Space-Efficient Manifest Contracts

Michael Greenberg
University of Pennsylvania

September 11, 2013

Abstract

Gradual types mediate the interaction between dynamic and simple types, offering an easy transition from scripts to programs; gradual types allow programmers to evolve prototype scripts into fully fledged, deployable programs. Similarly, *contracts* and *refinement types* mediate the interaction between simple types and more precise types, offering an easy transition from programs to robust, verified programs. A *full-spectrum* language with both gradual and refinement types offers low-level support for the development of programs throughout their lifecycle, from prototype script to verified program.

One attractive formulation of languages with gradual or refinement types uses *casts* to represent the runtime checks necessary for type changes (from dynamic to simple types, and from simple types to refinement types). Briefly, a cast $\langle T_1 \Rightarrow T_2 \rangle\ e$ takes a term e from type T_1 to T_2, possibly wrapping or tagging e in the process—or even failing, if e doesn't meet the criteria of the type T_2. Casts are attractive because they offer a unified view of changes in type, have straightforward operational semantics, and enjoy an interesting and fruitful relationship with subtyping.

One longstanding problem with casts is space efficiency: casts in their naïve formulation can consume unbounded amounts of space at runtime both through excessive wrapping as well as through tail-recursion-breaking stack growth. Prior work [21, 22, 36, 38] offers space-efficient solutions exclusively in the domain of gradual types. In this paper, we define a new full-spectrum language that is (a) more expressive than prior languages, and (b) space efficient. We are the first to obtain space-efficient refinement types. Our approach to space efficiency is based on the *coercion* calculi of Herman et al. [21] and Henglein's work [20], though our explicitly enumerated canonical coercions and our straightforward merge operator are a novel approach to coercions with a simpler theory. We show that space efficiency avoids some checks, failing and diverging less often than naïve calculi—but the two are otherwise observationally equivalent.

Figure 1: The dyn/refine spectrum of cast expressiveness

1 Introduction

The promise of a single language admitting the full development cycle—from a small script to a more manageable, statically typed program to a robust, verified system—has held great allure for some time. Prior attempts to fulfill this promise have attacked the problem piecemeal: script to program, and program to verified program. We find it useful to think of one axis of the design space as a spectrum of expressivity, ranging from dynamic types on the left to dependent refinement types on the right (illustrated in Figure 1). We call it the "dyn/refine spectrum". On the one hand, work in the script-to-program category goes back at least to Abadi et al.'s work with type Dynamic, with more recent work falling under Siek and Taha's rubric of "gradual typing" [1, 40, 37, 28, 41, 2, 36, 10, 7, 21, 38]. On the other hand, program verification is an enormous field in its own right; in this paper, we will focus on dynamic enforcement methods. These methods, often called *contracts*, go back at least as far Eiffel [29, 12, 19, 18, 3, 34, 17, 15]. More recent work takes a type-oriented, or *manifest*, approach to contracts, allowing so-called *refinement types* of the form $\{x{:}T \mid e\}$, inhabited by values v that satisfy the *predicate* e, i.e. $e[v/x] \longrightarrow^* \mathsf{true}$. For example, $\{x{:}\mathsf{Int} \mid x \neq 0\}$ denotes the non-zero integers. Dependency increases the expressiveness of refinement type systems. Consider the type $(x{:}\mathsf{Real}) \to \{y{:}\mathsf{Real} \mid \left|x - y^2\right| < \epsilon\}$. This type specifies the square root function: for any real number x, functions in this type produce a y such that y^2 is within ϵ of x. Note that the type of the result depends on the input *value*.

Over the last decade, the state of the art combining these two paradigms—and gradual types and manifest contracts in particular—has steadily progressed [4, 27, 42, 32]. Starting with Sage [27] and continuing with Wadler and Findler [42], many languages have expressed the interactions between dynamic and simple static types and between simple static types and refinement types using a single syntactic form: the cast. Written $\langle T_1 \Rightarrow T_2 \rangle$, we read the cast form as "cast from type T_1 to type T_2".[1]

Casts are promising. They offer a unified view of changes in type information, have straightforward operational semantics, and enjoy a fruitful relationship with subtyping (see [42, 36, 26, 18, 38, 3]). Our language will derive its semantics from a full-spectrum language with casts. Before we can continue, we must explain how casts work: with type dynamic, with refinement types, and the most interesting part—between function types.

On the dynamic side, casts go into and out of ?, the dynamic type. A cast of the form $\langle \mathsf{Int} \Rightarrow ? \rangle\ 5$ asks for the number 5, which has type Int, to be treated

[1] There have been many different notations for casts, each with good and bad points. We use this one out of habit.

as a value of type ?, the dynamic type. The operational semantics of such a cast will mark the value with a tag, as in $5_{\mathsf{Int!}}$. Similarly, a cast of the form $\langle ? \Rightarrow \mathsf{Int} \rangle\ 5_{\mathsf{Int!}}$ will project the tagged integer out of type dynamic, yielding the original value 5. In the case where the cast's argument isn't tagged correctly, the cast must raise an error. Consider the term $\langle ? \Rightarrow \mathsf{Int} \rangle\ \mathsf{true}_{\mathsf{Bool!}}$. It tries to project an Int out of type dynamic, but the dynamic value is really a Bool—a type error. All we can reasonably do is abort the program, evaluating to the uncatchable exception fail.[2]

Casting between refinement types works similarly: $\langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x \neq 0\} \rangle\ 5$ must first check that $(x \neq 0)[5/x] \longrightarrow^* \mathsf{true}$, i.e., that the refinement's predicate is satisfied. If so, it will yield a tagged value, just like before: $5_{\{x:\mathsf{Int}|x\neq 0\}?}$. (The exclamation point on the Int! tag represents *injection* into type dynamic, while the question mark on the $\{x{:}\mathsf{Int} \mid x \neq 0\}?$ tag represents the successful *checking* of the predicate.) If the predicate should fail—returning false or failing some nested predicate check—then the check will return fail and the entire program will fail, as well.

Casts on functions are the most interesting: values with functional casts on them, like $\langle T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22} \rangle\ v_1$, are themselves values; they are *wrapped* with a *function proxy*. When such wrapped values are applied to a value v_2, the cast unfolds:

$$(\langle T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22} \rangle\ v_1)\ v_2 \longrightarrow$$
$$\langle T_{12} \Rightarrow T_{22} \rangle\ (v_1\ (\langle T_{21} \Rightarrow T_{11} \rangle\ v_2))$$

Note that this rule is contravariant in the domain.[3]

The casts in this paper (and its most closely related work) have runtime effects and are *not* erasable in general. For example, consider the term $\langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid \mathrm{prime}\ x\} \rangle$. The cast isn't erasable: it isn't in general decidable whether or not its argument will evaluate to a prime number, so a runtime check is necessary. This runtime checking isn't limited to refinement types: the cast in $\langle ? \Rightarrow \mathsf{Int} \rangle$ isn't erasable because we can't, in general, decide whether or not its argument will evaluate to a dynamic value with an Int! tag as opposed to, say, a Bool! tag. (Some casts *are* erasable, though; e.g., from a subtype to a supertype. *Upcast elimination* and other such optimizations have been the subject of some study already [26, 3].)

