Deriving Interpretations of the Gradually-Typed Lambda Calculus

Álvaro García-Pérez
IMDEA Software Institute and
Universidad Politécnica de Madrid
agarcia@babel.ls.fi.upm.es

Pablo Nogueira
Universidad Politécnica de Madrid
pablo@babel.ls.fi.upm.es

Ilya Sergey
IMDEA Software Institute
ilya.sergey@imdea.org

Abstract

Siek and Garcia (2012) have explored the dynamic semantics of the gradually-typed lambda calculus by means of definitional interpreters and abstract machines. The correspondence between the calculus's mathematically described small-step reduction semantics and the implemented big-step definitional interpreters was left as a conjecture. We prove and generalise Siek and Garcia's conjectures using program transformation. We establish the correspondence between the definitional interpreters and the reduction semantics of a closure-converted gradually-typed lambda calculus that unifies and amends various versions of the calculus. We use a layered approach and two-level continuation-passing style so that the correspondence is parametric on the subsidiary coercion calculus. We have implemented the whole derivation for the eager error-detection policy and the downcast blame-tracking strategy. The correspondence can be established for other choices of error-detection policies and blame-tracking strategies, by plugging in the appropriate artefacts for the particular subsidiary coercion calculus.

Categories and Subject Descriptors D.3.1 [*Software*]: Programming Languages—Formal Definitions and Theory

General Terms Languages, Theory

Keywords program transformation, gradual types, layered semantics, 2-level continuation-passing style, closures

1. Introduction

Since the publication of [1] a decade ago there has been substantial research on inter-derivation by program transformation of implementations of 'semantic artefacts', *i.e.*, implementations of operational semantics, denotational semantics, and abstract machines. The research has established a semantics framework and has contributed to the repertoire of program transformation techniques. Unfortunately, inter-derivation remains underused. Languages and calculi constantly spring up but their semantics are specified on paper and their correspondences are either obviated, conjectured, or proven mathematically.

PEPM'14, January 22-24 2014, San Diego, CA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2619-3/14/01$15.00.
http://dx.doi.org/10.1145/2543728.2543742

We think inter-derivation is underused for various reasons. First, for readers unfamiliar with the technicalities, proving correspondences by program transformation provides the same assurance as proving them on paper. Formal verification must be brought into the process. This brings us to the second related criticism: the lack of tools. Inter-derivation may become more popular when techniques and folklore are collected, their requirements for program verification formally studied, and a tool developed, preferably integrated within a popular freely-available tool framework. An often suggested possibility is a Coq library for inter-derivation.

Reusability in the form of parametricity and modularity will be an important requirement for this endeavour, in particular, the support for derivation of parametric semantic artefacts. We think two important ingredients in this regard are *hybrid* or *layered* calculi [16, 17], and two-level continuation-passing style [12]. On the one hand hybrid or layered calculi depend on subsidiary sub-calculi, which ought to be turned into a parameter. On the other hand two-level continuation-passing style can be used to separate the hybrid and subsidiary continuation spaces and help parametrise on the subsidiary. In this paper we showcase the marriage of layered semantics and two-level CPS.

Our case study is the recently popular gradually-typed lambda calculus [15, 20–22]. Gradual typing is about giving programmers the freedom to move from dynamic typing to static typing by letting them add type annotations gradually to their programs. The gradually-typed lambda calculus $\lambda^{?}_{\rightarrow}$ is a simply-typed lambda calculus with a dynamic type `Dyn` that is assigned by the type system to expressions whose type is statically unknown. The expressions of $\lambda^{?}_{\rightarrow}$ are translated to the expressions of an intermediate language $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ with explicit casts that carry blame labels. A cast failure delivers a blame label that indicates the location of the failing cast. The dynamic semantics of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ depends on two design decisions (lazy or eager error-detection, downcast or upcast-downcast blame-tracking) which give rise to a design space with four different points: eager-downcast (**ED**), eager-upcast-downcast (**EUD**), lazy-downcast (**LD**), and lazy-upcast-downcast (**LUD**). These points are captured by different coercion sub-calculi.

In [20] we find several implemented denotational semantics (definitional interpreters using meta-level functions) that illustrate the implementation of the variants of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$. The small-step reduction semantics are defined mathematically [20, 22] and the correspondences with the denotational semantics are left as conjectures. We prove and generalise the conjectures using program derivation, parametrising the $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ artefact on the coercion artefact to permit derivations for the whole design space. The inter-derivation diagram of Figure 1 describes the derivation of the semantic artefacts in the paper. Here is our detailed list of **contributions**:

$$\begin{tikzcd}
\text{reduction semantics} \arrow[d, Rightarrow, "\text{syntactic corresp.}"] & & \text{denotational semantics} \arrow[d, Rightarrow, "\text{closure conversion}"] \\
\text{abstract machine} \arrow[r, Rightarrow, "\text{refunc.}"] & \text{2CPS-normaliser} & \text{natural semantics} \arrow[l, Rightarrow, "\text{2CPS transform}"']
\end{tikzcd}$$

Figure 1. Inter-derivation diagram.

- We present a coercion-based version of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ (Section 2) and an eager-downcast coercion calculus $\mathbf{ED}$ (Section 3). These calculi unify and slightly amend and emend the versions in [20, 22] so as to have an *implementable* reduction semantics satisfying unique-decomposition [13].
- In Section 4 we translate the definitional interpreter [20] to ML and derive an instantiation for $\mathbf{ED}$ dynamic semantics (eager-downcast, named $\mathbf{L}\cup\mathbf{D}\cup\mathbf{E}$ in [22]) which is the more appealing for 'it provides thorough error detection and intuitive blame assignment' [22, p.13]. In Section 5 we disentangle translation and normalisation to obtain a purely coercion-based interpreter for expressions that uses a self-contained subsidiary coercion interpreter. In Section 6 we finally produce a 2CPS-normaliser that implements the corresponding big-step natural semantics. These steps belong to the right-hand-side of Figure 1.
- We extend $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ to $\lambda\rho^{\langle\cdot\rangle}_{\rightarrow}$, the simply-typed lambda calculus of closures with explicit casts (Section 8) whose implementable reduction semantics is the starting point of the syntactic correspondence [6, 7, 9] on the left-hand-side of Figure 1. We state the theorems generalising the conjectures in [20] (Section 8.1), and prove them via program transformation by linking the two sides of the diagram at the 2CPS-normaliser (Section 10, etc).
- The small-step and big-step artefacts for expressions with coercion casts are parametric on the artefacts for coercions. We have presented a full derivation for $\mathbf{ED}$ dynamic semantics, but thanks to layering and 2CPS the artefacts for coercions can be replaced by other artefacts implementing different dynamic semantics. This technique provides the basis for modular derivations of any layered semantics, not limited to the definitional interpreters of the gradually-typed lambda calculus.

This paper makes contributions for 'program-derivationists' as well as for 'gradual-type-theorists'. To celebrate the union of the two lines of research, we have summarised in the main sections the important points for each readership, so they can understand the contributions at a glance. In particular, the program-derivationist need not know all details of, and our contributions to, $\lambda^{\langle\cdot\rangle}_{\rightarrow}$. The gradual-type-theorist will find the semantic artefacts in the paper written in traditional mathematical notation. We strongly encourage the program-derivationist to read the derivation parts of the paper alongside the code,[1] which is the main star of the film. The code is written in Standard ML and organised around the sectioning structure of the paper. Thus, 'Section 1.2' refers to a section of the paper whereas 'Code 3.1' refers to a section of the code.

2. $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ with implementable reduction semantics

Figure 2 shows the syntax, contraction rules, and implementable reduction semantics of a coercion-based version of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$. The foremost point for the program-derivationist is the boxed rule StepCst specifying that the contraction of a cast $\langle c\rangle s$ that applies a coercion c to a simple value s depends on the single-step reduction of c to c'

[1] `http://babel.ls.fi.upm.es/~agarcia/papers/Gradual`

Syntax: $\boxed{e \in \lambda^{\langle\cdot\rangle}_{\rightarrow}}$

$$\begin{array}{llll}
\text{base types} & B & = & \{\texttt{Int},\texttt{Bool}\} \\
\text{types} & T & ::= & B \mid \texttt{Dyn} \mid T \rightarrow T \\
\text{constants} & k & ::= & n \in \mathbb{N} \mid \texttt{t} \mid \texttt{f} \\
\text{operators} & op & ::= & \texttt{inc} \mid \texttt{dec} \mid \texttt{zero?} \\
\text{expressions} & e & ::= & k \mid op\ e \mid \texttt{if}\ e\ e\ e \mid x \mid \lambda x : T.e \mid \\
 & & & e\,e \mid \langle S \Leftarrow T\rangle^{\ell} e \mid \langle c\rangle e \mid \texttt{Blame}\ \ell \\
 & & & \\
\text{simple values} & s & ::= & k \mid \lambda x : T.e \\
\text{values} & v & ::= & s \mid \langle \overline{c}\rangle s \\
\text{results} & r & ::= & v \mid \texttt{Blame}\ \ell
\end{array}$$

Contraction: $\boxed{e \longrightarrow e}$

$$\begin{array}{rll}
(\lambda x : T.e)v \longrightarrow [x/v]e & & (\beta) \\
op\ n \longrightarrow \delta(op, n) & & (\delta) \\
\texttt{if}\ k\ e_1\ e_2 \longrightarrow \left\{ \begin{array}{ll} e_1 & \text{if } k = \texttt{t} \\ e_2 & \text{if } k = \texttt{f} \end{array}\right. & & (\textsc{If}) \\
\boxed{\langle c\rangle s \longrightarrow \langle c'\rangle s \quad \text{if } c \longmapsto_{\mathbf{X}} c' \ (\textsc{StepCst})} & & \\
\langle \iota\rangle s \longrightarrow s & & (\textsc{IdCst}) \\
\langle d\rangle\langle \overline{c}\rangle s \longrightarrow \langle \overline{c}\,; d\rangle s & & (\textsc{CmpCst}) \\
\langle \tilde{c} \rightarrow \tilde{d}\rangle s\,v \longrightarrow \langle \tilde{d}\rangle(s\,\langle \tilde{c}\rangle v) & & (\textsc{AppCst}) \\
\langle \texttt{Fail}^{\ell}\rangle s \longrightarrow \texttt{Blame}\ \ell & & (\textsc{FailCst}) \\
\langle (\tilde{c} \rightarrow \tilde{d})\,; \texttt{Fail}^{\ell}\rangle s \longrightarrow \texttt{Blame}\ \ell & & (\textsc{FailFC})
\end{array}$$

Reduction semantics: $\boxed{e \longmapsto e}$

$$\mathsf{E}[\,] \quad ::= \quad [\,] \mid (\mathsf{E}[\,])\,e \mid v\,(\mathsf{E}[\,]) \mid \langle c\rangle(\mathsf{E}[\,])$$

$$\frac{e \longrightarrow e'}{\mathsf{E}[e] \longmapsto \mathsf{E}[e']} \qquad \frac{}{\mathsf{E}[\texttt{Blame}\ \ell] \longmapsto \texttt{Blame}\ \ell}$$

Figure 2. Syntax, contraction rules, and implementable reduction semantics of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$.

in a coercion sub-calculus $\mathbf{X}$. Thus, $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ is a *hybrid* or *layered* calculus whose syntax, contraction rules and, by extension, reduction semantics, depend on a *subsidiary* coercion calculus. Fortunately, the dependency on the syntax is not such: coercion expressions are the same across coercion calculi, and what varies is the syntax of 'normal coercions' (normalised coercion expressions) which naturally depends on the reduction semantics. However, the syntax of normal coercions always includes the ones in the contraction rules of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$. Such rules have to be part of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ for reasons explained below. Thus, the real dependency is on '$\longmapsto_{\mathbf{X}}$', the reduction semantics for coercions. Observe that $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ admits type cast expressions $\langle S \Leftarrow T\rangle^{\ell} e$ which are present in [20, p.2] and in [22, Fig.1]. In particular, the interpreters in [20] work with them. However, type casts are translated off to coercion casts $\langle c\rangle e$. The translation function depends on the reduction semantics for coercions.

A point of interest to the program-derivationist and to the gradual-type-theorist is that in Figure 2 blames are expressions, and reduction lifts blames in any reduction context to a result. Consequently, the figure shows a reduction semantics proper that can be implemented (more details below). Blames are results but not expressions in [20, 22]. The other contents of the section explain whence our version of the calculus which is of interest mainly to the gradual-type-theorist.

