \\
\mathsf{applyCast} & M & v & (\mathsf{id}; m; \bot^{\ell}) & \mathtt{Afail} & = \mathtt{blame}\ \ell \\
\mathsf{applyCast} & (\mathtt{cons}\ V_1\ V_2) & v & (\mathsf{id}; c \times d; \mathsf{id}) & \mathtt{A}\times & = \mathtt{cons}\,(V_1\langle c \rangle)\,(V_2\langle d \rangle) \\
\mathsf{applyCast} & (\mathtt{inl}\ V) & v & (\mathsf{id}; c + d; \mathsf{id}) & \mathtt{A}+ & = \mathtt{inl}\,(V\langle c \rangle) \\
\mathsf{applyCast} & (\mathtt{inr}\ V) & v & (\mathsf{id}; c + d; \mathsf{id}) & \mathtt{A}+ & = \mathtt{inr}\,(V\langle d \rangle) \\
\mathsf{applyCast} & M & v & (\mathsf{id}; \mathsf{id}; \mathsf{id}) & \mathtt{Abase} & = M
\end{array}$$

Proposition 144. *Hypercoercions are an instance of the* ComposableCasts *structure.*

We import and instantiate the reduction semantics and proof of type safety from Section 6 to obtain the following definition and results.

Definition 145 (Reduction). *The reduction relation* $ctx \vdash M \longrightarrow_H N$ *of* λH *is the reduction relation of* $SC(\Rightarrow)$ *instantiated with* λH*'s instance of the* ComposableCasts *structure.*

Corollary 146 (Preservation for λH). *If* $M : \Gamma \vdash A$ *and* $M \longrightarrow_H M'$, *then* $M' : \Gamma \vdash A$.

Corollary 147 (Progress for λH). *If* $M : \emptyset \vdash A$, *then*

1. $M \longrightarrow_H M'$ *for some* M',
2. $\mathsf{Value}\, M$, *or*
3. $M \equiv \mathtt{blame}\ \ell$.

7.2.2 Space efficiency

Next, we establish that λH is an instance of the CastHeight structure so that we can apply Theorem 126 (Space Consumption) to obtain space efficiency for λH.

We define the height of a hypercoercion to be the height of its middle part:

$$\begin{aligned}
\mathsf{height}(\mathsf{id}) &= 0 \\
\mathsf{height}(p; m; i) &= \mathsf{height}(m) \\
\mathsf{height}(c \to d) &= 1 + \max(\mathsf{height}(c), \mathsf{height}(d)) \\
\mathsf{height}(c \times d) &= 1 + \max(\mathsf{height}(c), \mathsf{height}(d)) \\
\mathsf{height}(c + d) &= 1 + \max(\mathsf{height}(c), \mathsf{height}(d))
\end{aligned}$$

The size of a hypercoercion is given by the following definition:

$$\begin{aligned}
\mathsf{size}(\mathsf{id}) &= 0 \\
\mathsf{size}(\bot^{\ell}) &= 0
\end{aligned}$$

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

$$\begin{aligned}
\mathsf{size}(G!) &= 1 \\
\mathsf{size}(G?^{\ell}) &= 1 \\
\mathsf{size}(p; m; i) &= 2 + \mathsf{size}(p) + \mathsf{size}(m) + \mathsf{size}(i) \\
\mathsf{size}(c \to d) &= 1 + \mathsf{size}(c) + \mathsf{size}(d) \\
\mathsf{size}(c \times d) &= 1 + \mathsf{size}(c) + \mathsf{size}(d) \\
\mathsf{size}(c + d) &= 1 + \mathsf{size}(c) + \mathsf{size}(d)
\end{aligned}$$

The cast height of the result of applyCast applied to a simple value S and coercion c is less than the max of the cast height of S and the height of c.

Lemma 148. $\mathsf{c\text{-}height}(\mathsf{applyCast}\ S\ c) \leq \max(\mathsf{height}(S), \mathsf{height}(c))$

The dom, cod, fst, snd, inl, and inr operators on coercions all return coercions of equal or lesser height than their input.

Lemma 149.

1. $\mathsf{height}(\mathsf{dom}\ c\ x) \leq \mathsf{height}(c)$
2. $\mathsf{height}(\mathsf{cod}\ c\ x)) \leq \mathsf{height}(c)$
3. $\mathsf{height}(\mathsf{fst}\ c\ x) \leq \mathsf{height}(c)$
4. $\mathsf{height}(\mathsf{snd}\ c\ x) \leq \mathsf{height}(c)$
5. $\mathsf{height}(\mathsf{inl}\ c\ x) \leq \mathsf{height}(c)$
6. $\mathsf{height}(\mathsf{inr}\ c\ x) \leq \mathsf{height}(c)$

The size of a hypercoercion c is bounded by $9 \cdot 2^{\mathsf{height}(c)}$. We prove this by simultaneously proving the following three facts about the four kinds of hypercoercions.