One problem common to calculi with casts is the problem of *space efficiency*. In particular, casts can accumulate in an unbounded way: redundant casts can grow the stack arbitrarily; casting functions can introduce an arbitrary number of function proxies. This unbounded growth of casts can, in the extreme, change the asymptotic complexity of programs. To see why the naïve treatment isn't space efficient, consider a mutually recursive definition of even and odd, adapted

[2] More realistic systems will make such exceptions catchable. To keep things simple, we ignore exception handling.

[3] A common alternative semantics is to introduce new lambdas, e.g., $\langle T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22} \rangle\ v \longrightarrow \lambda x{:}T_{21}.\ \langle T_{12} \Rightarrow T_{22} \rangle\ (v\ (\langle T_{21} \Rightarrow T_{11} \rangle\ x))$. Since our subject is space efficiency, introducing extra closures for explicit function proxies is a non-starter.

$$
\begin{array}{ll}
 & \mathsf{odd}\ 3 \\
\longrightarrow & (\langle ?\to ? \Rightarrow \mathsf{Int}\to\mathsf{Bool}\rangle\ \mathsf{even})\ 2 \\
\longrightarrow & \langle ? \Rightarrow \mathsf{Bool}\rangle\ (\mathsf{even}\ (\langle \mathsf{Int} \Rightarrow ?\rangle\ 2)) \\
\longrightarrow & \langle ? \Rightarrow \mathsf{Bool}\rangle\ (\mathsf{even}\ 2_{\mathsf{Int!}}) \\
\longrightarrow & \langle ? \Rightarrow \mathsf{Bool}\rangle\ ((\langle \mathsf{Int}\to\mathsf{Bool} \Rightarrow ?\to?\rangle\ (\lambda x{:}\mathsf{Int}.\ \ldots))\ 2_{\mathsf{Int!}}) \\
\longrightarrow & \langle ? \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow ?\rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)\ (\langle ? \Rightarrow \mathsf{Int}\rangle\ 2_{\mathsf{Int!}}))) \\
\longrightarrow & \langle ? \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow ?\rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)\ 2)) \\
\longrightarrow & \langle ? \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow ?\rangle\ (\mathsf{odd}\ 1)) \\
\longrightarrow & \langle ? \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow ?\rangle\ ((\langle ?\to ? \Rightarrow \mathsf{Int}\to\mathsf{Bool}\rangle\ \mathsf{even})\ 0)) \\
\longrightarrow & \langle ? \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow ?\rangle\ (\langle ? \Rightarrow \mathsf{Bool}\rangle\ (\mathsf{even}\ (\langle \mathsf{Int} \Rightarrow ?\rangle\ 0)))) \\
\longrightarrow & \langle ? \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow ?\rangle\ (\langle ? \Rightarrow \mathsf{Bool}\rangle\ (\mathsf{even}\ 0_{\mathsf{Int!}}))) \\
\longrightarrow & \langle ? \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow ?\rangle\ (\langle ? \Rightarrow \mathsf{Bool}\rangle \\
 & \quad ((\langle \mathsf{Int}\to\mathsf{Bool} \Rightarrow ?\to?\rangle\ (\lambda x{:}\mathsf{Int}.\ \ldots))\ 0_{\mathsf{Int!}}))) \\
\longrightarrow & \langle ? \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow ?\rangle\ (\langle ? \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow ?\rangle \\
 & \quad ((\lambda x{:}\mathsf{Int}.\ \ldots)\ (\langle ? \Rightarrow \mathsf{Int}\rangle\ 0_{\mathsf{Int!}}))))) \\
\longrightarrow & \langle ? \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow ?\rangle\ (\langle ? \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow ?\rangle \\
 & \quad ((\lambda x{:}\mathsf{Int}.\ \ldots)\ 0)))) \\
\longrightarrow & \langle ? \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow ?\rangle\ (\langle ? \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow ?\rangle\ \mathsf{true}))) \\
\longrightarrow & \langle ? \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow ?\rangle\ (\langle ? \Rightarrow \mathsf{Bool}\rangle\ \mathsf{true}_{\mathsf{Bool!}})) \\
\longrightarrow & \langle ? \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow ?\rangle\ \mathsf{true}) \\
\longrightarrow & \langle ? \Rightarrow \mathsf{Bool}\rangle\ \mathsf{true}_{\mathsf{Bool!}} \\
\longrightarrow & \mathsf{true}
\end{array}
$$

Figure 2: Space-inefficient reduction

from Herman et al. [21]:

$$
\begin{array}{l}
\mathsf{even} : ?\to? = \langle \mathsf{Int}\to\mathsf{Bool} \Rightarrow ?\to?\rangle\ \lambda x{:}\mathsf{Int}. \\
\quad \mathsf{if}\ x = 0\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{odd}\ (x-1) \\
\mathsf{odd} : \mathsf{Int}\to\mathsf{Bool} = \lambda x{:}\mathsf{Int}. \\
\quad \mathsf{if}\ x = 0\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ (\langle ?\to ? \Rightarrow \mathsf{Int}\to\mathsf{Bool}\rangle\ \mathsf{even})\ (x-1)
\end{array}
$$

In this example, even is written in a more dynamically typed style than odd. (While this example is contrived, it is easy to imagine mutually recursive modules with a mix of typing paradigms.) The cast $\langle \mathsf{Int}\to\mathsf{Bool} \Rightarrow ?\to?\rangle\ (\lambda x{:}\mathsf{Int}.\ \ldots)$ in the definition of even will (a) check that even's dynamically typed arguments are in fact Ints and (b) tag the resulting booleans into the dynamic type. The symmetric cast on even in the definition of odd serves to make the function even behave as if it were typed. This cast will ultimately cast the integer value $n-1$ into the dynamic type, ?, as well as projecting even's result out of type ? and into type Bool. Now consider the reduction sequence in Figure 2, observing how the number of coercions grows (redexes are highlighted).

While the operational semantics doesn't have an explicit stack, we can still see the accumulation of pending casts. What *were* tail-recursive calls are accumulating extra work in the continuation.[4] What's more, the work is redundant: we must tag and untag true twice. In short, casts have taken an algorithm that

[4]The call to even in the else branch of odd doesn't look like a tail call, but if the coercions

should use a constant amount of stack space and turned it into an algorithm that uses $O(n)$ stack space. Such a large space overhead is impractical: casts aren't space efficient.