The syntax of types, constants, and primitive operators is the same as in [20] and is unmysterious.[2] The syntax of expressions includes the expressions of [22, Fig.7], namely, variables, constants, type-annotated abstractions, expression applications $e\,e$, and coercion casts $\langle c\rangle e$. The unannotated abstractions $\lambda x.e$ of [20] can be represented by $\lambda x\!:\!\texttt{Dyn}.e$. The syntax of expressions also includes applications of primitive operators to expressions, and conditional expressions, both present in [20]. Finally, expressions also include $\texttt{Blame}\ \ell$ expressions because they can be the result of a contraction, and must thus be a particular kind of expression. The type system for $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ can be found in Appendix A.

Now to operational semantics. Results r are now expressions: either values v or blames. Values v are simple values s or a coercion expression that applies a wrapper coercion $\bar{c}$ to a simple value. Wrapper coercions are a subset of normal coercions $\hat{c}$, those that may be applied to simple values. A definition of wrapper and normal coercion for the eager coercion calculus with downcast blame-tracking is given in Section 3. In the definitional interpreters of [20], simple values include meta-level functions because the interpreters implement denotational semantics (Section 4).

The contraction rules are straightforward. The first three specify the contraction of β-, δ-, and conditional redices. (Function δ can be found in Appendix A.) Rule StepCst has already been discussed. The remaining rules, except CmpCst, deal with casts containing normal coercions. Rule CmpCst contracts nested casts (expressions of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$) to coercion sequences. Rule IdCst contracts a cast with an identity coercion. Rule AppCst contracts the application of an arrow coercion $\langle\tilde{c}\to\tilde{d}\rangle$ to a simple value s (an abstraction if well-typed) when the application is in turn applied to an operand v. The cast is performed against the operand and the result of the application. This is the standard solution for higher-order casts (arrow coercions) [22]. Rules FailCst and FailFC contract fail coercions to blames. Rule FailFC is given in $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ to preserve confluence in the coercion calculus (Section 3).[3]

The reduction semantics at the bottom of the figure consists of reduction contexts and single-step contraction rules for redices within context holes. Observe that blames are short-circuited to results by lifting a blame in an arbitrary reduction context to the top level. The reduction contexts specify call-by-value reduction. The reduction of casts would consist of reducing the expression to a simple value, then reducing the coercion within the subsidiary coercion calculus, and the appropriate contraction rule in $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ would take it from there.

3. The ED coercion calculus

Figure 3 shows the syntax, contraction rules, and implementable reduction semantics of **ED**, our version of the eager coercion calculus with downcast blame-tracking, which unifies and slightly amends and emends the versions in [20, 22]. The foremost point for both the program-derivationist and the gradual-type-theorist is that the reduction semantics in the figure satisfies the unique-decomposition property [13] required for implementation, *i.e.*, every coercion expression is uniquely-decomposable into a coercion reduction context with a redex within the whole. The reduction semantics in [22, Fig.4] does not have that property (more details below), and [20] is about big-step definitional interpreters so, naturally, it is unconcerned with reduction semantics. The rest of the discussion about our version of the calculus is of interest mainly to the gradual-type-theorist.

Syntax: $\boxed{c \in \mathbf{ED}}$

$$\begin{array}{llcll}
\text{injectable types} & I & ::= & B \mid T \to T & \\
\text{coercions} & c, d & ::= & \iota \mid I! \mid I?^{\ell} \mid c \to c \mid c\,;c \mid \texttt{Fail}^{\ell} & \\
 & & & & \\
\text{wrappers} & \bar{c} & ::= & \tilde{c} & \text{where } \tilde{c} \neq (\tilde{c} \to \tilde{d}\,;\texttt{Fail}^{\ell}) \\
 & & & & \text{and } \tilde{c} \neq \iota \\
\text{normal parts} & \tilde{c} & ::= & \hat{c} & \text{where } \hat{c} \neq \texttt{Fail}^{\ell} \\
\text{normal coercions} & \hat{c} & ::= & c & \text{if } \nexists c'.c \longmapsto_{\mathbf{ED}} c'
\end{array}$$

Contraction: $\boxed{c \longrightarrow_{\mathbf{ED}} c}$

$$\begin{array}{rr}
I_1!\,; I_2?^{\ell} \longrightarrow_{\mathbf{ED}} \langle\!\langle I_2 \Leftarrow I_1 \rangle\!\rangle^{\ell} & (\textsc{InOut}) \\
(\tilde{c}_1 \to \tilde{c}_2)\,;(\tilde{d}_1 \to \tilde{d}_2) \longrightarrow_{\mathbf{ED}} ((\tilde{d}_1\,;\tilde{c}_1) \to (\tilde{c}_2\,;\tilde{d}_2)) & (\textsc{Arr}) \\
\iota\,;\hat{c} \longrightarrow_{\mathbf{ED}} \hat{c} & (\textsc{IdL}) \\
\hat{c}\,;\iota \longrightarrow_{\mathbf{ED}} \hat{c} & (\textsc{IdR}) \\
\texttt{Fail}^{\ell}\,;\hat{c} \longrightarrow_{\mathbf{ED}} \texttt{Fail}^{\ell} & (\textsc{FailCo}) \\
I!\,;\texttt{Fail}^{\ell} \longrightarrow_{\mathbf{ED}} \texttt{Fail}^{\ell} & (\textsc{InFail}) \\
(\texttt{Fail}^{\ell} \to \hat{d}) \longrightarrow_{\mathbf{ED}} \texttt{Fail}^{\ell} & (\textsc{FailL}) \\
(\tilde{c} \to \texttt{Fail}^{\ell}) \longrightarrow_{\mathbf{ED}} \texttt{Fail}^{\ell} & (\textsc{FailR})
\end{array}$$

Reduction semantics: $\boxed{c \longmapsto_{\mathbf{ED}} c}$

$$\begin{array}{lcll}
\mathsf{C}_c[\,] & ::= & [\,] & \text{if } c \text{ is a redex} \\
 & \mid & \mathsf{C}_{c_1}[\,]\,;c_2 & \text{if } c \equiv c_1\,;c_2 \\
 & \mid & \hat{c}_1\,;\mathsf{C}_{c_2}[\,] & \text{if } c \equiv \hat{c}_1\,;c_2 \\
 & \mid & \mathsf{C}_{c_1}[\,] \to c_2 & \text{if } c \equiv c_1 \to c_2 \\
 & \mid & \hat{c}_1 \to \mathsf{C}_{c_2}[\,] & \text{if } c \equiv \hat{c}_1 \to c_2 \\
 & \mid & \hat{c}_{11}\,;\mathsf{C}_{(\hat{c}_{12}\,;\hat{c}_2)}[\,] & \text{if } c \equiv (\hat{c}_{11}\,;\hat{c}_{12})\,;\hat{c}_2 \\
 & \mid & \mathsf{C}_{(\hat{c}_1\,;\hat{c}_{12})}[\,]\,;\hat{c}_{22} & \text{if } c \equiv \hat{c}_1\,;(\hat{c}_{21}\,;\hat{c}_{22})
\end{array}$$

$$\frac{c \longrightarrow_{\mathbf{ED}} c'}{c_1 \equiv \mathsf{C}_{c_1}[c] \longmapsto_{\mathbf{ED}} \mathsf{C}_{c_2}[c'] \equiv c_2}$$

Figure 3. Syntax, contraction rules, and reduction semantics of **ED**.

First, we discuss coercion expressions. Injectable types to `Dyn` are those types other than `Dyn` because injecting or projecting `Dyn` to `Dyn` is equivalent to the identity coercion. Coercion expressions consist of the identity coercion ι, an injection $I!$ of an injectable type I to `Dyn`, a projection from the dynamic type $I?^{\ell}$ to injectable type I decorated with a blame label ℓ, arrow coercions $c \to d$, sequences $c\,;d$ (diagrammatic composition) and fail coercions $\texttt{Fail}^{\ell}$ decorated with a blame label. So far, no differences with [20, 22] other than notational. The type system of **ED** can be found in Appendix B.

The differences arise in the reduction semantics. The reduction semantics at the bottom of Figure 3 satisfies unique-decomposition and is therefore implementable. The one in [22, Fig.4] is defined modulo a congruence on sequences $(c_1\,;c_2)\,;c_3 \cong c_1\,;(c_2\,;c_3)$ that permits the definition of simpler reduction contexts:

$$\mathsf{C}[\,] \quad ::= \quad [\,] \mid \mathsf{C}[\,]\,;c \mid \hat{c}\,;\mathsf{C}[\,] \mid \mathsf{C}[\,] \to c \mid \tilde{c} \to \mathsf{C}[\,]$$

$$\frac{c \cong \mathsf{C}[c_1] \quad c_1 \longrightarrow_{\mathbf{ED}} c_2 \quad \mathsf{C}[c_2] \cong c'}{c \longmapsto_{\mathbf{ED}} c'}$$

To resolve the ambiguity introduced by congruence we fix the association by defining reduction contexts $\mathsf{C}_c[\,]$ which are indexed by the input coercion c, so that decomposition is guided by the shape of c. (Note the difference between syntactic identity '$\equiv$' in

[2] For the thorough reader: in [20] some expressions carry labels, but in the coercion-based versions only coercions and blames carry labels. Our version is coercion-based and therefore only coercions and blames carry labels, together with type cast expressions which will be later removed.

[3] For the thorough reader: in [20] rule FailFC is implemented in different places for different semantics. In the lazy artefacts the rule is implemented by `seq_lazy`, and in the eager artefacts by `mk_cast_eager`.

Figure 3 and congruence '$\cong$' in the original reduction semantics.) Unique-decomposition is proven by structural induction on c.

The difficulties in implementing a deterministic semantics for coercions have already been acknowledged by Garcia [15]. He introduces a composition-free representation for coercions, named *supercoercions*, and a new set of contraction rules. His approach solves the challenge of being 'complete in the face of inert compositions and associativity' [15], superseding the 'ad hoc reassociation scheme' in [20]. Here we stick to the ad hoc scheme, since we aim to prove the conjectures relative to [20].

The rest of this section is addressed to the gradual-type-theorist. Recall from Section 2 that wrapper coercions $\bar{c}$ are normal coercions $\hat{c}$ that may be applied to simple values. Normal coercions are those that cannot be further reduced, and normal parts are coercions other than the fail coercion. The notion of wrapper is induced by the treatment of ι, $\texttt{Fail}^{\ell}$, and $\tilde{c} \to \tilde{d}; \texttt{Fail}^{\ell}$. Wrappers can only coerce constants and abstractions. We add the required side-condition $\tilde{c} \neq (\tilde{c} \to \tilde{d}; \texttt{Fail}^{\ell})$ to wrappers which is missing in [22, Fig.7]. The side-condition is required because the contraction rule for $\langle(\tilde{c} \to \tilde{d}; \texttt{Fail}^{\ell})\rangle s$ in Figure 2 delivers a blame. In [20] an extensional definition of normal coercions is provided that rules out the ill-typed ones according to the type system of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ and $\mathbf{ED}$.

The contraction rule InOut coalesces an injection followed by a projection using a translation function $\langle\!\langle I_2 \Leftarrow I_1 \rangle\!\rangle^{\ell}$ (Appendix B). This function translates a type cast to a normal coercion. If the projection is illegal the translation function delivers a fail coercion decorated with the projection's blame label. The contraction rules Arr to FailR are those in [20, Sec.6.1] but with normal coercions $\hat{c}$ substituted for arbitrary coercions c in order to have a reduction semantics with unique-decomposition and preserve confluence: arbitrarily long sequences ending in $\texttt{Fail}^{\ell}$ must be allowed to fail due to previous coercions early in the sequence. (This is also the reason why FailFC is specified in $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ and not in $\mathbf{ED}$, see [22] for details.)

Like [20] but unlike [22] we omit rule $\iota \to \iota \longrightarrow \iota$ because it is superfluous: according to Arr, IdL, and IdR, sequencing the $\iota \to \iota$ arrow to any other arrow has the same effect as sequencing the identity ι.

4. Interpretations of the gradually-typed lambda calculus

In [20] a definitional interpreter `interp` for a type-cast-based $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ is given that is parametric on functions `cast` and `apply`. The choice and name of parameters do much more than reflect the dependency on the coercion sub-calculus $\mathbf{X}$. Among other things they also accommodate the translation to coercion casts, and apportion the implementation of contraction rules. Broadly, the `cast` parameter is instantiated to `apply_cast_`$\mathbf{X}$ which given a type cast and a value, it first invokes the translation function `mk_coerce_`$\mathbf{X}$ to the type cast to obtain a *normal* coercion, and then invokes `apply_coercion_`$\mathbf{X}$ that applies that normal coercion to the value. The `apply` parameter is instantiated to `apply_`$\mathbf{X}$ which, broadly, realises rule AppCst in Figure 2. The `cast` parameter realises the other rules in the figure dealing with coercions.