Lemma 150.

1. $\mathsf{size}(c) + 5 \leq 9 \cdot 2^{\mathsf{height}(c)}$
2. $\mathsf{size}(p) \leq 1$
3. $\mathsf{size}(i) \leq 1$
4. $\mathsf{size}(m) + 9 \leq 9 \cdot 2^{\mathsf{height}(m)}$

The above lemmas and definitions establish the following.

Proposition 151. λH *is an instance of the* CastHeight *structure.*

We apply Theorem 139 (Space Consumption) to obtain the following result for λH.

Corollary 152 (Space Consumption for λH). *If* $M : \Gamma \vdash A$*, then there exists* c *such that for any* $M' : \Gamma \vdash A$ *where* $ctx \vdash \mathcal{C}[\![M]\!] \longrightarrow_H^* M'$*, we have* $\mathsf{real\text{-}size}(M') \leq c \cdot \mathit{ideal\text{-}size}(M')$.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press

8 Conclusion

In this paper, we present two parameterized cast calculi, $CC(\Rightarrow)$ and its space-efficient partner $SC(\Rightarrow)$. We prove type safety, blame safety, and the gradual guarantee for the former. We prove type safety and space efficiency for the later. We instantiate $CC(\Rightarrow)$ a half-dozen ways to reproduce some results from the literature but to also fill in many gaps. We instantiate $SC(\Rightarrow)$ two different ways to reproduce λS and to create a new space-efficient calculus based on hypercoercions. All of this is formalized in Agda.

Conflicts of interest

The authors are employed at Indiana University.

References

Ahmed, A., Findler, R. B., Siek, J. G. & Wadler, P. (January 2011) Blame for all. In *Symposium on Principles of Programming Languages*.

Ahmed, A., Jamner, D., Siek, J. G. & Wadler, P. (September 2017) Theorems for free for free: Parametricity, with and without types. In *International Conference on Functional Programming*, ICFP.

Allende, E., Fabry, J. & Tanter, E. (2013) Cast insertion strategies for gradually-typed objects. In *Proceedings of the 9th Symposium on Dynamic Languages*, *DLS '13*, New York, NY, USA: ACM, pp. 27–36.

Aydemir, B. E., Bohannon, A., Fairbairn, M., Foster, J. N., Pierce, B. C., Sewell, P., Vytiniotis, D., Weirich, G. W. S. & Zdancewic, S. (May 2005) *Mechanized Metatheory for the Masses: The POPLmark Challenge*.

Bañados Schwerter, F., Clark, A. M., Jafery, K. A. & Garcia, R. (January 2021) Abstracting gradual typing moving forward: Precise and space-efficient. *Proc. ACM Program. Lang.* **5**(POPL). doi: 10.1145/3434342.

Bierman, G., Abadi, M. & Torgersen, M. (2014) Understanding TypeScript. In *ECOOP 2014 – Object-Oriented Programming*, Jones, R. (ed), vol. 8586. *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, pp. 257–281.

Bove, A., Dybjer, P. & Norell, U. (2009) A brief overview of agda — A functional language with dependent types. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, *TPHOLs '09*, Berlin, Heidelberg: Springer-Verlag, pp. 73–78.

Bracha, G. (2004) Pluggable type systems. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*.

Bracha, G. & Griswold, D. (1993) Strongtalk: typechecking Smalltalk in a production environment. In *OOPSLA '93: Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, New York, NY, USA: ACM Press, pp. 215–230. ISBN 0-89791-587-9.

Castagna, G. & Lanvin, V. (2017) Gradual typing with union and intersection types. In *International Conference on Functional Programming*.

Castagna, G., Lanvin, V., Petrucciani, T. & Siek, J. G. (2019) Gradual typing: A new perspective. *Proc. ACM Program. Lang.* **3**(POPL), 16:1–16:32. ISSN 2475-1421. doi: 10.1145/3290329.

Chaudhuri, A. Flow: A static type checker for Javascript. Available at: http://flowtype.org/

Chung, B., Li, P., Nardelli, F. Z. & Vitek, J. (2018) KafKa: Gradual typing for objects. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, Millstein, T. (ed), vol. 109. *Leibniz International Proceedings in Informatics (LIPIcs),* Dagstuhl, Germany, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 12:1–12:24. ISBN 978-3-95977-079-8. doi: 10.

4230/LIPIcs.ECOOP.2018.12. Available at: http://drops.dagstuhl.de/opus/volltexte/2018/9217.

Eremondi, J., Tanter, E. & Garcia, R. (July 2019) Approximate normalization for gradual dependent types. *Proc. ACM Program. Lang.* **3**(ICFP). doi: 10.1145/3341692.