Two of the existing approaches to space efficiency in the world of gradual typing [21, 36, 38] factor casts into their constituent *coercions*, adapting Henglein's system [20]. We will take a similar approach. For example, the cast $\langle ? \Rightarrow \mathsf{Bool} \rangle$, which checks that a dynamic value is a boolean and then untags it, is written as the coercion $\mathsf{Bool}?$; the cast $\langle \mathsf{Bool} \Rightarrow ? \rangle$, which tags a boolean into type dynamic, is written $\mathsf{Bool}!$. Most importantly, coercions can be composed, so $\langle \mathsf{Bool}? \rangle\ (\langle \mathsf{Bool}! \rangle\ e) \longrightarrow \langle \mathsf{Bool}!; \mathsf{Bool}? \rangle\ e$. Herman et al. normalize the coercion $\mathsf{Bool}!; \mathsf{Bool}?$ into the no-op coercion $\mathbf{Id}$. This normalization process is how they achieve space efficiency. For example:

$$\langle \mathsf{Bool}? \rangle\ (\langle \mathsf{Bool}! \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)\ (\langle \mathsf{Int}? \rangle\ 2_{\mathsf{Int}!}))) \longrightarrow \langle \mathbf{Id} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)\ (\langle \mathsf{Int}? \rangle\ 2_{\mathsf{Int}!}))$$

This composition and normalization of pending coercions allows them to prove a bound on the size of any coercion that occurs during the run of a given program. This bound effectively restores the possibility of tail-call optimization.

However, it isn't obvious how to extend Herman et al.'s [21] coercion system to refinement types. When do we test that values satisfy predicates? How do refinement type coercions normalize? We show that refinement types should have a checking coercion $\{x{:}T \mid e\}?$ and an (un)tagging coercion $\{x{:}T \mid e\}!$; the key insight for space efficiency is that the composition $\{x{:}T \mid e\}?; \{x{:}T \mid e\}!$ should normalize to $\mathbf{Id}$, i.e., checks that are immediately forgotten should be thrown away. Throwing away checks sounds dangerous, but the calculus is still sound—values typed at refinement types must satisfy their refinements. On the one hand, this is great news—space-efficiency is not only more practical, but there are fewer errors! On the other hand, the space-efficient semantics aren't *exactly* equivalent to the naïve semantics. Whether or not this is good news, these dropped checks are part and parcel of space efficiency. We develop this idea in detail in Section 5; we show that this means that space-efficient programs fail less often than their naïve counterparts in Section 6.

This paper makes several contributions, extending the existing solutions in a number of dimensions.

- We present a simplified approach to coercions that extends the earlier work to refinement types while condensing and clarifying the formulation (Section 5). In particular, earlier systems have given either term rewriting systems modulo equational theories, which lack straightfoward implementations [20, 21, 22, 36], or they have given complicated algorithms for merging casts [38]. We rework the definition of coercions to admit a straightforward term rewriting system, for which we enumerate the exact set of canonical coercions (as in [22], though we offer proof). We then define a straightforward merge operator on canonical coercions, offering a simpler and clearer theory.

are inserted automatically—as in Herman et al.—then it can be very difficult to tell what is and isn't a tail call. See related work (Section 9) for a discussion of cast insertion.

- We introduce what is, at present, the most expressive full-spectrum language, offering type dynamic as well as refinements of both base types and the dynamic type (Section 3 and Section 5). This new language narrowly edges out Wadler and Findler [42] by including refinements of type dynamic.
- We show that this language is space efficient, i.e., there are a bounded number of coercions in any program, and those coercions are bounded in size (Section 7).
- We also show that our new language is sound with respect to the naïve, inefficient semantics: if the naïve semantics reaches a value, so does the space-efficient one, but occasionally, the naïve semantics will fail when the space-efficient one succeeds (Section 6). Of the prior work, only Siek and Wadler [38] prove a similar soundness theorem, showing that their space-efficient gradual typing calculus is (exactly) equivalent to their original, naïve calculus.

Our coercion system elucidates some differences in the equational theory underlying casts between dynamic and simple types and between simple and refinement types.

Outline

Then we define a naïve, space-inefficient coercion calculus, Naive, with dynamic and refinement types (Section 3); in Section 5 we make the naïve calculus into a space-efficient one, which we call Efficient. In Section 6, we relate the two calculi, showing that the two are *mostly* observationally equivalent: if the naïve semantics reduces a term to a value, so will the space efficient one; sometimes the naïve semantics will diverge or produce a failure when the space-efficient one succeeds. We give formal justification for our claims of space efficiency in Section 7. Before concluding, we give a brief overview of our design philosophy in Section 8. Finally, we review related work in Section 9 and lay out future work in Section 10. In an appendix, we give a table of the merges of canonical coercions.

2 A cast calculus

In this section, we define Cast, a cast calculus with dynamic and refinement types. It is a small extension of Wadler and Findler's system [42].

Rule naming conventions

Before we begin our technical work in earnest, a word about conventions. We are defining three calculi: one with casts (Cast) and two with coercions (Naive and Efficient). The latter two largely share syntax and typing rules; all typing rules will be of the form T_Name. In general, we will rely on context to

Types and base types

$$\begin{array}{lcl} T & ::= & B \mid T_1 \to T_2 \mid \{x{:}B \mid e\} \mid ? \mid \{x{:}? \mid e\} \\ B & ::= & \mathsf{Bool} \mid \mathsf{Int} \mid \ldots \end{array}$$

Terms, results, values, and pre-values

$$\begin{array}{lcl} e & ::= & x \mid r \mid op(e_1, \ldots, e_n) \mid e_1\ e_2 \mid \langle T_1 \Rightarrow T_2 \rangle\ e \mid \langle \{x{:}T \mid e_1\}, e_2, v \rangle \\ r & ::= & v \mid \mathsf{fail} \\ v & ::= & u_{\mathbf{Id}} \mid v_{B!} \mid v_{\mathbf{Fun!}} \mid v_{\{x:T \mid e\}?} \mid v_{\langle T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22} \rangle} \\ u & ::= & k \mid \lambda x{:}T.\ e \end{array}$$

Typing contexts

$$\begin{array}{lcl} \Gamma & ::= & \emptyset \mid \Gamma, x{:}T \end{array}$$

Figure 3: Cast calculus syntax

differentiate terms. The evaluation rules for Cast, in this section, are named G_Name, using a subscripted arrow $\longrightarrow_c$. The evaluation rules for the naïve calculus, Naive, in Section 3 are named F_Name, using a subscripted arrow $\longrightarrow_n$ for the reduction relation; the space-efficient evaluation rules for Efficient are in Section 5 are named E_Name using a plain arrow $\longrightarrow$.

2.1 Syntax and typing

We give the syntax of Cast in Figure 3.

The typing rules for Cast are in Figure 4. Most of the rule are standard. We follow Wadler and Findler [42] in tracking completed casts with explicit tags—this approach plays particularly well with the coercion-based approach we take in the sequel.