The definitional interpreter is environment-based and implements a denotational semantics. It delivers meta-level-function results, nicknamed 'procedures' in [20]. Figure 4 defines in mathematical notation the environments and the hierarchy of results used by the definitional interpreter. An environment is a colon-separated list of bindings $x \mapsto v$, where x is a variable and v is a value. Values are either simple values or coercion expressions applying a wrapper over a simple value. A simple value is either a constant or a procedure `fn y =>` $r_{\mathtt{y}}$ (with *fresh* formal parameter `y`), that takes

Syntax:

environments	ρ	$::=$	$\epsilon \mid (x \mapsto v) : \rho$
procedures	$\mathit{proc} \in \mathrm{V} \to \mathrm{R}$	$::=$	`fn y =>` $r_{\mathtt{y}}$
simple values	s	$::=$	$k \mid \mathit{proc}$
values	$v \in \mathrm{V}$	$::=$	$s \mid \langle\bar{c}\rangle s$
results	$r \in \mathrm{R}$	$::=$	$v \mid \texttt{Blame}\ \ell$

Figure 4. Environments and values for the interpreter in [20].

a value and returns a result that depends on this value. Finally, a result is a value or a blame label.

4.1 Translating the original interpreter to ML

We have translated to Standard ML the original definitional interpreter in [20] which is written in Scheme. We are more comfortable with ML, which is used in many papers in program derivation. The ML translation can be found in Code 1.1. In the code, we use a nameless representation with de Bruijn indices, but we keep the traditional nameful representation in the mathematical notation. For readability, and to avoid the (un)packing of data constructors, we have embedded the whole hierarchy of normal coercions in one datatype `coercion`. For the hierarchy of values and results we use the mutually dependent datatypes `value` and `result`, with procedures represented by clause `VPROC of value -> result`.

In the ML translation, we have replaced the monadic macro for let-expressions `letB` in [20] by case expressions that short-circuit the blames to results.

4.2 Instantiating the definitional interpreter

We have to 'instantiate' the parametric definitional interpreter to a particular dynamic semantics in order to establish the correspondence with an implementation of a reduction semantics that allegedly realises exactly that dynamic semantics. We choose the $\mathbf{ED}$ semantics (Section 1). To obtain an instantiation we inline the function calls and produce functions `mk_arrow`, `translate_cast`, `compose_coercion`, `mk_cast`, `apply_coercion`, `apply_cast`, `apply`, and `eval` (our name for `interp_`$\mathbf{ED}$) all found in Code 1.2. Function `compose_coercion` corresponds to `seq_eager` in [20]. Function `translate_cast` is the instantiation of `mk_coerce_d` with parameter `mk_arrow_eager`. Function `translate_cast` implements $\langle\!\langle S \Leftarrow T \rangle\!\rangle^{\ell}$ defined in Appendix B.

Figure 5 shows in mathematical notation the denotational semantics implemented by Code 1.2. We have written $e[\rho] =_{\langle\cdot\rangle} r$ for the evaluation of expression e in an environment ρ with result r. The notation $e[\rho]$ stands for an unfolded representation of a closure. This denotational semantics is the one on the top right corner of the inter-derivation diagram (Figure 1). The mathematical semantics can be checked by both the gradual-type-theorist and the program-derivationist against the Scheme or ML versions of the instantiated definitional interpreter. Function `eval` in Code 1.2 is specified by the evaluation section of the figure. Function `apply` by the application section, `apply_cast` by the cast application section, `apply_coercion` by the coercion application section, and `compose_coercion` by the coercion composition section.

The evaluation rule EvProc specifies the evaluation of an abstraction in an environment. The evaluation results in a procedure with fresh meta-variable `y`. The value v stands for the one passed to the procedure in rule AppProc.

4.3 The correctness conjectures

In [20] several correspondences between some instantiations of the definitional interpreter and the relevant reduction semantics are conjectured, which we quote:

Evaluation: $\boxed{e[\rho] =_{\langle\cdot\rangle} r}$

$$\frac{}{k[\rho] =_{\langle\cdot\rangle} k}\ (\textsc{EvConst}) \qquad \frac{e[\rho] =_{\langle\cdot\rangle} k}{(op\ e)[\rho] =_{\langle\cdot\rangle} \delta(op,k)}\ (\textsc{EvOper}) \qquad \frac{e[\rho] =_{\langle\cdot\rangle} \texttt{Blame}\ \ell}{(op\ e)[\rho] =_{\langle\cdot\rangle} \texttt{Blame}\ \ell}\ (\textsc{EvOperB})$$

$$\frac{e_1[\rho] =_{\langle\cdot\rangle} \texttt{t} \qquad e_2[\rho] =_{\langle\cdot\rangle} r}{(\texttt{if}\ e_1\ e_2\ e_3)[\rho] =_{\langle\cdot\rangle} r}\ (\textsc{EvIfL}) \qquad \frac{e_1[\rho] =_{\langle\cdot\rangle} \texttt{f} \qquad e_3[\rho] =_{\langle\cdot\rangle} r}{(\texttt{if}\ e_1\ e_2\ e_3)[\rho] =_{\langle\cdot\rangle} r}\ (\textsc{EvIfR}) \qquad \frac{e_1[\rho] =_{\langle\cdot\rangle} \texttt{Blame}\ \ell}{(\texttt{if}\ e_1\ e_2\ e_3)[\rho] =_{\langle\cdot\rangle} \texttt{Blame}\ \ell}\ (\textsc{EvIfB})$$

$$\frac{\rho\,!\,x =_{\langle\cdot\rangle} r}{x[\rho] =_{\langle\cdot\rangle} r}\ (\textsc{EvVar}) \qquad \frac{(\mathtt{y}\ \text{fresh}) \qquad [v/\mathtt{y}](e[(x \mapsto \mathtt{y}) : \rho]) =_{\langle\cdot\rangle} r}{(\lambda x : T.e)[\rho] =_{\langle\cdot\rangle} (\texttt{fn y => } r_{\mathtt{y}})}\ (\textsc{EvProc}) \qquad \frac{e_1[\rho] =_{\langle\cdot\rangle} \texttt{Blame}\ \ell}{(e_1\ e_2)[\rho] =_{\langle\cdot\rangle} \texttt{Blame}\ \ell}\ (\textsc{EvAppBL})$$

$$\frac{e_1[\rho] =_{\langle\cdot\rangle} v_1 \qquad e_2[\rho] =_{\langle\cdot\rangle} \texttt{Blame}\ \ell}{(e_1\ e_2)[\rho] =_{\langle\cdot\rangle} \texttt{Blame}\ \ell}\ (\textsc{EvAppBR}) \qquad \frac{e_1[\rho] =_{\langle\cdot\rangle} v_1 \qquad e_2[\rho] =_{\langle\cdot\rangle} v_2 \qquad v_1\ v_2 \Downarrow^{co}_{ap} v_3}{(e_1\ e_2)[\rho] =_{\langle\cdot\rangle} v_3}\ (\textsc{EvApp})$$

$$\frac{e[\rho] =_{\langle\cdot\rangle} v \qquad \langle S \Leftarrow T\rangle^{\ell} v \Downarrow^{co}_{cs} r}{(\langle S \Leftarrow T\rangle^{\ell} e)[\rho] =_{\langle\cdot\rangle} r}\ (\textsc{EvCast}) \qquad \frac{e[\rho] =_{\langle\cdot\rangle} \texttt{Blame}\ \ell}{(\langle S \Leftarrow T\rangle^{\ell} e)[\rho] =_{\langle\cdot\rangle} \texttt{Blame}\ \ell}\ (\textsc{EvCastB}) \qquad \frac{}{(\texttt{Blame}\ \ell)[\rho] =_{\langle\cdot\rangle} \texttt{Blame}\ \ell}\ (\textsc{EvBlame})$$

Application: $\boxed{v\ v \Downarrow^{co}_{ap} r}$

$$\frac{}{proc\ v \Downarrow^{co}_{ap} proc(v)}\ (\textsc{ApProc}) \qquad \frac{\langle\tilde{c}\rangle v \Downarrow^{co}_{cr} v_1 \qquad proc\ v_1 \Downarrow^{co}_{ap} v_2 \qquad \langle\tilde{d}\rangle v_2 \Downarrow^{co}_{cr} r}{(\langle\tilde{c} \to \tilde{d}\rangle proc) v \Downarrow^{co}_{ap} r}\ (\textsc{ApArr})$$

$$\frac{\langle\tilde{c}\rangle v \Downarrow^{co}_{cr} \texttt{Blame}\ \ell}{(\langle\tilde{c} \to \tilde{d}\rangle proc) v \Downarrow^{co}_{ap} \texttt{Blame}\ \ell}\ (\textsc{ApArrBL}) \qquad \frac{\langle\tilde{c}\rangle v \Downarrow^{co}_{cr} v_1 \qquad proc\ v_1 \Downarrow^{co}_{ap} \texttt{Blame}\ \ell}{(\langle\tilde{c} \to \tilde{d}\rangle proc) v \Downarrow^{co}_{ap} \texttt{Blame}\ \ell}\ (\textsc{ApArrBR})$$

Cast application: $\boxed{\langle S \Leftarrow T\rangle^{\ell} v \Downarrow^{co}_{cs} r}$ **Coercion application:** $\boxed{\langle\hat{c}\rangle v \Downarrow^{co}_{cr} r}$

$$\frac{\langle\ \langle\!\langle S \Leftarrow T\rangle\!\rangle^{\ell}\ \rangle v \Downarrow^{co}_{cr} r}{\langle S \Leftarrow T\rangle^{\ell} v \Downarrow^{co}_{cs} r}\ (\textsc{Cst}) \qquad \frac{\bar{c}\,;\hat{d} \Downarrow^{co}_{\mathbf{ED}} \hat{c}_1}{\langle\hat{d}\rangle(\langle\bar{c}\rangle s) \Downarrow^{co}_{cr} mkCast(\hat{c}_1, s)}\ (\textsc{CoeComp}) \qquad \frac{}{\langle\hat{c}\rangle s \Downarrow^{co}_{cr} mkCast(\hat{c}, s)}\ (\textsc{CoeNorm})$$

Coercion composition: $\boxed{\hat{c}\,;\hat{c} \Downarrow^{co}_{\mathbf{ED}} \hat{c}}$

$$\frac{}{I!\,; J?^{\ell} \Downarrow^{co}_{\mathbf{ED}} \langle\!\langle J \Leftarrow I\rangle\!\rangle^{\ell}}\ (\textsc{ComInOut}) \qquad \frac{\hat{c}_1\,;\hat{c}_2 \in \mathrm{NC}}{\hat{c}_1\,;\hat{c}_2 \Downarrow^{co}_{\mathbf{ED}} \hat{c}}\ (\textsc{ComNorm}) \qquad \frac{\tilde{c}_{21}\,;\tilde{c}_{11} \Downarrow^{co}_{\mathbf{ED}} \hat{c}_3 \qquad \tilde{c}_{22}\,;\tilde{c}_{12} \Downarrow^{co}_{\mathbf{ED}} \hat{c}_4}{(\tilde{c}_{11} \to \tilde{c}_{12})\,;(\tilde{c}_{21} \to \tilde{c}_{22}) \Downarrow^{co}_{\mathbf{ED}} mkArr(\hat{c}_3, \hat{c}_4)}\ (\textsc{ComArr})$$

$$\frac{\hat{c}_{12}\,;\hat{c}_2 \Downarrow^{co}_{\mathbf{ED}} \hat{c}_3 \qquad \hat{c}_{11}\,;\hat{c}_3 \Downarrow^{co}_{\mathbf{ED}} \hat{c}_4}{(\hat{c}_{11}\,;\hat{c}_{12})\,;\hat{c}_2 \Downarrow^{co}_{\mathbf{ED}} \hat{c}_4}\ (\textsc{ComAssL}) \qquad \frac{\hat{c}_1\,;\hat{c}_{21} \Downarrow^{co}_{\mathbf{ED}} \hat{c}_3 \qquad \hat{c}_3\,;\hat{c}_{22} \Downarrow^{co}_{\mathbf{ED}} \hat{c}_4}{\hat{c}_1\,;(\hat{c}_{21}\,;\hat{c}_{22}) \Downarrow^{co}_{\mathbf{ED}} \hat{c}_4}\ (\textsc{ComAssR})$$

$$\frac{}{\iota\,;\hat{c} \Downarrow^{co}_{\mathbf{ED}} \hat{c}}\ (\textsc{ComIdL}) \qquad \frac{}{\hat{c}\,;\iota \Downarrow^{co}_{\mathbf{ED}} \hat{c}}\ (\textsc{ComIdR}) \qquad \frac{}{\texttt{Fail}^{\ell}\,;\hat{c} \Downarrow^{co}_{\mathbf{ED}} \texttt{Fail}^{\ell}}\ (\textsc{ComFailL}) \qquad \frac{}{I!\,;\texttt{Fail}^{\ell} \Downarrow^{co}_{\mathbf{ED}} \texttt{Fail}^{\ell}}\ (\textsc{ComFailR})$$

Figure 5. Denotational semantics in mathematical notation implemented by equivalent definitional interpreters `interp_ed` ([20] and Code 1.1) and `eval` (Code 1.2). Auxiliary functions $\rho\,!\,x$, $\langle\!\langle S \Leftarrow T\rangle\!\rangle^{\ell}$, *mkCast*, and *mkArr* are defined in Appendix C.