Felleisen, M. & Friedman, D. P. (1986) Control operators, the SECD-machine and the lambda-calculus, pp. 193–217.

Flanagan, C. (January 2006) Hybrid type checking. In *POPL 2006: The 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Charleston, South Carolina, pp. 245–256.

Garcia, R. (2013) Calculating threesomes, with blame. In *ICFP '13: Proceedings of the International Conference on Functional Programming*.

Garcia, R. & Cimini, M. (2015) Principal type schemes for gradual programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, *POPL '15*. ACM, pp. 303–315.

Garcia, R., Clark, A. M. & Tanter, E. (2016) Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, *POPL 2016*, New York, NY, USA: ACM, pp. 429–442. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837670.

García-Pérez, A., Nogueira, P. & Sergey, I. (2014) Deriving interpretations of the gradually-typed lambda calculus. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, *PEPM '14*, New York, NY, USA: ACM, pp. 157–168. ISBN 978-1-4503-2619-3. doi: 10.1145/2543728.2543742.

Greenman, B. (November 2020) *Deep and Shallow Types*. PhD thesis, Northeastern University.

Greenman, B. & Felleisen, M. (2018) A spectrum of type soundness and performance. *Proc. ACM Program. Lang.* **2**(ICFP), 71:1–71:32. ISSN 2475-1421. doi: 10.1145/3236766.

Greenman, B. & Migeed, Z. (2018) On the cost of type-tag soundness. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, *PEPM '18*, New York, NY, USA: ACM, pp. 30–39. ISBN 978-1-4503-5587-2. doi: 10.1145/3162066.

Gronski, J., Knowles, K., Tomb, A., Freund, S. N. & Flanagan, C. (2006) Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, pp. 93–104.

Henglein, F. (1994). Dynamic typing: Syntax and proof theory. *Sci. Comput. Program.* **22**(3), 197–230.

Herman, D., Tomb, A. & Flanagan, C. (2007) Space-efficient gradual typing. In *Trends in Functional Prog. (TFP)*, p. XXVIII.

Herman, D., Tomb, A. & Flanagan, C. (2010) Space-efficient gradual typing. *Higher-Order Symb. Comput.* **23**(2), 167–189.

Howard, W. A. (1980). *The Formulae-as-Types Notion of Construction*. Academic Press.

Kuhlenschmidt, A., Almahallawi, D. & Siek, J. G. (2019) Toward efficient gradual typing for structural types via coercions. In *Conference on Programming Language Design and Implementation*, PLDI. ACM.

Lennon-Bertrand, M., Maillard, K., Tabareau, N. & Tanter, É. (2020) *Gradualizing the Calculus of Inductive Constructions*.

Lu, K.-C. (April 2020) *Equivalence of Cast Representations in Gradual Typing*. Master's thesis, Indiana University.

Lu, K.-C., Siek, J. G. & Kuhlenschmidt, A. (2020). Hypercoercions and a framework for equivalence of cast calculi. In *Workshop on Gradual Typing*.

Maidl, A. M., Mascarenhas, F. & Ierusalimschy, R. (2014) Typed lua: An optional type system for lua. In *Proceedings of the Workshop on Dynamic Languages and Applications*, *Dyla'14*, New York, NY, USA: ACM, pp. 3:1–3:10.

Matthews, J. & Findler, R. B. (January 2007) Operational semantics for multi-language programs. In *The 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Muehlboeck, F. & Tate, R. (2017). Sound gradual typing is nominally alive and well. *Proc. ACM Program. Lang.* **1**(OOPSLA), 56:1–56:30. ISSN 2475-1421. doi: 10.1145/3133880.

New, M. S., Jamner, D. & Ahmed, A. (2019) Graduality and parametricity: Together again for the first time. *Proc. ACM Program. Lang.* **4**(POPL), doi: 10.1145/3371114.

Nipkow, T., Paulson, L. C. & Wenzel, M. (November 2007) *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, vol. 2283. *LNCS*. Springer.

Siek, J. G. & Garcia, R. (2012) Interpretations of the gradually-typed lambda calculus. In *Scheme and Functional Programming Workshop*.

Siek, J. G. & Taha, W. (2006a) Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pp. 81–92.

Siek, J. G. & Taha, W. (December 2006b). *Gradual Typing for Objects: Isabelle Formaliztaion*. Technical Report CU-CS-1021-06, Boulder, CO: University of Colorado.

Siek, J. G. & Taha, W. (2006c). *Gradual Typing: Isabelle/Isar Formalization*. Technical Report TR06-874, Houston, Texas: Rice University.