2.2 Operational semantics

	$\{x{:}? \mid e\}$	$?$	$?\to?$	$T_1 \to T_2$	B	$\{x{:}B \mid e\}$
$\{x{:}? \mid e\}$	CID/(CPP, CPC, CC)	CPP	CPP, CFFB/CFF, CID	CPP, CFFB/(CFF, CFW)	CPP, CFFB	(CPC, CFFB)
$?$	CC	CID	CFFB/(CFF, CID)	CFFB/(CFF, CFW)	CBB/CBFB/CBFF	CPC, CBB/CBFB/CBFF
$?\to?$	CPC, CF, CC	CF	CFD, CFW, CF	CFW	not well typed	
$T_1 \to T_2$	CPC, CFD, CFW, CF, CC	CFD, CFW, CF	CFW	CFW	not well typed	
B	CPC, CB, CC	CB	not well typed		CID	CC
$\{x{:}B \mid e\}$	CPP, CPC, CB, CC	CPP, CB	not well typed		CPP	CID/(CPP, CPC, CC)

2.3 Proofs

2.1 Lemma [Canonical forms]: If $\Gamma \vdash v : T$, then:

- $T = ?$ implies that $v = v'_{B!}$ or $v = v'_{\mathbf{Fun!}}$ for some v'.
- $T = B$ implies that $v = k_{\mathbf{Id}}$.
- $T = T_1 \to T_2$ implies that $v = \lambda x{:}T_1.\ e_{\mathbf{Id}}$ or $v = v'_{\langle T'_1 \to T'_2 \Rightarrow T_1 \to T_2 \rangle}$ for some v'.

Well formed contexts and types $\boxed{\vdash \Gamma}$ $\boxed{\vdash T}$

$$\frac{}{\vdash \emptyset}\ \text{WF\_Empty} \qquad \frac{\vdash \Gamma \quad \vdash T}{\vdash \Gamma, x{:}T}\ \text{WF\_Extend}$$

$$\frac{}{\vdash B}\ \text{WF\_Base} \qquad \frac{}{\vdash ?}\ \text{WF\_Dyn} \qquad \frac{\vdash T_1 \quad \vdash T_2}{\vdash T_1 \to T_2}\ \text{WF\_Fun}$$

$$\frac{x{:}T \vdash e : \mathsf{Bool} \quad T = B \text{ or } T = ?}{\vdash \{x{:}T \mid e\}}\ \text{WF\_Refine}$$

Similar types $\boxed{\vdash T_1 \parallel T_2}$

$$\frac{T \neq T_1 \to T_2}{\vdash T \parallel T}\ \text{P\_Id} \qquad \frac{}{\vdash ? \parallel T}\ \text{P\_DynL} \qquad \frac{}{\vdash T \parallel ?}\ \text{P\_DynR}$$

$$\frac{\vdash T_{11} \parallel T_{21} \quad \vdash T_{12} \parallel T_{22}}{\vdash T_{11} \to T_{12} \parallel T_{21} \to T_{22}}\ \text{P\_Fun}$$

$$\frac{\vdash T_1 \parallel T_2}{\vdash \{x{:}T_1 \mid e\} \parallel T_2}\ \text{P\_RefineL} \qquad \frac{\vdash T_1 \parallel T_2}{\vdash T_1 \parallel \{x{:}T_2 \mid e\}}\ \text{P\_RefineR}$$

Well typed terms and values $\boxed{\Gamma \vdash u : T}$ $\boxed{\Gamma \vdash e : T}$

$$\frac{\vdash \Gamma}{\Gamma \vdash k : \mathsf{ty}(k)}\ \text{T\_Const} \qquad \frac{\vdash T_1 \quad \Gamma, x{:}T_1 \vdash e_{12} : T_2}{\Gamma \vdash \lambda x{:}T_1.\ e_{12} : T_1 \to T_2}\ \text{T\_Abs}$$

$$\frac{\Gamma \vdash u : T}{\Gamma \vdash u_{\mathbf{Id}} : T}\ \text{T\_PreVal} \qquad \frac{\Gamma \vdash v : B}{\Gamma \vdash v_{B!} : ?}\ \text{T\_TagB} \qquad \frac{\Gamma \vdash v : ? \to ?}{\Gamma \vdash v_{\mathbf{Fun}!} : ?}\ \text{T\_TagFun}$$

$$\frac{\vdash \Gamma \quad \emptyset \vdash v : T \quad \vdash \{x{:}T \mid e\} \quad e[v/x] \longrightarrow_c^* \mathsf{true}_{\mathbf{Id}}}{\Gamma \vdash v_{\{x:T \mid e\}?} : \{x{:}T \mid e\}}\ \text{T\_TagRefine}$$

$$\frac{\Gamma \vdash v : T_{11} \to T_{12} \quad \vdash T_{21} \to T_{22} \quad \vdash T_{11} \to T_{12} \parallel T_{21} \to T_{22}}{\Gamma \vdash v_{\langle T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22} \rangle} : T_{21} \to T_{22}}\ \text{T\_Wrap}$$

$$\frac{\vdash \Gamma \quad x{:}T \in \Gamma}{\Gamma \vdash x : T}\ \text{T\_Var} \qquad \frac{\vdash T \quad \vdash \Gamma}{\Gamma \vdash \mathsf{fail} : T}\ \text{T\_Fail}$$

$$\frac{\Gamma \vdash e : T_1 \quad \vdash T_2 \quad \vdash T_1 \parallel T_2}{\Gamma \vdash \langle T_1 \Rightarrow T_2 \rangle\ e : T_2}\ \text{T\_Cast} \qquad \frac{\Gamma \vdash e_1 : (T_1 \to T_2) \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}\ \text{T\_App}$$

$$\frac{\mathsf{ty}(op) = T_1 \to \ldots \to T_n \to T \quad \Gamma \vdash e_i : T_i}{\Gamma \vdash op(e_1, \ldots, e_n) : T}\ \text{T\_Op}$$

$$\frac{\vdash \Gamma \quad \vdash \{x{:}T \mid e_1\} \quad \emptyset \vdash v : T \quad \emptyset \vdash e_2 : \mathsf{Bool} \quad e_1[v/x] \longrightarrow_c^* e_2}{\Gamma \vdash \langle \{x{:}T \mid e_1\}, e_2, v \rangle : \{x{:}T \mid e_1\}}\ \text{T\_CheckCast}$$