Conjecture 1. *If the unique cast labelled with ℓ in program e respects subtyping, then* `eval_ld(`e`)` $\neq$ `Blame` ℓ.

Conjecture 2. *For any well-typed program e,* `eval_ld(`e`)` $=$ *o if and only if* $\langle\!\langle e\rangle\!\rangle \longmapsto^{*}_{\mathbf{LD}} r$ *and* `observe(`r`)` $= o$.

Conjecture 9.1. *Given two well-typed coercions in normal form, c_1 and c_2, we have* `seq_ed(`c_1`,`c_2`)` $= \hat{c}_3$ *and* $(c_1\,; c_2) \longmapsto^{*}_{\mathbf{ED}} \hat{c}_3$.

Conjectures 1 and 2 state the correctness of the instantiated interpreter relative to the **LD** semantics. Conjecture 1 states subtyping soundness (by the way, `eval_ld` type-checks expressions and invokes `interp_ld`). Conjecture 2 states the correspondence between the instantiated definitional interpreter and the **LD** reduction semantics. Function `observe` is a reflect function as in normalisation by evaluation: it produces denotational (meta-level) results from results produced by reduction. Expressions with type casts are translated by $\langle\!\langle\cdot\rangle\!\rangle$ to expressions with coercion casts. There are analogous Conjectures 3 and 4 in [20] for the **LUD** semantics. However, no conjectures are stated for the **ED** and the **EUD** semantics. Conjecture 9.1 states the correctness relative to the **ED** semantics of the composition of two normal coercions.

As discussed in the introduction, we generalise and prove these conjectures for the **ED** semantics (Section 5.2 and 8) by interderiving the reduction semantics and the instantiated definitional interpreter. The other conjectures can be proven similarly by plugging a different reduction semantics for coercions (Section 11.2).

5. Prelude: from casts to coercions

The denotational semantics in Figure 5 is defined for a type-cast-based $\lambda^{\langle\cdot\rangle}_{\to}$. In this section we discuss how we have turned it (actually, its implementation `eval`) into a purely coercion-based semantics by applying two program transformation steps. The bulk of this section will be of interest to the program-derivationist who is advised to read Code 2 alongside this section.

5.1 Fissioning evaluator and translation function

Function `apply_cast` (Section 4.2, Code 1.2) first translates type casts to coercion casts and then invokes `apply_coercion`. We want to get rid of the translation and obtain a coercion-based normaliser. We inline `apply_cast` in `eval` to get rid of `apply_cast`, and then perform lightweight *fission* by fix point promotion to separate `translate_expression` from `eval1`, the obtained translation-free evaluation function. The fission transformation (a.k.a. trampoline transformation) is the inverse of the *fusion* transformation described in [19]. Function `apply_cast` is no longer used. The resulting coercion-based interpreter `eval1` and the `translate_expression` are found in Code 2.1. The latter implements $\langle\!\langle\cdot\rangle\!\rangle$ in Conjecture 2. It fires on type casts, is an identity on variables, constants, and blame expressions, and recursively proceeds over other expressions. Hereafter we can forget about type casts.

5.2 Deriving a self-contained coercion normaliser

As a result of the previous inlining, the coercion-based interpreter for expressions `eval1` invokes `compose_coercion` ($\Downarrow^{co}_{\mathbf{ED}}$ in Figure 5) to normalise sequences of normal coercions. In order to prove the correspondence with the reduction semantics we need a self-contained coercion normaliser. We have produced such normaliser `normalise_coercion_nor` in Code 2.2.3, which implements the natural semantics $\Downarrow_{\mathbf{ED}}$ shown in Figure 6. To obtain this normaliser we first write a coercion normaliser (Code 2.2) that normalises sequences and arrows left-to-right, invoking respectively `compose_coercion` or `mk_arrow` afterwards, and is an identity on the other normal coercions. We replace the calls to `compose_coercion` by recursive calls to the normaliser on a sequence (Code 2.2.1), inline sequencing within normalisation (Code 2.2.2) and defer the normalisation of sequences of arrows by constructing intermediate arrows with sequences (Code 2.2.3). All these steps are equivalence-preserving, and `normalise_coercion_nor` behaves like `compose_coercion` for sequences of normal coercions. Function `eval_nor` in Code 2.2.3 is the instantiated definitional interpreter that employs the self-contained `normalise_coercion_nor`.

6. From denotational semantics to 2CPS-normaliser

In this section, we apply closure conversion [8] to defunctionalise the meta-level procedures of the definitional interpreter. We obtain a natural semantics (a big-step normaliser) that is the starting point of the functional correspondence [1, 7, 11] (the derivation of an abstract machine from a natural semantics).

6.1 Closure conversion

Closure-conversion consists of defunctionalising the procedures in `eval_nor` (Code 2.2.3) by enumerating the inhabitants of the function space and by introducing a datatype constructor (defunctionalised continuation) for each of the inhabitants. An auxiliary function will apply such constructors to the intermediate results of computation. There is only one inhabitant, namely, the function packed within the `VPROC` constructor, which takes an operand and invokes `eval_nor` on the procedure body, passing an environment enlarged with the operand. We defunctionalise and introduce constructor `VPROC1` in datatype `value_clos` (Code 3.1). The constructor stores the body and the environment of the lambda expression representing the procedure, which together make up a *closure* [18]. The auxiliary function that applies the defunctionalised continuation is inlined in function `apply_clos`. The resulting natural semantics is shown in Figure 7 and corresponds to the bottom right corner of the inter-derivation diagram (Figure 1).

In mathematical notation `VPROC1` will be represented by symbol $\lambda\!\!\lambda$ to suggest the relationship with the meta-level. The symbol will also help us discriminate between the result $\lambda\!\!\lambda.e[\rho']$ of evaluating an abstraction closure $(\lambda x : T.e)[\rho]$, and an input closure $e[\rho']$. The closure-converted result hierarchy is morally the same as the original result hierarchy (Figure 4) except that procedures are represented at the object-level by $\lambda\!\!\lambda.e[\rho']$.

In rule NSFUN, the type-annotated formal parameter $(x : T)$ is stored directly in the environment instead than attached to $\lambda\!\!\lambda$ for lexical scoping reasons explained in Section 8. The definition of environment and look-up is duly adapted (Appendix C).

Datatypes `item_clos` and `environment_clos` in Code 3.1 implement the closure-converted environments in Appendix C. Function `mk_cast_clos` is the closure-converted cast combinator, with values and results in the closure-converted results hierarchy. Functions `apply_coercion_clos`, `apply_clos`, and `eval_clos` in Code 3.1 implement the natural semantics in Figure 7.

Recall from Section 4.2 that the code uses an unfolded representation of closures. A datatype for closures will be introduced in Section 10 for the reduction semantics.

6.2 2-Level continuation-passing-style transformation

The layered nature of the semantic artefacts require specific CPS transformation techniques to keep coercion and expression semantics apart. We use 2-level continuation-passing style (2CPS) [12] to introduce two function spaces for the rest of the computation: an inner space of continuations for coercion normalisation, and an outer space of meta-continuations for expression normalisation. We 2CPS-transform `eval_clos` by naming intermediate results of computation, respectively for coercion normalisation and for expression normalisation, and by turning all the calls into tail calls. Functions `normalise_coercion_cps`, `mk_cast_cps`, `apply_coercion_cps`, `apply_cps`, and `eval_cps` in Code 3.2 implement the 2CPS-normaliser (a refunctionalised abstract machine) in the bottom middle of Figure 1.

After 2CPS transformation, the functional correspondence would have continued by defunctionalising the 2CPS-normaliser. However, we halt the functional correspondence at this point and move on to the reduction semantics (top left corner of Figure 1). In Section 8, we introduce the calculus of closures $\lambda\rho^{\langle\cdot\rangle}_{\rightarrow}$, which allows to define the closure-converted small-step reduction semantics.

7. Tackling the other side of the diagram

Section 4 to Section 6 have dealt with the right-hand side of the inter-derivation diagram (Figure 1). We now move to the other side. The 2CPS-normaliser is an artefact with closures, but the reduction semantics given in Sections 2 and 3 are for plain expressions. In Section 8 we extend $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ to $\lambda\rho^{\langle\cdot\rangle}_{\rightarrow}$, a simply-typed lambda calculus of closures with explicit casts, whose reduction semantics is the starting point of the syntactic correspondence that will arrive at the 2CPS-normaliser.

8. The calculus of closures

Figure 8 shows the syntax, contraction rules, and implementable reduction semantics of $\lambda\rho^{\langle\cdot\rangle}_{\rightarrow}$. This section is also of interest to the gradual-type-theorist. Observe that boxed rule $\mathrm{STEPCST}_{\rho}$ is present.

Closures cl consist of *proper closures* $e[\rho]$ and some additional *ephemeral closures*, in the spirit of [2], that lift expression scopes

Coercion normalisation: $\boxed{c \Downarrow_{\mathbf{ED}} \hat{c}}$

$$\frac{c_1 \Downarrow_{\mathbf{ED}} I! \qquad c_2 \Downarrow_{\mathbf{ED}} J?^{\ell}}{c_1\,;c_2 \Downarrow_{\mathbf{ED}} \langle\!\langle J \Leftarrow I\rangle\!\rangle^{\ell}}\ (\textsc{CoeInOut}) \qquad \frac{c_1 \Downarrow_{\mathbf{ED}} \hat{c}_1 \qquad c_2 \Downarrow_{\mathbf{ED}} \iota}{c_1\,;c_2 \Downarrow_{\mathbf{ED}} \hat{c}_1}\ (\textsc{CoeIdL}) \qquad \frac{c_1 \Downarrow_{\mathbf{ED}} \iota \qquad c_2 \Downarrow_{\mathbf{ED}} \hat{c}_2}{c_1\,;c_2 \Downarrow_{\mathbf{ED}} \hat{c}_2}\ (\textsc{CoeIdR})$$

$$\frac{c_1 \Downarrow_{\mathbf{ED}} (\tilde{c}_{11} \to \tilde{c}_{12}) \qquad c_2 \Downarrow_{\mathbf{ED}} (\tilde{c}_{21} \to \tilde{c}_{22}) \qquad ((\tilde{c}_{21}\,;\tilde{c}_{11}) \to (\tilde{c}_{12}\,;\tilde{c}_{22})) \Downarrow_{\mathbf{ED}} \hat{c}_3}{c_1\,;c_2 \Downarrow_{\mathbf{ED}} \hat{c}_3}\ (\textsc{CoeSeqArr})$$

$$\frac{c_1 \Downarrow_{\mathbf{ED}} (\hat{c}_{11}\,;\hat{c}_{12}) \qquad c_2 \Downarrow_{\mathbf{ED}} \hat{c}_2 \qquad \hat{c}_{12}\,;\hat{c}_2 \Downarrow_{\mathbf{ED}} \hat{c}_3 \qquad \hat{c}_{11}\,;\hat{c}_3 \Downarrow_{\mathbf{ED}} \hat{c}_4}{c_1\,;c_2 \Downarrow_{\mathbf{ED}} \hat{c}_4}\ (\textsc{CoeAssL}) \qquad \frac{c \Downarrow_{\mathbf{ED}} \hat{c} \qquad d \Downarrow_{\mathbf{ED}} \hat{d}}{(c \to d) \Downarrow_{\mathbf{ED}} \mathit{mkArr}(\hat{c}, \hat{d})}\ (\textsc{CoeArr})$$

$$\frac{c_1 \Downarrow_{\mathbf{ED}} \hat{c}_1 \qquad c_2 \Downarrow_{\mathbf{ED}} (\hat{c}_{21}\,;\hat{c}_{22}) \qquad \hat{c}_1\,;\hat{c}_{21} \Downarrow_{\mathbf{ED}} \hat{c}_3 \qquad \hat{c}_3\,;\hat{c}_{22} \Downarrow_{\mathbf{ED}} \hat{c}_4}{c_1\,;c_2 \Downarrow_{\mathbf{ED}} \hat{c}_4}\ (\textsc{CoeAssR}) \qquad \frac{}{\hat{c} \Downarrow_{\mathbf{ED}} \hat{c}}\ (\textsc{CoeTriv})$$

$$\frac{c_1 \Downarrow_{\mathbf{ED}} \mathtt{Fail}^{\ell} \qquad c_2 \Downarrow_{\mathbf{ED}} \hat{c}_2}{c_1\,;c_2 \Downarrow_{\mathbf{ED}} \mathtt{Fail}^{\ell}}\ (\textsc{CoeFailL}) \qquad \frac{c_1 \Downarrow_{\mathbf{ED}} I! \qquad c_2 \Downarrow_{\mathbf{ED}} \mathtt{Fail}^{\ell}}{c_1\,;c_2 \Downarrow_{\mathbf{ED}} \mathtt{Fail}^{\ell}}\ (\textsc{CoeFailR})$$

Figure 6. Natural semantics for coercion normalisation.