Siek, J. G. & Taha, W. (August 2007) Gradual typing for objects. In *European Conference on Object-Oriented Programming*, vol. 4609. *LNCS*, pp. 2–27.

Siek, J. G. & Vachharajani, M. (2008) Gradual typing and unification-based inference. In *DLS*.

Siek, J. G. & Vitousek, M. M. (2013) Monotonic references for gradual typing. *CoRR*, abs/1312.0694.

Siek, J. G. & Wadler, P. (January 2010) Threesomes, with and without blame. In *Symposium on Principles of Programming Languages*, POPL, pp. 365–376.

Siek, J. G., Garcia, R. & Taha, W. (March 2009) Exploring the design space of higher-order casts. In *European Symposium on Programming, ESOP*, pp. 17–31.

Siek, J. G., Thiemann, P. & Wadler, P. (June 2015a) Blame and coercion: Together again for the first time. In *Conference on Programming Language Design and Implementation*, *PLDI*.

Siek, J. G., Vitousek, M. M., Cimini, M. & Boyland, J. T. (May 2015b) Refined criteria for gradual typing. In *SNAPL: Summit on Advances in Programming Languages*, LIPIcs: Leibniz International Proceedings in Informatics.

Siek, J. G., Vitousek, M. M., Cimini, M., Tobin-Hochstadt, S. & Garcia, R. (April 2015c) Monotonic references for efficient gradual typing. In *European Symposium on Programming*, *ESOP*.

Takikawa, A. (April 2016) *The Design, Implementation, and Evaluation of a Gradual Type System for Dynamic Class Composition*. PhD thesis, Northeastern University.

Takikawa, A., Strickland, T. S., Dimoulas, C., Tobin-Hochstadt, S. & Felleisen, M. (2012) Gradual typing for first-class classes. In *Conference on Object Oriented Programming Systems Languages and Applications*, *OOPSLA '12*, pp. 793–810.

Takikawa, A., Feltey, D., Greenman, B., New, M., Vitek, J. & Felleisen, M. (January 2016) Is sound gradual typing dead? In *Principles of Programming Languages*, *POPL*. ACM.

The Coq Dev. Team. (April 2004) *The Coq Proof Assistant Reference Manual – Version V8.0.* Available at: http://coq.inria.fr.

Tobin-Hochstadt, S. & Felleisen, M. (2006) Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium*.

Tobin-Hochstadt, S. & Felleisen, M. (January 2008) The design and implementation of Typed Scheme. In *Symposium on Principles of Programming Languages*.

Toro, M. & Tanter, É. (2020) Abstracting gradual references. *Sci. Comput. Program.* **197**, 102496. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2020.102496. Available at: http://www.sciencedirect.com/science/article/pii/S0167642320301052.

Toro, M., Labrada, E. & Tanter, E. (2019) Gradual parametricity, revisited. *Proc. ACM Program. Lang.* **3**(POPL), 17:1–17:30. ISSN 2475-1421. doi: 10.1145/3290330.

Verlaguet, J. & Menghrajani, A. Hack: A new programming langauge for HHVM. Available at: https://code.facebook.com/posts/264544830379293/hack-a-new-programming-language-for-hhvm/

Vitek, J. (2016) Gradual types for real-world objects. In *Script To Program Evolution Workshop*, *STOP*.

Vitousek, M. & Siek, J. (2016a) From optional to gradual typing via transient checks. In *Script To Program Evolution Workshop*, *STOP*.

Vitousek, M., Swords, C. & Siek, J. G. (2017) Big types in little runtime. In *Symposium on Principles of Programming Languages*, *POPL*.

Vitousek, M. M. & Siek, J. G. (October 206b) *Gradual Typing in An Open World*. Technical Report TR729, Indiana University.

Vitousek, M. M., Siek, J. G. & Chaudhuri, A. (2019) Optimizing and evaluating transient gradual typing. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2019, New York, NY, USA: Association for Computing Machinery, pp. 28–41. ISBN 9781450369961. doi: 10.1145/3359619.3359742.

Wadler, P. & Findler, R. B. (2007) Well-typed programs can't be blamed. In *Workshop on Scheme and Functional Programming*, pp. 15–26.

Wadler, P. & Findler, R. B. (March 2009) Well-typed programs can't be blamed. In *European Symposium on Programming*, *ESOP*, pp. 1–16.

Wadler, P. & Kokke, W. (2019) *Programming Language Foundations in Agda*. Available at http://plfa.inf.ed.ac.uk/

Xie, N., Bi, X. & Oliveira, B. C. d. S. (2018) Consistent subtyping for all. In *Programming Languages and Systems*, Ahmed, A. (ed), pp. 3–30, Cham: Springer International Publishing. ISBN 978-3-319-89884-1.

https://doi.org/10.1017/S0956796821000241 Published online by Cambridge University Press