Figure 4: Typing for Cast

$$\frac{}{(\lambda x{:}T.\ e_{12})_{\mathbf{Id}}\ v_2 \longrightarrow_c e_{12}[v_2/x]}\ \text{G\_Beta} \qquad \frac{}{op(v_1, \ldots, v_n) \longrightarrow_c [\![op]\!]\,(v_1, \ldots, v_n)}\ \text{G\_Op}$$

$$\frac{}{v_{1\,\langle T_{11}\to T_{12}\Rightarrow T_{21}\to T_{22}\rangle}\ v_2 \longrightarrow_c \langle T_{12}\Rightarrow T_{22}\rangle\ (v_1\ (\langle T_{21}\Rightarrow T_{11}\rangle\ v_2))}\ \text{G\_Fun}$$

$$\frac{}{\langle T \Rightarrow \{x{:}T \mid e\}\rangle\ v \longrightarrow_c \langle \{x{:}T \mid e\}, e[v/x], v\rangle}\ \text{G\_CastCheck}$$

$$\frac{}{\langle \{x{:}T \mid e\}, \mathsf{true}_{\mathbf{Id}}, v\rangle \longrightarrow_c v_{\{x:T|e\}?}}\ \text{G\_CheckOK} \qquad \frac{}{\langle \{x{:}T \mid e\}, \mathsf{false}_{\mathbf{Id}}, v\rangle \longrightarrow_c \mathsf{fail}}\ \text{G\_CheckFail}$$

$$\frac{T \neq T_1{\to}T_2}{\langle T \Rightarrow T\rangle\ v \longrightarrow_c v}\ \text{G\_CastId} \qquad \frac{}{\langle ? \Rightarrow T_1{\to}T_2\rangle\ v_{B!} \longrightarrow_c \mathsf{fail}}\ \text{G\_CastFunFailB}$$

$$\frac{}{\langle ? \Rightarrow T_1{\to}T_2\rangle\ v_{\mathbf{Fun}!} \longrightarrow_c \langle ?{\to}? \Rightarrow T_1{\to}T_2\rangle\ v}\ \text{G\_CastFunGuess} \qquad \frac{}{\langle ? \Rightarrow ?\rangle\ v \longrightarrow_c v_{B!}}\ \text{G\_CastB}$$

$$\frac{}{\langle ?{\to}? \Rightarrow ?\rangle\ v \longrightarrow_c v_{\mathbf{Fun}!}}\ \text{G\_CastFun} \qquad \frac{}{\langle ? \Rightarrow B\rangle\ v_{B!} \longrightarrow_c v}\ \text{G\_CastBB}$$

$$\frac{B \neq B'}{\langle ? \Rightarrow B\rangle\ v_{B'!} \longrightarrow_c \mathsf{fail}}\ \text{G\_CastBFailB} \qquad \frac{}{\langle ? \Rightarrow B\rangle\ v_{\mathbf{Fun}!} \longrightarrow_c \mathsf{fail}}\ \text{G\_CastBFailFun}$$

$$\frac{}{\langle T_{11}{\to}T_{12} \Rightarrow ?\rangle\ v \longrightarrow_c \langle ?{\to}? \Rightarrow ?\rangle\ (\langle T_{11}{\to}T_{12} \Rightarrow ?{\to}?\rangle\ v)}\ \text{G\_CastFunDyn} \qquad \frac{}{\langle T_{11}{\to}T_{12} \Rightarrow T_{21}{\to}T_{22}\rangle\ v \longrightarrow_c v_{\langle T_{11}\to T_{12}\Rightarrow T_{21}\to T_{22}\rangle}}\ \text{G\_CastFunWrap}$$

$$\frac{T_2 \neq \{x{:}T_1 \mid e\}}{\langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle\ v_{\{x:T_1|e\}?} \longrightarrow_c \langle T_1 \Rightarrow T_2\rangle\ v}\ \text{G\_CastPredPred} \qquad \frac{T_1 \neq T_2 \quad T_1 \neq \{x{:}T_1' \mid e'\}}{\langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle\ v \longrightarrow_c \langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle\ (\langle T_1 \Rightarrow T_2\rangle\ v)}\ \text{G\_CastPre}$$

$$\frac{e_1 \longrightarrow_c e_1'}{e_1\ e_2 \longrightarrow_c e_1'\ e_2}\ \text{G\_AppL} \qquad \frac{e_2 \longrightarrow_c e_2'}{v_1\ e_2 \longrightarrow_c v_1\ e_2'}\ \text{G\_AppR}$$

$$\frac{e_i \longrightarrow_c e_i'}{op(v_1, \ldots, v_{i-1}, e_i, \ldots, e_n) \longrightarrow_c op(v_1, \ldots, v_{i-1}, e_i', \ldots, e_n)}\ \text{G\_OpInner}$$

$$\frac{e \longrightarrow_c e'}{\langle T_1 \Rightarrow T_2\rangle\ e \longrightarrow_c \langle T_1 \Rightarrow T_2\rangle\ e'}\ \text{G\_CastInner}$$

$$\frac{e_2 \longrightarrow_c e_2'}{\langle \{x{:}T \mid e_1\}, e_2, v\rangle \longrightarrow_c \langle \{x{:}T \mid e_1\}, e_2', v\rangle}\ \text{G\_CheckInner}$$

$$\frac{}{\langle T_1 \Rightarrow T_2\rangle\ \mathsf{fail} \longrightarrow_c \mathsf{fail}}\ \text{G\_CastRaise} \qquad \frac{}{\mathsf{fail}\ e_2 \longrightarrow_c \mathsf{fail}}\ \text{G\_AppRaiseL} \qquad \frac{}{v_1\ \mathsf{fail} \longrightarrow_c \mathsf{fail}}\ \text{G\_AppRaiseR}$$

$$\frac{}{op(v_1, \ldots, v_{i-1}, \mathsf{fail}, \ldots, e_n) \longrightarrow_c \mathsf{fail}}\ \text{G\_OpRaise} \qquad \frac{}{\langle \{x{:}T \mid e\}, \mathsf{fail}, v\rangle \longrightarrow_c \mathsf{fail}}\ \text{G\_CheckRaise}$$

Figure 5: Cast operational semantics

- $T = \{x{:}T' \mid e\}$ implies that $v = v'_{\{x:T|e\}?}$ for some v'.

Proof: By case analysis on the typing derivation.

(T_Var) Contradictory—not a value.

(T_PreVal) The only types for pre-values are base and arrow types; $k_{\mathbf{Id}}$ fits the bill for $T = B$, and $\lambda x{:}T_1.\ e_{\mathbf{Id}}$ fits the bill for $T = T_1 \to T_2$.

(T_TagB) Fits the bill for $T = ?$; other types are contradictory.

(T_TagFun) Fits the bill for $T = ?$; other types are contradictory.

(T_TagRefine) Fits the bill for $T = \{x{:}T' \mid e\}$; other types are contradictory.

(T_Wrap) Fits the bill for $T = T_{21} \to T_{22}$; other types are contradictory.

(T_Cast) Contradictory—not a value.

(T_Fail) Contradictory—not a value.

(T_App) Contradictory—not a value.

(T_Op) Contradictory—not a value.

(T_CheckCast) Contradictory—not a value. □

2.2 Lemma [Progress]: If $\emptyset \vdash e : T$ then there exists an e' such that $e \longrightarrow_c e'$ or e is a result.

Proof: By induction on the typing derivation.

(T_PreVal) $u_{\mathbf{Id}}$ is a result.

(T_TagB) $v_{B!}$ is a result.