Evaluation: $\boxed{e[\rho] \Downarrow r}$

$$\frac{}{k[\rho] \Downarrow k}\ (\textsc{NSConst}) \qquad \frac{e[\rho] \Downarrow n}{(op\ e)[\rho] \Downarrow \delta(op, n)}\ (\textsc{NSOper}) \qquad \frac{e[\rho] \Downarrow \mathtt{Blame}\ \ell}{(op\ e)[\rho] \Downarrow \mathtt{Blame}\ \ell}\ (\textsc{NSOperB})$$

$$\frac{e_1[\rho] \Downarrow \mathtt{t} \qquad e_2[\rho] \Downarrow r}{(\mathtt{if}\ e_1\ e_2\ e_3)[\rho] \Downarrow r}\ (\textsc{NSIfL}) \qquad \frac{e_1[\rho] \Downarrow \mathtt{f} \qquad e_3[\rho] \Downarrow r}{(\mathtt{if}\ e_1\ e_2\ e_3)[\rho] \Downarrow r}\ (\textsc{NSIfR}) \qquad \frac{e_1[\rho] \Downarrow \mathtt{Blame}\ \ell}{(\mathtt{if}\ e_1\ e_2\ e_3)[\rho] \Downarrow \mathtt{Blame}\ \ell}\ (\textsc{NSIfB})$$

$$\frac{\rho\,!\,x \Downarrow r}{x[\rho_1] \Downarrow r}\ (\textsc{NSVar}) \qquad \frac{}{(\lambda x : T.e)[\rho] \Downarrow (\lambda\!\!\lambda.e[(x:T):\rho])}\ (\textsc{NSFun}) \qquad \frac{e_1[\rho] \Downarrow \mathtt{Blame}\ \ell}{(e_1\,e_2)[\rho] \Downarrow \mathtt{Blame}\ \ell}\ (\textsc{NSAppBL})$$

$$\frac{e_1[\rho] \Downarrow e[(x:T):\rho'] \qquad e_2[\rho] \Downarrow \mathtt{Blame}\ \ell}{(e_1\,e_2)[\rho] \Downarrow \mathtt{Blame}\ \ell}\ (\textsc{NSAppBR}) \qquad \frac{e_1[\rho] \Downarrow v_1 \qquad e_2[\rho] \Downarrow v \qquad v_1\ v_2 \Downarrow_{ap} r}{(e_1\,e_2)[\rho] \Downarrow r}\ (\textsc{NSApp})$$

$$\frac{e[\rho] \Downarrow v \qquad \langle c\rangle v \Downarrow_{cr} r}{(\langle c\rangle e)[\rho] \Downarrow r}\ (\textsc{NSCoe}) \qquad \frac{e[\rho] \Downarrow \mathtt{Blame}\ \ell}{(\langle c\rangle e)[\rho] \Downarrow \mathtt{Blame}\ \ell}\ (\textsc{NSCoeB}) \qquad \frac{}{(\mathtt{Blame}\ \ell)[\rho] \Downarrow \mathtt{Blame}\ \ell}\ (\textsc{NSBla})$$

Coercion application: $\boxed{\langle\hat{c}\rangle v \Downarrow_{cr} r}$

$$\frac{\overline{c}\,;\hat{d} \Downarrow_{\mathbf{ED}} \hat{c}_1}{\langle\hat{d}\rangle(\langle\overline{c}\rangle s) \Downarrow_{cr} \mathit{mkCast}(\hat{c}_1, s)}\ (\textsc{NSComp}) \qquad \frac{}{\langle\hat{c}\rangle s \Downarrow_{cr} \mathit{mkCast}(\hat{c}, s)}\ (\textsc{NSNorm})$$

Application: $\boxed{v\,v \Downarrow_{ap} r}$

$$\frac{e[(x \mapsto v):\rho] \Downarrow r}{(\lambda\!\!\lambda.e[(x:T):\rho])v \Downarrow_{ap} r}\ (\textsc{NsProc}) \qquad \frac{\langle\tilde{c}\rangle v \Downarrow_{cr} v_1 \qquad (e[(x:T):\rho])v_1 \Downarrow_{ap} v_2 \qquad \langle\tilde{d}\rangle v_2 \Downarrow_{cr} r}{(\langle\tilde{c} \to \tilde{d}\rangle(\lambda\!\!\lambda.e[(x:T):\rho]))v \Downarrow_{ap} r}\ (\textsc{NsArr})$$

$$\frac{\langle\tilde{c}\rangle v \Downarrow_{cr} \mathtt{Blame}\ \ell}{(\langle\tilde{c} \to \tilde{d}\rangle(\lambda\!\!\lambda.\mathsf{cl}))v \Downarrow_{ap} \mathtt{Blame}\ \ell}\ (\textsc{NsArrBL}) \qquad \frac{\langle\tilde{c}\rangle v \Downarrow_{cr} v_1 \qquad (\lambda\!\!\lambda.\mathsf{cl})\ v_1 \Downarrow_{ap} \mathtt{Blame}\ \ell}{(\langle\tilde{c} \to \tilde{d}\rangle(\lambda\!\!\lambda.\mathsf{cl}))v \Downarrow_{ap} \mathtt{Blame}\ \ell}\ (\textsc{NsArrBR})$$

Figure 7. Natural semantics for closure normalisation.

to closure scopes, and are needed to define the reduction contexts $\mathsf{Cl}[\,]$ in the bottom of the figure. Ephemeral constructors consist of closure constants **con** k, closure primitive application **prim** *op* cl, closure conditionals **if** $\mathsf{cl}\ \mathsf{cl}\ \mathsf{cl}$, closure applications $\mathsf{cl} \cdot \mathsf{cl}$, closure abstractions $\lambda\!\!\lambda.\mathsf{cl}$, closure coercion casts $\langle c\rangle\mathsf{cl}$, and closure blames **Blame** ℓ. The type system for closures is a straightforward extension of the type-system for expressions (Appendix A) and we omit it for lack of space. The hierarchy of results is the one for expressions but lifted to ephemeral closures. Observe that closure blames are closure results.

The contraction rules are separated in three groups. The first seven rules induce *ephemeral expansion*, *i.e.*, a relation that lifts proper closures to their corresponding ephemeral constructors, and distributes the outermost environment over the closure scopes. Ephemeral expansion is needed in small-step artefacts, but will be shortcut [2] when deriving the big-step semantics by applying compression of corridor transitions in Section 10.4.

Observe that in rule $\textsc{Lam}_\rho$ lambda abstractions are ephemerally expanded although reduction will not 'go under lambda'. We have introduced the ephemeral closure abstraction $\lambda\!\!\lambda.\mathsf{cl}$ to match the procedure representations in the closure-converted natural semantics of Figure 7. The $\lambda\!\!\lambda$ symbol helps discriminate between an input closure and the result of reducing an abstraction closure (recall the similar discussion in Section 6.1). Rule $\textsc{Lam}_\rho$ also pushes a type annotation $(x : T)$ on the environment, similar to rule NS-$\textsc{Fun}$ in Figure 7. The purpose of this is to close the scope of the abstraction body e such that every variable points to some element in the environment, preventing dangling variables in e.[4]

The second group of rules consists of rule $\textsc{Var}_\rho$ alone, which performs substitution on demand by looking up the binding of a variable. The lookup function always returns a closure cl because the reduction semantics never goes under lambda and the type system enforces that all the variables are bound.

The third and last group are the closure versions of the contraction rules in $\lambda^{\langle\cdot\rangle}_{\rightarrow}$. These rules induce reduction proper. Notice that (β_ρ) discards the $\lambda\!\!\lambda$ in the operator and replaces the formal parameter's type annotation by the actual binding.

The reduction contexts $\mathsf{Cl}[\,]$ and the reduction semantics $\longmapsto_\rho$ for closures are simply the closure version of the reduction contexts and reduction semantics of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$. The reduction semantics of $\lambda\rho^{\langle\cdot\rangle}_{\rightarrow}$ simulates step-by-step-wise the reduction semantics in $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ by means of substitution function σ that flattens all the delayed substitutions in a closure (see Appendix D for details).

8.1 The correctness theorems

We are now in a position to state the correspondence between the instantiated definitional interpreter and the reduction semantics for closures with respect to $\mathbf{ED}$:

Theorem 8.1. *Given a well-typed coercion c_1 we have $c_1 \Downarrow_{\mathbf{ED}} \hat{c}_2$ iff $c_1 \longmapsto^*_{\mathbf{ED}} \hat{c}_2$.*

Proof. By establishing the correspondence between the functions `normalise_coercion_nor` (Code 2.2.3, Section 5.2) and `normalise_coercion` (Code 5.3.2, Section 9). □

Theorem 8.2. *If every coercion labelled with ℓ in program e respects subtyping, then $e[\epsilon] \not\longmapsto^*_\rho$ $\mathsf{\mathbf{Blame}}\ \ell$.*

Proof. The proof is straightforward by induction on $\longmapsto_\rho$. □

Theorem 8.3. *Given a well-typed expression e, we have $e[\epsilon] \Downarrow \mathsf{r}$ iff $e[\epsilon] \longmapsto^*_\rho \mathsf{r}$.*

Proof. By establishing the correspondence between `eval_clos` (Code 3.1, Section 6.1) and `normalise` (Code 5.3.2, Section 9). □

Different from the conjectures in Section 4.3, we do not need a separate theorem stating soundness of subtyping for the natural semantics $\Downarrow$ because Theorem 8.3 proves it equivalent to reduction semantics $\longmapsto_\rho$.

[4] This feature is reminiscent of the dummy bindings standing for formal parameters in Cregut's full-reducing Krivine machine [3, 4], and in the equivalent semantic artefacts inter-derived in [16].

Syntax: $\boxed{\mathsf{cl} \in \lambda\rho^{\langle\cdot\rangle}_{\rightarrow}}$

environments	ρ	$::=$	$\epsilon \mid (x \mapsto \mathsf{v}) : \rho \mid (x : T) : \rho$
closures	cl	$::=$	$e[\rho] \mid \mathbf{con}\ k \mid \mathbf{prim}\ op\ \mathsf{cl} \mid \mathbf{if}\ \mathsf{cl}\ \mathsf{cl}\ \mathsf{cl} \mid \lambda\!\!\lambda.\mathsf{cl} \mid \mathsf{cl} \cdot \mathsf{cl} \mid \langle c \rangle \mathsf{cl} \mid \mathbf{Blame}\ \ell$
closure simple values	s	$::=$	$\mathbf{con}\ k \mid (\lambda\!\!\lambda.e[(x : T) : \rho])$
closure values	v	$::=$	$\mathsf{s} \mid \langle \overline{c} \rangle \mathsf{s}$
closure results	r	$::=$	$\mathsf{v} \mid \mathbf{Blame}\ \ell$

Contraction: $\boxed{\mathsf{cl} \longrightarrow_\rho \mathsf{cl}}$

$$k[\rho] \longrightarrow_\rho \mathbf{con}\ k \qquad (\textsc{Con}_\rho)$$
$$(op\ e)[\rho] \longrightarrow_\rho \mathbf{prim}\ op\ (e[\rho]) \qquad (\textsc{Prim}_\rho)$$
$$(\mathtt{if}\ e_1\ e_2\ e_3)[\rho] \longrightarrow_\rho \mathbf{if}\ (e_1[\rho])\ (e_2[\rho])\ (e_3[\rho]) \qquad (\textsc{IfTE}_\rho)$$
$$(\lambda x : T.e)[\rho] \longrightarrow_\rho (\lambda\!\!\lambda.e[(x : T) : \rho]) \qquad (\textsc{Lam}_\rho)$$
$$(e_1\ e_2)[\rho] \longrightarrow_\rho (e_1[\rho]) \cdot (e_2[\rho]) \qquad (\textsc{App}_\rho)$$
$$(\langle c \rangle e)[\rho] \longrightarrow_\rho \langle c \rangle (e[\rho]) \qquad (\textsc{Coer}_\rho)$$
$$(\mathtt{Blame}\ \ell)[\rho] \longrightarrow_\rho \mathbf{Blame}\ \ell \qquad (\textsc{Bla}_\rho)$$

$$x[\rho] \longrightarrow_\rho \mathsf{cl} \quad \text{where } \rho\,!\,x = \mathsf{cl} \qquad (\textsc{Var}_\rho)$$

$$(\lambda\!\!\lambda.e[(x : T) : \rho]) \cdot \mathsf{v} \longrightarrow_\rho e[(x \mapsto \mathsf{v}) : \rho] \qquad (\beta_\rho)$$
$$\mathbf{prim}\ op\ (n[\rho]) \longrightarrow_\rho \mathbf{con}\ (\delta(op, n)) \qquad (\delta_\rho)$$
$$\mathbf{if}\ (\mathbf{con}\ k)\ \mathsf{cl}_1\ \mathsf{cl}_2 \longrightarrow_\rho \begin{cases} \mathsf{cl}_1 & \text{if } k = \mathtt{t} \\ \mathsf{cl}_2 & \text{if } k = \mathtt{f} \end{cases} \qquad (\textsc{If}_\rho)$$
$$\boxed{\langle c_1 \rangle \mathsf{s} \longrightarrow_\rho \langle c_2 \rangle \mathsf{s} \quad \text{if } c_1 \longmapsto_{\mathbf{X}} c_2 \qquad (\textsc{StepCst}_\rho)}$$
$$\langle \iota \rangle \mathsf{s} \longrightarrow_\rho \mathsf{s} \qquad (\textsc{IdCst}_\rho)$$
$$\langle d \rangle \langle \overline{c} \rangle \mathsf{s} \longrightarrow_\rho \langle \overline{c}\,; d \rangle \mathsf{s} \qquad (\textsc{CmpCst}_\rho)$$
$$(\langle \tilde{c} \to \tilde{d} \rangle \mathsf{s}) \cdot \mathsf{v} \longrightarrow_\rho \langle \tilde{d} \rangle (\mathsf{s} \cdot \langle \tilde{c} \rangle \mathsf{v}) \qquad (\textsc{AppCst}_\rho)$$
$$\langle \mathtt{Fail}^\ell \rangle \mathsf{s} \longrightarrow_\rho \mathbf{Blame}\ \ell \qquad (\textsc{FailCast}_\rho)$$
$$\langle (\tilde{c} \to \tilde{d})\,; \mathtt{Fail}^\ell \rangle \mathsf{s} \longrightarrow_\rho \mathbf{Blame}\ \ell \qquad (\textsc{FailFC}_\rho)$$

Reduction semantics: $\boxed{\mathsf{cl} \longmapsto_\rho \mathsf{cl}}$

$$\mathsf{Cl}[\,] \quad ::= \quad [\,] \mid \mathbf{prim}\ op\ (\mathsf{Cl}[\,]) \mid (\mathsf{Cl}[\,]) \cdot \mathsf{cl} \mid \mathsf{v} \cdot (\mathsf{Cl}[\,]) \mid \mathbf{if}\ (\mathsf{Cl}[\,])\ \mathsf{cl}\ \mathsf{cl} \mid \langle c \rangle (\mathsf{Cl}[\,])$$

$$\frac{\mathsf{cl} \longrightarrow_\rho \mathsf{cl}'}{\mathsf{Cl}[\mathsf{cl}] \longmapsto_\rho \mathsf{Cl}[\mathsf{cl}']} \qquad \frac{}{\mathsf{Cl}[\mathbf{Blame}\ \ell] \longmapsto_\rho \mathbf{Blame}\ \ell}$$

Figure 8. Syntax, contraction rules, and implementable reduction semantics of $\lambda\rho^{\langle\cdot\rangle}_{\rightarrow}$.

9. Implementing the reduction semantics

We turn to the implementation of $\longmapsto_\rho$ in Figure 8. Similarly to the 2CPS discussion of Section 6.2 we use continuations for $\longmapsto_{\mathbf{X}}$ (the reduction semantics of coercions) and meta-continuations for $\longmapsto_\rho$ (the reduction semantics of closures). In Code 5 we describe a transformation step which is uninteresting to the gradual-type-theorist and for lack of space we merely outline it. We start with layered search functions that implement a structural operational semantics. We then derive the reduction semantics form the search functions, by CPS transformation, simplification, and defunctionalisation. This standard practice [2, 7, 8, 10, 11] is not essential to establish a syntactic correspondence, but it reveals better the correspondence between reduction contexts and defunctionalised continuations. Moreover, the transformation step clarifies two important points and justifies the accompanying design decisions. We elaborate on this two points in the following paragraphs. We strongly advise the program-derivationist to read Code 5 alongside this section.

The first point: the simplification step prescribes that the search functions discard the current continuation when a redex is found. However, a tail-recursive implementation of our layered semantics would require to keep the closure meta-continuation in order to throw into it the found coercion redex. Since the closure meta-continuation will be dropped by the simplified coercion semantics, the closure semantics needs to invoke the coercion semantics in non-tail-recursive fashion, by delimiting its invocation by passing the initial continuation.[5] Thus, the implementation of the small-step semantics is not a 2CPS program anymore, but two 1-level CPS programs which are glued together by the closure semantics delimiting the invocations of the coercion semantics. All this is unavoidable. Dropping the meta-continuation in the inner simplified semantics is essential for the separated transformation of closure and coercion semantics. For the coercions, the tail calls to iterate need to happen immediately after decompose, enabling to lightweight fuse them in Code 6.3.

The second point: decomposition (to have a term and its context) is fundamental to implement a trampolined style reduction semantics [14] (a driver loop iterating decomposition, contraction, and recomposition). We have a layered reduction semantics involving closures and coercions. The following rule (implicitly entailed by $\textsc{StepCst}_\rho$ in Figure 8) illustrates the inclusion of the inner semantics in the outer one:

$$\frac{\mathsf{C}[c] \longmapsto_{\mathtt{x}} \mathsf{C}[c']}{\mathsf{Cl}[\langle \mathsf{C}[c]\rangle s] \longmapsto_{\rho} \mathsf{Cl}[\langle \mathsf{C}[c']\rangle s]}$$

In order to implement the subsidiary coercion semantics $\longmapsto_{\mathtt{x}}$ in trampolined style, we have to modify the datatype representing the outer redices $\mathsf{Cl}[\langle \mathsf{C}[c]\rangle s]$ to include the inner decomposition $\mathsf{C}[c]$, rather than just a plain coercion $c_1 \equiv \mathsf{C}[c]$. This is implemented by clause `ESTEPCST1 of decomposition * simple_value` of datatype `redex1` in Code 5.3.1.

Code 5.3 implements the reduction semantics $\longmapsto_\rho$ in Figure 8, which corresponds to the top left corner in Figure 1. In the following sections, we apply the syntactic correspondence and arrive at an abstract machine which will be refunctionalised into the 2CPS-normaliser in Section 6.2 and Code 3.2, thus closing the gap and completing the inter-derivation.

10. The syntactic correspondence

In Section 9, we implemented the reduction semantics $\longmapsto_\rho$, which is the starting point of the syntactic correspondence arriving at the abstract machine on the bottom left corner of Figure 1. The syntactic correspondence [6, 7, 9] consists of refocusing, inlining of contraction function, lightweight-fusion by fix point promotion [19], and compression of corridor transitions. These steps are standard and hence merely outlined in the paper, except for the specific details concerning our layered semantics. In the fourth step, we elaborate on two different classes of corridor transitions found in the literature [9] (Section 10.4).

On occasion, we generically refer to both the coercion and closure artefacts by naming the entry function, *e.g.*, `normalise1` in Code 6.1.

[5] The sceptical program-derivationist is invited to attempt the simplification step in a true 2CPS program implementing a semantics with layered redices, like the ones entailed by rule $\textsc{StepCst}_\rho$ in Figure 8.

10.1 Refocusing

The refocus function maps a pair (contractum, context) to the decomposition for the next redex in the reduction sequence. Extensional refocus consist of (respectively on coercions and closures) recomposition followed by decomposition. The refocusing step deforests this detour, turning the extensional refocus function into an intensional refocus function which is an alias for the decompose function [6].

Since our semantics is layered, we apply refocusing to the coercion and the closure artefacts in succession. First, we turn the extensional `refocus_coercion` in Code 5.3.2 into the intensional `refocus1_coercion` in Code 6.1 which is an alias for `decompose_coercion`. Functions `iterate1_coercion` and `normalise1_coercion` follow from that. Before performing the same operation in the closure artefact, we coalesce all the $\textsc{StepCst}_\rho$ steps in the closure reduction semantics. The modified inclusion-of-semantics rule now reads:

$$\frac{c \longmapsto_{\mathtt{x}}^{*} \hat{c}}{\mathsf{Cl}[\langle c\rangle s] \longmapsto_{\rho} \mathsf{Cl}[\langle \hat{c}\rangle s]}$$

This transformation trivially preserves equivalence. To implement the rule above, we modify the decomposition of closures accordingly. In Code 6.1, the clause for meta-continuation `MC5` in function `decompose1_meta_cont` now invokes `normalise1_coercion`. In the same program clause, there is no case returning a `ESTEPCST1` redex, since the coercion found `nc` (on which the program is pattern-matching after `normalise1_coercion`) is trivially a normal coercion.

We turn the extensional `refocus_closure` in Code 5.3.2 into the intensional `refocus1_closure` in Code 6.1 which is an alias for `decompose1_closure`.

10.2 Inlining the contraction function

We inline the contraction functions in the corresponding iterate functions [9], obtaining `normalise2` in Code 6.2. Due to the modified rule in Section 10.1, the case for `ESTEPCST1` redices in `contract_closure` is no longer considered.

10.3 Lightweight-fusing decompose and iterate

There are several invocations of decomposition followed by iteration in the iterate and normalise functions. We fuse them together in a single normalise function by applying lightweight fusion. As in Section 10.1, we proceed in succession for the coercion and the closure artefacts. The resulting reduction-free normaliser `normalise3` is in Code 6.3.

10.4 Compressing *static* and *dynamic* corridor transitions

Some of the transitions in `normalise3` are to configurations where there is only one possible further transition. These are called *corridor* transitions, and by hereditarily compressing them, the iterate functions will become unused and could be safely removed.

The conventional corridor transitions (for which we use the epithet *static*) are those detected by looking at the code of the normalising functions, *i.e.*, the program clauses where the right-hand side consists of a single tail call, or of a selection statement having a unique case [7, 9]. These transitions are compressed by successively unfolding the right-hand side of the program clauses involved. The shortcut operation coalescing ephemeral expansion [2] belongs to this category of corridor transitions. By compressing the static corridor transitions we obtain the big-step normaliser `normalise4` in Code 6.4.

The not-so-conventional corridor transitions (for which we use the epithet *dynamic*), are those starting at a configuration where the input term is irreducible, *i.e.*, a normal coercion or a value [9, p.140-141]. For coercions, remember from Section 4 that we use a single `coercion` datatype both for arbitrary coercions and for the hierarchy of normal coercions. For the closures, in Code 4 we introduced datatype `value_clos` and function `embed_clos`, the latter implementing the embedding function $\downarrow$v in Appendix C.[6] Since the programs are in defunctionalised CPS, all the calls are tail calls (respectively to the coercion or closure semantics). The computation will eventually throw the irreducible input term into the current continuation (meta-continuation respectively). Thus, these administrative transitions can be coalesced until that point, *i.e.*, a call to `normalise4_cont` or `normalise4_meta_cont` respectively. Compressing dynamic corridor transitions reveals more opportunities to compress static corridor transitions. By compressing them all we obtain `normalise5` in Code 6.5, which implements the abstract machine at the bottom left corner of Figure 1.