(T_TagFun) $v_{\mathbf{Fun}!}$ is a result.

(T_TagRefine) $v_{\{x:T|e\}?}$ is a result.

(T_Wrap) $v_{\langle T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22} \rangle}$ is a result.

(T_Var) Contradictory—variables aren't well typed in the empty context.

(T_Fail) fail is a result.

(T_Cast) By the IH on $\emptyset \vdash e : T_1$, either $e \longrightarrow_c e'$ or e is a result. In the former case, we step by G_CastInner. If e is a result, then it is either fail or a value v. In the former case, we step by G_CastRaise. Otherwise, we go by cases on T_1 and T_2.

($T_1 = ?$) By cases on T_2.

($T_2 = ?$) We step by G_CastId.

($T_2 = B$) By canonical forms (Lemma 2.1), $v = v'_{B'!}$ or $v = v'_{\mathbf{Fun}!}$. We step by G_CastBB (if $B = B'$), G_CastBFailB (if not), or G_CastBFailFun.

($T_2 = T_{21} \to T_{22}$) By canonical forms (Lemma 2.1), $v = v'_{B'!}$ or $v = v'_{\mathbf{Fun}!}$. We step by G_CastFunFailB or G_CastFunFun.

($T_2 = \{x{:}T \mid e\}$) If $T = ?$, we step by G_CastCheck. If $T = B$ (the only other option), we step by G_CastPreCheck.

($T_1 = B$) By cases on T_2.

($T_2 = ?$) We step by G_CastB.

($T_2 = B'$) By inversion of $\vdash B \parallel B'$, we have $B = B'$. We step by G_CastId.

($T_2 = T_{21} \to T_{22}$) Contradictory, since it is not the case that $\vdash B \parallel T_{21} \to T_{22}$.

($T_2 = \{x{:}T \mid e\}$) If $T = B'$ (and necessarily, $B' = B$), we step by G_CastCheck. Otherwise we step by G_CastPreCheck.

($T_1 = T_{11} \to T_{12}$) By cases on T_2.

($T_2 = ?$) We step by G_CastFun or G_CastFunDyn.

($T_2 = B$) Contradictory, since it is not the case that $\vdash T_{21} \to T_{22} \parallel B$.

($T_2 = T_{21} \to T_{22}$) We step by G_CastFunWrap.

($T_2 = \{x{:}T \mid e\}$) It must be that $T = ?$, since it is not the case that $\vdash T_{21} \to T_{22} \parallel B$. We step by G_CastPreCheck.

($T_1 = \{x{:}T \mid e\}$) By cases on T_2.

($T_2 = ?$) We step by G_CastPredPred.

($T_2 = B$) We step by G_CastPredPred.

($T_2 = T_{21} \to T_{22}$) We step by G_CastPredPred.

($T_2 = \{x{:}T' \mid e'\}$) We step by G_CastPredPred or G_CastId.

(T_App) We have $\emptyset \vdash e_1\ e_2 : T_2$. By the IH on $\emptyset \vdash e_1 : T_1 \to T_2$, either e_1 steps, or it is a result. In the former case, we go by G_AppL. In the latter, e_1 is either fail (and we step by G_AppRaiseL) or e_1 is a value.

Similarly, by the IH on $\emptyset \vdash e_2 : T_1$, either e_2 steps or is a result. We can apply G_AppR or G_AppRaiseR (using that e_1 is a value), unless e_2 is some value v_2. In that case, we use canonical forms to see that e_1 is either $\lambda x{:}T_1.\ e'_{1\,\mathbf{Id}}$ or $v_{1\,\langle T'_1 \to T'_2 \Rightarrow T_1 \to T_2 \rangle}$. We step by G_Beta and G_Fun, respectively.

(T_Op) By induction on n, applying the IH to step by either G_OpInner or G_OpFail. If all of the arguments are values, we step by G_Op.

(T_CheckCast) By the IH, we can either step the active check term by G_CastInner or G_CastRaise. If it's a value, we have $\emptyset \vdash v_2 : \mathsf{Bool}$, so v_2 is either $\mathsf{true}_{\mathbf{Id}}$ or $\mathsf{false}_{\mathbf{Id}}$. We step by G_CheckOK and G_CheckFail, respectively.

□

2.3 Lemma [Regularity]:

- If $\Gamma \vdash e : T$ then $\vdash \Gamma$ and $\vdash T$.
- If $\Gamma \vdash u : T$ then $\vdash \Gamma$ and $\vdash T$.

Proof: By mutual induction on the derivations.

(T_Var) $\vdash \Gamma$ by assumption, which gives us $\vdash T$.

(T_Const) $\vdash \Gamma$ By assumption, and we assume that $\vdash \mathsf{ty}(k)$.

(T_Abs) We have $\vdash T_1$ and $\Gamma, x{:}T_1 \vdash e_{12} : T_2$ by assumption. By the IH, $\vdash \Gamma, x{:}T_1$ and $\vdash T_2$. By inversion, $\vdash \Gamma$. We have $\vdash T_1 \to T_2$ by WF_Fun.

(T_PreVal) By the IH.

(T_TagB) By the IH and WF_Dyn.

(T_TagFun) By the IH and WF_Dyn.

(T_TagRefine) By the IH and assumption.

(T_Wrap) By the IH and assumption.

(T_Cast) By the IH and assumption.

(T_Fail) By assumption.

(T_App) By the IH.

(T_Op) By the IH and the assumption that operators have well formed types.

(T_CheckCast) By assumption.

□

2.4 Lemma [Substitution]:

2.5 Lemma [Preservation]: If $\emptyset \vdash e : T$ and $e \longrightarrow_c e'$, then $\emptyset \vdash e' : T$.

Proof: By induction on the evaluation derivation.

(G_Beta) By inversion, $x{:}T_1 \vdash e_1 : T_2$ and $\emptyset \vdash v_2 : T_1$. By substitution (Lemma 2.4).

(G_Fun) By inversion of T_Wrap, we know that $\vdash T_{11} \to T_{12} \parallel T_{21} \to T_{22}$. By T_Cast, T_App, and T_Cast; we find the similarities necessary for the T_Cast rules by inversion, since only P_Fun could have applied.

(G_Op) By assumption.

(G_CastId) Immediate.

(G_CastFunFailB) We have $\vdash T_1 \to T_2$ by inversion; we are done by T_Fail.

(G_CastFunFun) We have $\vdash ? \parallel T_i$ by P_DynL; by P_Fun, T_Cast, and assumption.

(G_CastB) By assumption and T_TagB.

(G_CastFun) By assumption and T_TagFun.

(G_CastBB) By assumption.

(G_CastBFailB) We have $\vdash B$ immediately; by T_Fail.

(G_CastBFailFun) We have $\vdash B$ immediately; by T_Fail.

(G_CastFunDyn) We use P_DynR, P_Fun in two applications of T_Cast.