Let us show one of such dynamic corridor transitions:

```
  normalise4_closure
  (embed_clos f1, MC3 (CCOER (c1, embed_clos v),
                       MC5 (c2, mk)))
= normalise4_meta_cont
  (MC3 (CCOER (c1, embed_clos v), MC5 (c2, mk)), f1)
= normalise4_closure
  (CCOER (c1, embed_clos v), MC4 (f1, MC5 (c2, mk)))
= normalise4_closure
  (embed_clos v, MC5 (c1, MC4 (f1, MC5 (c2, mk))))
= normalise4_meta_cont
  (MC5 (c1, MC4 (f1, MC5 (c2, mk))), v)
```

The normaliser specifies a control-flow invariant for the cases matching the initial clause of the corridor transition. The invariant allows the meta-continuation stack to be loaded with a *fixed* sequence of defunctionalised meta-continuations, which will help us refunctionalise the abstract machine into several mutually recursive functions in Section 11.1.

11. Closing the gap

In this section we close the gap between the right- and left-hand sides of the inter-derivation diagram (Figure 1). We defunctionalise the abstract machine into a 2CPS program with several mutually recursive functions which is almost the 2CPS-normaliser in Figure 1. Then, we apply some cosmetic transformations to remove minor differences between the refunctionalised abstract machine and the 2CPS-normaliser, thus concluding the derivation.

11.1 Refunctionalising the abstract machine

We observe two facts about function `normalise5_meta_cont`:

1. In the clause for `MC5`, the program either invokes the clause itself (`normalise5_meta_cont` passing `MC5`), or makes a delimited non-tail call to `normalise5_coercion` and then returns a blame or throws some intermediate result into the current meta-continuation. This clause can be refunctionalised into a standalone recursive function, which we name `apply6_coercion` (Code 7.1).

2. In the clause for `MC4`, the program either calls the function `normalise5_closure`, or invokes `normalise5_meta_cont` passing `MC5 (c1, MC4 (f1, MC5 (c2, mk)))`. This clause can be refunctionalised into a stand alone recursive function which unwinds the defunctionalised continuation, and invokes `apply6_coercion` and itself according to the occurrences of `MC4` and `MC5` in the defunctionalised continuation. We name the function `apply6` (Code 7.1).

   Although the clause for `MC4` invokes `normalise5_closure`, equivalence is preserved because the latter never invokes directly `normalise5_meta_cont` passing `MC4`.

The rest of the clauses and functions are straightforwardly defunctionalised.

We also undelimit the inner continuations, turning the non-tail calls into tail calls. This is only possible at a reduction-free artefact, since the program does not need to consider the individual redices in the reduction sequence, in particular the coercion redices (recall the discussion in Section 9). We have decided to do this transformation after refunctionalisation to save us from introducing a new datatype for defunctionalised continuations. The result is the 2CPS refunctionalised abstract machine `normalise6` in Code 7.1.

11.2 Cosmetic transformations

We remove some minor differences between the refunctionalised abstract machine `normalise6` in Code 7.1 and the 2CPS-normaliser `eval_cps` in Code 3.2.

We inline `apply6_coercion` in Code 7.1 into itself (notice that the value `sv` passed in the recursive call is a simple value), duplicating the selection statement in the second clause. This selection statement is, in turn, protruded (*i.e.*, inversely inlined) into combinator `mk_cast7` (Code 7.2). We unfold datatype `closure` into a pair (`expression`, `environment_clos`) and protrude combinator `mk_arrow7` (Code 7.2). The result is `normalise7` in Code 7.2, which is exactly the same as the 2CPS-normaliser `eval_clos` in Code 3.2.

This establishes the correspondence between $=_{\langle\cdot\rangle}$ *and* $\longmapsto$*, and between* $\Downarrow_{\mathbf{ED}}$ *and* $\longmapsto_{\mathbf{ED}}$*, constituting a proof by program transformation of Theorems 8.1 and 8.3.*

Theorems 8.1 and 8.3 can be generalised for other choices of dynamic semantics by applying the correspondence described through the paper starting with a different set of coercion artefacts. Layering and 2CPS allows us to reuse the off-the-shelf infrastructure, in particular the closure artefacts.

12. Conclusions and related work

We have shown the inter-derivation of semantic artefacts for $\lambda^{\langle\cdot\rangle}_{\rightarrow}$. Our choice of the 2CPS is motivated by the need for a modular coercion semantics that can be plugged into the $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ layered semantics. This allows us to generalise the theorems for a family of dynamic semantics reusing most of the inter-derivation for $\lambda^{\langle\cdot\rangle}_{\rightarrow}$.

We have presented the calculus $\lambda\rho^{\langle\cdot\rangle}_{\rightarrow}$, which is an important ingredient to inter-derive the closure-converted version of the definitional interpreters in [20]. The semantics in $\lambda\rho^{\langle\cdot\rangle}_{\rightarrow}$ simulates step-by-step-wise the semantics in $\lambda^{\langle\cdot\rangle}_{\rightarrow}$.

In [12] the 2CPS is applied to inter-derive a full-reducing *eval-readback* machine of Curien [5] that normalises pure untyped lambda calculus terms. The machine relies on a *hybrid*, or *layered*, reduction strategy with two separated stages for eval and readback. We have independently investigated in [16] a different approach for *single-stage* (as opposed to eval-readback) hybrid artefacts, showcasing the derivation of the full-reducing Krivine machine [3] from the operational semantics of normal order. In [12] the subsidiary strategy is modular, but this introduces a conceptual overhead in the 2CPS transformations. In [16] we show how to use plain CPS when the target strategy is single-staged, but this requires reasoning on the shape of the continuation stack. Both approaches differ in their weaknesses and strengths, as well as in their range of applicability.

The semantic artefacts in this paper are qualitatively different from those in [12] and [16]. The semantics $\Downarrow$ here is single-stage (there is only one pass of the big-step definitional interpreter) but its implementation is 2-levelled. In the big-step artefacts we use 2CPS. In the small-step artefacts we disentangle the inner continuation

[6] The embedding function is used in `contract_closure` in Code 5.3.2. Embed only considers closure values because closure blames are short-circuited to closure results and do not appear in redices. Observe that embed followed by normalise is the identity.

space by delimiting the continuations in the non-tail calls to the coercion semantics. This way, we keep the semantics in [20], but arrive at a solution which is modular with respect to the coercion semantics.

Garcia [15] has tackled and solved the challenge of defining a reduction semantics for coercions which is 'complete in the face of inert compositions and associativity'. We have rather followed the 'ad hoc reassociation scheme' in [20], proving the correctness conjectures there. Garcia also introduces threesome-based variants of the Blame Calculus. We believe that the semantics for the threesome-based gradually-typed lambda calculi are good candidates for applying the techniques in this paper. Thanks to layering and 2CPS, modularity with respect to the blame calculi would be straightforward.

Acknowledgements We are thankful to Olivier Danvy for introducing us to [20] and for suggesting the problem.

References

[1] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, 2003.

[2] M. Biernacka and O. Danvy. A concrete framework for environment machines. *ACM Trans. Comput. Log*, 9(1):6:1–6:29, December 2007.

[3] P. Crégut. An abstract machine for lambda-terms normalization. In *Proceedings of LISP and Functional Programming*, pages 333–340, 1990.

[4] P. Crégut. Strongly reducing variants of the Krivine abstract machine. *Higher-Order and Symbolic Computation*, 20(3):209–230, September 2007.

[5] P-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Progress in Theoretical Computer Science. Birkhaüser, 1993.

[6] O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. Technical Report RS-04-26, BRICS, Department of Computer Science, Aarhus University, Denmark, November 2004.

[7] O. Danvy, J. Johannsen, and I. Zerny. A walk in the semantic park. In *Proceedings of the Workshop on Partial Evaluation and Program Manipulation*, 2011.

[8] Olivier Danvy. Defunctionalized interpreters for programming languages. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, 2008.

[9] Olivier Danvy. From reduction-based to reduction-free normalization. In *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 66–164. Springer, 2008.

[10] Olivier Danvy and Kevin Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Information Processing Letters*, 106(3):100–109, 2008.

[11] Olivier Danvy and Kevin Millikin. Refunctionalization at work. *Sci. Comput. Program*, 74(8):534–549, 2009.

[12] Olivier Danvy, Kevin Millikin, and Johan Munk. A correspondence between full normalization by reduction and full normalization by evaluation, 2013. A scientific meeting in honor of Pierre-Louis Curien.

[13] M. Felleisen. *The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Department of Computer Science, Indiana University, 1987.

[14] S. E. Ganz, D. P. Friedman, and M. Wand. Trampolined style. In *Proceedings of International Conference on Functional Programming*, 1999.

[15] Ronald Garcia. Calculating threesomes, with blame. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, 2013.

[16] A. García-Pérez, Pablo Nogueira, and Juan José Moreno-Navarro. Deriving the *Full-Reducing* krivine machine from the small-step operational semantics of normal order. In *Proceedings of the 15th International Symposium on Principles and Practice of Declarative Programming*, 2013.

[17] Álvaro García-Pérez and P. Nogueira. A syntactic and functional correspondence between reduction semantics and reduction-free full normalisers. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM*, 2013.

[18] P. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, January 1964.

[19] A. Ohori and I. Sasano. Lightweight fusion by fixed point promotion. In *Proceedings of the Symposium on Principles of Programming Languages*, 2007.

[20] Jeremy G. Siek and Ronald Garcia. Interpretations of the gradually-typed lambda calculus. In *Proceedings of the Workshop on Scheme and Functional Programming*, 2012.

[21] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Proceedings of the Workshop on Scheme and Functional Programming*, 2006.

[22] Jeremy G. Siek, Ronald Garcia, and Walid Taha. Exploring the design space of higher-order casts. In *Proceedings of the 18th European Symposium on Programming Languages and Systems*, 2009.

A. Subsystems of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$

Complements of Figure 2 on page 2.

δ-rules $\boxed{\delta(op, n) = k}$

$$\begin{array}{rcl}
\delta(\texttt{inc}, n) & = & n+1 \\
\delta(\texttt{dec}, n) & = & n-1 \\
\delta(\texttt{zero?}, 0) & = & \texttt{t} \\
\delta(\texttt{zero?}, n) & = & \texttt{f} \quad (n \neq 0)
\end{array}$$

Type system for expressions of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ $\boxed{\Gamma \vdash e : T}$

$$\frac{}{\Gamma \vdash k : \mathit{typeC}(k)} \qquad \frac{}{\Gamma \vdash \mathit{op} : \mathit{typeO}(\mathit{op})} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \qquad \Gamma \vdash e_2 : T \qquad \Gamma \vdash e_3 : T}{\Gamma \vdash \texttt{if}\ e_1\ e_2\ e_3 : T}$$

$$\frac{\Gamma, x \mapsto T \vdash e : S}{\Gamma \vdash (\lambda x : T.e) : T \to S} \qquad \frac{\Gamma \vdash e_1 : T \to S \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1\ e_2 : S}$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \langle S \Leftarrow T \rangle^{\ell} e : S} \qquad \frac{\vdash c : S \Leftarrow T \qquad \Gamma \vdash e : T}{\Gamma \vdash \langle c \rangle e : S}$$

$$\frac{}{\Gamma \vdash \texttt{Blame}\ \ell : T}$$

Type of constants: $\boxed{\mathit{typeC}(k) = B}$

$$\begin{array}{rcl}
\mathit{typeC}(n) & = & \texttt{Int} \\
\mathit{typeC}(\texttt{t}) & = & \texttt{Bool} \\
\mathit{typeC}(\texttt{f}) & = & \texttt{Bool}
\end{array}$$

Type of operators: $\boxed{\mathit{typeO}(\mathit{op}) = B \to B}$

$$\begin{array}{rcl}
\mathit{typeO}(\texttt{inc}) & = & \texttt{Int} \to \texttt{Int} \\
\mathit{typeO}(\texttt{dec}) & = & \texttt{Int} \to \texttt{Int} \\
\mathit{typeO}(\texttt{zero?}) & = & \texttt{Int} \to \texttt{Bool}
\end{array}$$

B. Subsystems of ED

Complements of Figure 3 on page 3.