(G_CastFunWrap) By T_Wrap.

(G_CastPredPred) By assumption and T_Cast, using $\vdash T_1 \parallel T_2$ from the inversion of $\vdash \{x{:}T_1 \mid e\} \parallel T_2$.

(G_CastPreCheck) By assumption and T_Cast. We use $\vdash T_1 \parallel T_2$ from the inversion of $\vdash T_1 \parallel \{x{:}T_2 \mid e\}$ in the first case and P_Id (it can't be a function type!) with P_RefineR in the second.

(G_CastCheck) By T_CheckCast, using $e[v/x] \longrightarrow_c^* e[v/x]$.

(G_CheckOK) By T_TagRefine, using $e[v/x] \longrightarrow_c^* \mathsf{true}_{\mathbf{Id}}$.

(G_CheckFail) By T_Fail, using the assumption that $\vdash \{x{:}T \mid e\}$.

(G_AppL) By T_App and the IH.

(G_AppR) By T_App and the IH.

(G_OpInner) By T_Op and the IH.

(G_CastInner) By T_Cast and the IH.

(G_CheckInner) By T_CheckCast and the IH, extending $e[v/x] \longrightarrow_c^* e_2 \longrightarrow_c e_2'$.

(G_AppRaiseL) By regularity (Lemma 2.3) and T_Fail.

(G_AppRaiseR) By regularity (Lemma 2.3) and T_Fail.

(G_OpRaise) By regularity (Lemma 2.3) and T_Fail.

(G_CastRaise) By inversion, $\vdash T_2$; then, by T_Fail.

(G_CheckRaise) By inversion, $\vdash \{x{:}T \mid e\}$; then, by T_Fail.

□

2.6 Theorem [Type soundness]: If $\emptyset \vdash e : T$, then either $e \longrightarrow^* r$ such that $\emptyset \vdash r : T$ or e diverges.

Proof: Using progress (Lemma 2.2) and preservation (Lemma 2.5). □

3 A naïve coercion calculus

In this section, we define a naïve coercion calculus, Naive. Its syntax is in Figure 6, its typing rules are in Figure 7, and its operational semantics are in Figure 9. To be truly complete, we ought to define a cast calculus, showing that it and the naïve calculus are observationally equivalent. To save space, we omit the definition of a cast calculus and a corresponding proof of observational equivalence with its corresponding naïve coercion calculus, opting to simple give the naïve coercion calculus directly. The relationship between a cast calculus and the naïve coercion calculus is straightforward, and establishing the relationship is not hard. (Siek and Wadler [38] establish a more interesting one, relating their space-efficient threesome casts and a coercion calculus.)

Our language adheres to a design philosophy of "simple types by default, dynamic and refinement types by coercion". We believe this is a novel philosophy of how to build full-spectrum languages. Our design philosophy has three principles. First, base values have simple types; e.g., all integers are typed at Int. Second, we give operations types precise enough to guarantee totality; e.g., division has a type at least as precise as $\mathsf{Int}{\to}\{x{:}\mathsf{Int} \mid x \neq 0\}{\to}\mathsf{Int}$. We understand refinement types as being designed for protecting partial operations (the original name is due to a method for protecting partial pattern matches [16]); giving operations types that make them total means that our reasoning about runtime errors can entirely revolve around cast (here, coercion) failures. And third, values satisfy their refinement types; e.g., if $\emptyset \vdash v : \{x{:}T \mid e\}$, then $e[v/x] \longrightarrow^* \mathsf{true}$. Every sound refinement type calculus has this, but ours will have it as an inversion of the typing rule. Since we are defining call-by-value (CBV) languages, this means that functions can assume their inputs actually satisfy their refinement types.

3.1 Syntax and typing

Most of the terms here are standard parts of the lambda calculus. The most pertinent extension here is the coercion term, $\langle c \rangle\ e$; we describe our language of coercions in greater detail below. Evaluation returns *results*: either a value or a failure fail. The term fail represents coercion failure. Coercion failures can occur when the predicate fails—i.e., $e_1[v/x] \longrightarrow_n^* \mathsf{false}_{\mathbf{Id}}$ (see F_CheckFail)—or when dynamically typed values don't match their type—e.g., $\langle \mathsf{Bool?} \rangle\ 5_{\mathsf{Int!}}$ (see F_TagFail). We treat fail as an uncatchable exception. Our values split in two parts: *pre-values* u are the typical values of other languages: constants and lambdas; *values* v are pre-values with a stack of primitive coercions. (See below for an explanation of the different kinds of coercions.) Technically, a value is either pre-values tagged with the identity coercion, $u_{\mathbf{Id}}$, or a value *tagged* with an extra coercion v_d. That is, in this language every value has a list of coercions. Values are introduced in source programs with the identity coercion, $u_{\mathbf{Id}}$. Keeping a coercion on *every* value is a slight departure from prior formulations. Doing so is technically expedient—simplifying the structure of the language and clearly differentiating terms with pending coercions and

Types and base types

$$
\begin{array}{lcl}
T & ::= & B \mid T_1 \to T_2 \mid \{x{:}B \mid e\} \mid ? \mid \{x{:}? \mid e\} \\
B & ::= & \mathsf{Bool} \mid \mathsf{Int} \mid \ldots
\end{array}
$$

Coercions, primitive coercions, and type tags

$$
\begin{array}{lcl}
c & ::= & d_1; \ldots; d_n \\
d & ::= & D! \mid D? \mid c_1 \mapsto c_2 \mid \mathbf{Fail} \\
D & ::= & B \mid \mathbf{Fun} \mid \{x{:}B \mid e\} \mid \{x{:}? \mid e\}
\end{array}
$$

Terms, results, values, and pre-values

$$
\begin{array}{lcl}
e & ::= & x \mid r \mid op(e_1, \ldots, e_n) \mid e_1\ e_2 \mid \langle c \rangle\ e \mid \\
 & & \langle \{x{:}T \mid e_1\}, e_2, v \rangle \\
r & ::= & v \mid \mathsf{fail} \\
v & ::= & u_{\mathbf{Id}} \mid v_{B!} \mid v_{\mathbf{Fun}!} \mid v_{\{x:T \mid e\}?} \mid v_{c_1 \mapsto c_2} \\
u & ::= & k \mid \lambda x{:}T.\ e
\end{array}
$$

Typing contexts

$$
\begin{array}{lcl}
\Gamma & ::= & \emptyset \mid \Gamma, x{:}T
\end{array}
$$

Figure 6: NAIVE syntax

values with tags.

The terms of our calculus are otherwise fairly unremarkable: we have variables, application, and a fixed set of built-in operations. We have two additional runtime terms. The *active check* $\langle \{x{:}T \mid e_1\}, e_2, v \rangle$ represents an ongoing check that the value v satisfies the predicate e_1; it is invariant that $e_1[v/x] \longrightarrow_n^* e_2$.