Translation function for casts $\boxed{\langle\!\langle T \Leftarrow T \rangle\!\rangle^{\ell} = \hat{c}}$

$$
\begin{array}{rcll}
\langle\!\langle B \Leftarrow B \rangle\!\rangle^{\ell} & = & \iota & \\
\langle\!\langle B_2 \Leftarrow B_1 \rangle\!\rangle^{\ell} & = & \mathtt{Fail}^{\ell} & \text{if } B_1 \neq B_2 \\
\langle\!\langle \mathtt{Dyn} \Leftarrow \mathtt{Dyn} \rangle\!\rangle^{\ell} & = & \iota & \\
\langle\!\langle \mathtt{Dyn} \Leftarrow B \rangle\!\rangle^{\ell} & = & B! & \\
\langle\!\langle B \Leftarrow \mathtt{Dyn} \rangle\!\rangle^{\ell} & = & B?^{\ell} & \\
\langle\!\langle T_1 \to T_2 \Leftarrow B \rangle\!\rangle^{\ell} & = & \mathtt{Fail}^{\ell} & \\
\langle\!\langle B \Leftarrow S_1 \to S_2 \rangle\!\rangle^{\ell} & = & \mathtt{Fail}^{\ell} & \\
\langle\!\langle T_1 \to T_2 \Leftarrow S_1 \to S_2 \rangle\!\rangle^{\ell} & = & \mathit{mkArr}(\,\langle\!\langle S_1 \Leftarrow T_1 \rangle\!\rangle^{\ell}, & \\
 & & \qquad\quad \langle\!\langle T_2 \Leftarrow S_2 \rangle\!\rangle^{\ell}) & \\
\langle\!\langle \mathtt{Dyn} \Leftarrow S_1 \to S_2 \rangle\!\rangle^{\ell} & = & S_1 \to S_2! & \\
\langle\!\langle T_1 \to T_2 \Leftarrow \mathtt{Dyn} \rangle\!\rangle^{\ell} & = & T_1 \to T_2?^{\ell} &
\end{array}
$$

Arrow combinator $\boxed{\mathit{mkArr}(\hat{c}, \hat{c}) = \hat{c}}$

$$
\begin{array}{rcll}
\mathit{mkArr}(\mathtt{Fail}^{\ell}, \hat{c}_2) & = & \mathtt{Fail}^{\ell} & \\
\mathit{mkArr}(\hat{c}_1, \mathtt{Fail}^{\ell}) & = & \mathtt{Fail}^{\ell} & \\
\mathit{mkArr}(\hat{c}_1, \hat{c}_2) & = & \hat{c}_1 \to \hat{c}_2 & \text{otherwise}
\end{array}
$$

Type system for coercions $\boxed{\vdash c : T \Leftarrow T}$

$$
\frac{}{\vdash \iota : T \Leftarrow T} \qquad \frac{}{\vdash T! : \mathtt{Dyn} \Leftarrow T} \qquad \frac{}{\vdash T?^{\ell} : T \Leftarrow \mathtt{Dyn}}
$$

$$
\frac{\vdash c : S_1 \Leftarrow T_1 \qquad \vdash d : T_2 \Leftarrow S_2}{\vdash (c \to d) : (T_1 \to T_2) \Leftarrow (S_1 \to S_2)}
$$

$$
\frac{\vdash c : T_1 \Leftarrow T_2 \qquad \vdash d : T_2 \Leftarrow T_3}{\vdash (c\,;d) : T_1 \Leftarrow T_3} \qquad \frac{}{\vdash \mathtt{Fail}^{\ell} : T \Leftarrow S}
$$

C. Auxiliary functions and combinators

Look-up function for the original interpreter: $\boxed{\rho\,!\,x = \mathsf{cl}}$

$\rho\,!\,x = \mathsf{cl}$ where $(x \mapsto \mathsf{cl})$ contains the first occurrence of x in ρ

Cast combinator $\boxed{\mathit{mkCast}(\hat{c}, s) = r}$

$$
\begin{array}{rcl}
\mathit{mkCast}(\iota, s) & = & s \\
\mathit{mkCast}(\mathtt{Fail}^{\ell}, s) & = & \mathtt{Blame}\,\ell \\
\mathit{mkCast}((\tilde{c} \to \tilde{d}\,; \mathtt{Fail}^{\ell}), s) & = & \mathtt{Blame}\,\ell \\
\mathit{mkCast}(\overline{c}, s) & = & \langle \overline{c} \rangle s
\end{array}
$$

Environments for the closure-converted interpreter:

$$
\rho \quad ::= \quad \epsilon \mid (x \mapsto v) : \rho \mid (x : T) : \rho
$$

Closure-converted look-up function: $\boxed{\rho\,!\,x = \{\mathsf{cl} | T\}}$

$$
\rho\,!\,x \quad = \quad \begin{cases} \mathsf{cl} & \text{if } \mathit{bind} = (x \mapsto \mathsf{cl}) \\ T & \text{if } \mathit{bind} = (x : T) \end{cases}
$$
where *bind* contains the first occurrence of x in ρ

Embed function for closure values: $\boxed{\downarrow \mathsf{v} = e[\rho]}$

$$
\begin{array}{rcll}
\downarrow (\textsf{con}\ k) & = & k[\epsilon] & \\
\downarrow (\boldsymbol{\lambda}.e[(x : T) : \rho]) & = & (\lambda x : T.e)[\rho] & \\
\downarrow (\boldsymbol{\langle} c \boldsymbol{\rangle} \mathsf{v}) & = & (\langle c \rangle e)[\rho] & \text{where } \downarrow \mathsf{v} = e[\rho]
\end{array}
$$

D. Step-by-step simulation

The substitution function σ in Figure 9 flattens a closure by performing all the delayed substitutions in its environment. Function σ simulates capture-avoiding substitution in $\lambda^{\langle\cdot\rangle}_{\rightarrow}$, that is,

$$
\sigma(e_1[(x \mapsto e_2[\rho]) : \rho']) \equiv \sigma(([\sigma(e_2[\rho])/x]e_1)[\rho'])
$$

Flattening delayed substitutions: $\boxed{\sigma(\mathsf{cl}) = e}$

$$
\begin{array}{rcl}
\sigma(k[\rho]) & = & \sigma(\textsf{con}\ k) \\
\sigma((op\ e)[\rho]) & = & \sigma(\textsf{prim}\ op\ (e[\rho])) \\
\sigma((\mathtt{if}\ e_1\ e_2\ e_3)[\rho]) & = & \sigma(\textsf{if}\ (e_1[\rho])\ (e_2[\rho])\ (e_3[\rho])) \\
\sigma(x[\rho]) & = & \begin{cases} \sigma(\mathsf{cl}) & \text{if } \rho\,!\,x = \mathsf{cl} \\ x & \text{if } \rho\,!\,x = T \end{cases} \\
\sigma((\lambda x : T.e)[\rho]) & = & \sigma(\boldsymbol{\lambda}.e[(x : T) : \rho]) \\
\sigma((e_1\ e_2)[\rho]) & = & (\sigma(e_1[\rho]))(\sigma(e_2[\rho])) \\
\sigma((\langle c \rangle e)[\rho]) & = & \sigma(\boldsymbol{\langle} c \boldsymbol{\rangle}(e[\rho])) \\
\sigma((\mathtt{Blame}\ \ell)[\rho]) & = & \sigma(\textsf{Blame}\ \ell) \\
\sigma(\textsf{con}\ k) & = & k \\
\sigma(\textsf{prim}\ op\ \mathsf{cl}) & = & op\ (\sigma(\mathsf{cl})) \\
\sigma(\textsf{if}\ \mathsf{cl}_1\ \mathsf{cl}_2\ \mathsf{cl}_3) & = & \mathtt{if}\ (\sigma(\mathsf{cl}_1))\ (\sigma(\mathsf{cl}_2))\ (\sigma(\mathsf{cl}_3)) \\
\sigma(\boldsymbol{\lambda}.e[(x : T) : \rho]) & = & \lambda x : T.\sigma(e[\rho]) \\
\sigma(\mathsf{cl}_1 \cdot \mathsf{cl}_2) & = & (\sigma(\mathsf{cl}_1))(\sigma(\mathsf{cl}_2)) \\
\boldsymbol{\langle} c \boldsymbol{\rangle}\mathsf{cl} & = & \langle c \rangle(\sigma(\mathsf{cl})) \\
\sigma(\textsf{Blame}\ \ell) & = & \mathtt{Blame}\ \ell
\end{array}
$$

Height of a proper closure: $\boxed{h(\mathsf{cl}) = n}$

$$
\begin{array}{rcl}
h(k[\rho]) & = & h(\textsf{con}\ k) \\
h((op\ e)[\rho]) & = & h(\textsf{prim}\ op\ (e[\rho])) \\
h((\mathtt{if}\ e_1\ e_2\ e_3)[\rho]) & = & h(\textsf{if}\ (e_1[\rho])\ (e_2[\rho])\ (e_3[\rho])) \\
h(x[\rho]) & = & \begin{cases} h(\mathsf{cl}) & \text{if } \rho\,!\,x = \mathsf{cl} \\ 0 & \text{if } \rho\,!\,x = T \end{cases} \\
h((\lambda x : T.e)[\rho]) & = & h(\boldsymbol{\lambda}.e[(x : T) : \rho]) \\
h((e_1\ e_2)[\rho]) & = & h(\mathsf{cl}_1 \cdot \mathsf{cl}_2) \\
h((\langle c \rangle e)[\rho]) & = & h(\boldsymbol{\langle} c \boldsymbol{\rangle} e[\rho]) \\
h((\mathtt{Blame}\ \ell)[\rho]) & = & h(\textsf{Blame}\ \ell) \\
h(\textsf{con}\ k) & = & 0 \\
h(\textsf{prim}\ op\ \mathsf{cl}) & = & h(\mathsf{cl}) \\
h(\textsf{if}\ \mathsf{cl}_1\ \mathsf{cl}_2\ \mathsf{cl}_3) & = & \max\{h(\mathsf{cl}_1), h(\mathsf{cl}_2), h(\mathsf{cl}_3)\} \\
h(\boldsymbol{\lambda}.e[(x : T) : \rho]) & = & 1 + h(e[(x : T) : \rho]) \\
h(\mathsf{cl}_1 \cdot \mathsf{cl}_2) & = & \max\{h(\mathsf{cl}_1), h(\mathsf{cl}_2)\} \\
h(\boldsymbol{\langle} c \boldsymbol{\rangle}\mathsf{cl}) & = & h(\mathsf{cl}) \\
h(\textsf{Blame}\ \ell) & = & 0
\end{array}
$$

Figure 9. Substitution function and height of a closure.

This simulation property is proven by induction on the height of $e_1[(x \mapsto e_2[\rho]) : \rho']$, which is calculated by function h shown in Figure 9. The function σ connects $\longmapsto_{\rho}$ in $\lambda\rho^{\langle\cdot\rangle}_{\rightarrow}$ with $\longmapsto$ in $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ at a *step-by-step* level. The following diagram illustrates:

$$
\begin{tikzcd}
\mathsf{cl} \arrow[r, dashed, mapsto, "*", "\rho"'] & \mathsf{cl}' \arrow[r, mapsto, "\rho"'] & \mathsf{cl}_1 \arrow[r, dashed, mapsto, "*", "\rho"'] \arrow[d, "\sigma"'] & \mathsf{cl}_1' \arrow[r, mapsto, "\rho"'] & \mathsf{cl}_2 \cdots\cdots \arrow[d, "\sigma"'] \\
e \arrow[u, hook] \arrow[rr, mapsto] & & e_1 \arrow[rr, mapsto] & & e_2 \cdots\cdots
\end{tikzcd}
$$

The input expression e is injected into the closure $e[\epsilon]$, abbreviated cl. The closures cl_i map via σ to expressions e_i which are the result of step-by-step reduction relation in $\lambda^{\langle\cdot\rangle}_{\rightarrow}$. In the upper row of the diagram, the dashed arrow denotes the union of ephemeral expansion with on-demand substitution, and the solid arrow denotes reduction proper. Due to the reduction contexts $\mathsf{Cl}[\,]$, which are exactly $\mathsf{E}[\,]$ from $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ but lifted to the closure level, the dashed arrow will ephemerally expand and substitute cl until finding the redex corresponding to the next $\longmapsto$ step. The step-by-step connection rests on the property that σ simulates capture-avoiding substitution. The σ commutes with $\longmapsto_{\rho}$, *i.e.*, given $e_1[\rho_1] \longmapsto_{\rho} e_2[\rho_2]$ and $e_3[\rho_3] \longmapsto_{\rho} e_4[\rho_4]$, it is the case that

$$
\sigma(e_1[\rho_1]) \equiv \sigma(e_3[\rho_3]) \quad \text{iff} \quad \sigma(e_2[\rho_2]) \equiv \sigma(e_4[\rho_4])
$$