Our calculus has: simple types, where B is a base type and $T_1 \to T_2$ is the standard function type; the dynamic type ?; and refinements of both base types and type dynamic. The base refinement $\{x{:}B \mid e\}$ includes all constants k of type B such that $e[k_{\mathbf{Id}}/x] \longrightarrow_n^* \mathsf{true}_{\mathbf{Id}}$. Similarly, the dynamic refinement $\{x{:}? \mid e\}$ includes all *values* v such that v has type ? and $e[v/x] \longrightarrow_n^* \mathsf{true}_{\mathbf{Id}}$. We sometimes write $\{x{:}T \mid e\}$ when it doesn't matter whether the underlying type is ? or B. Notice that refinements of dynamic indirectly include refinements of functions. At the cost of having even more canonical coercions in Section 5.1, we could add refinements of functions. We omit them because they would have brought complexity without new insights. Going beyond refinements of functions, however, is challenging future work (see Section 10). When defining inference rules in this fashion (e.g., T_TAGVALREFINE), we treat such a rule as a rule schema that expands into two separate rules. We find it useful to think of three different "domains" of types: the base domain with the types B and $\{x{:}B \mid e\}$; the functional domain with the types of the form $T_1 \to T_2$, with special attention paid to functions on dynamic values, of type $? \to ?$; and the dynamic domain with the types ? and $\{x{:}? \mid e\}$.

Well formedness of types and contexts is defined straightforwardly in Figure 7. It is worth noting, however, that well formedness of refinements refers

Well formed contexts and types $\boxed{\vdash \Gamma}$ $\boxed{\vdash T}$

$$\frac{}{\vdash \emptyset}\ \text{WF\_Empty} \qquad \frac{\vdash \Gamma \quad \vdash T}{\vdash \Gamma, x{:}T}\ \text{WF\_Extend}$$

$$\frac{}{\vdash B}\ \text{WF\_Base} \quad \frac{}{\vdash ?}\ \text{WF\_Dyn} \quad \frac{\vdash T_1 \quad \vdash T_2}{\vdash T_1 \to T_2}\ \text{WF\_Fun} \quad \frac{x{:}T \vdash e : \mathsf{Bool} \quad T = B \text{ or } T = ?}{\vdash \{x{:}T \mid e\}}\ \text{WF\_Refine}$$

Well typed terms and values $\boxed{\Gamma \vdash u : T}$ $\boxed{\Gamma \vdash e : T}$

$$\frac{\vdash \Gamma}{\Gamma \vdash k : \mathsf{ty}(k)}\ \text{T\_Const} \qquad \frac{\vdash T_1 \quad \Gamma, x{:}T_1 \vdash e_{12} : T_2}{\Gamma \vdash \lambda x{:}T_1.\ e_{12} : T_1 \to T_2}\ \text{T\_Abs}$$

$$\frac{\Gamma \vdash u : T}{\Gamma \vdash u_{\mathbf{Id}} : T}\ \text{T\_PreVal} \qquad \frac{\Gamma \vdash v : T_1 \quad \vdash d : T_1 \rightsquigarrow T_2 \quad d \neq \{x{:}T \mid e\}?}{\Gamma \vdash v_d : T_2}\ \text{T\_TagVal}$$

$$\frac{\vdash \Gamma \quad \emptyset \vdash v : T \quad \vdash \{x{:}T \mid e\} \quad e[v/x] \longrightarrow_n^* \mathsf{true}_{\mathbf{Id}}}{\Gamma \vdash v_{\{x:T \mid e\}?} : T}\ \text{T\_TagValRefine}$$

$$\frac{\vdash \Gamma \quad x{:}T \in \Gamma}{\Gamma \vdash x : T}\ \text{T\_Var} \qquad \frac{\vdash T \quad \vdash \Gamma}{\Gamma \vdash \mathsf{fail} : T}\ \text{T\_Fail}$$

$$\frac{\vdash c : T_1 \rightsquigarrow T_2 \quad \Gamma \vdash e : T_1}{\Gamma \vdash \langle c \rangle\ e : T_2}\ \text{T\_Coerce}$$

$$\frac{\mathsf{ty}(op) = T_1 \to \ldots \to T_n \to T \quad \Gamma \vdash e_i : T_i}{\Gamma \vdash op(e_1, \ldots, e_n) : T}\ \text{T\_Op} \qquad \frac{\Gamma \vdash e_1 : (T_1 \to T_2) \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}\ \text{T\_App}$$

$$\frac{\vdash \Gamma \quad \vdash \{x{:}T \mid e_1\} \quad \emptyset \vdash v : T \quad \emptyset \vdash e_2 : \mathsf{Bool} \quad e_1[v/x] \longrightarrow_n^* e_2}{\Gamma \vdash \langle \{x{:}T \mid e_1\}, e_2, v \rangle : \{x{:}T \mid e_1\}}\ \text{T\_CheckNaive}$$

Well typed coercions $\boxed{\vdash c : T_1 \rightsquigarrow T_2}$ $\boxed{\vdash d : T_1 \rightsquigarrow T_2}$

$$\frac{\vdash T}{\vdash \mathbf{Id} : T \rightsquigarrow T}\ \text{C\_Id} \qquad \frac{\vdash d_1 : T_1 \rightsquigarrow T' \quad \vdash \ldots; d_n : T' \rightsquigarrow T_2}{\vdash d_1; \ldots; d_n : T_1 \rightsquigarrow T_2}\ \text{C\_Compose}$$

$$\frac{\vdash T_1 \quad \vdash T_2}{\vdash \mathbf{Fail} : T_1 \rightsquigarrow T_2}\ \text{C\_Fail}$$

$$\frac{}{\vdash B? : ? \rightsquigarrow B}\ \text{C\_BUntag} \qquad \frac{}{\vdash B! : B \rightsquigarrow ?}\ \text{C\_BTag}$$

$$\frac{}{\vdash \mathbf{Fun}? : ? \rightsquigarrow (? \to ?)}\ \text{C\_FunUntag}$$

$$\frac{\vdash c_1 : T_{21} \rightsquigarrow T_{11} \quad \vdash c_2 : T_{12} \rightsquigarrow T_{22}}{\vdash c_1 \mapsto c_2 : (T_{11} \to T_{12}) \rightsquigarrow (T_{21} \to T_{22})}\ \text{C\_Fun}$$

$$\frac{}{\vdash \mathbf{Fun}! : (? \to ?) \rightsquigarrow ?}\ \text{C\_FunTag} \qquad \frac{\vdash \{x{:}T \mid e\}}{\vdash \{x{:}T \mid e\}? : T \rightsquigarrow \{x{:}T \mid e\}}\ \text{C\_PredUntag}$$

back to the term typing judgment.

Term typing (also defined in Figure 7) is mostly standard. Readers should find the rules for constants (T_Const), variables (T_Var), functions (T_Abs), failure (T_Fail), application (T_App), and built-in operations (T_Op) familiar. It is worth taking a moment to comment on the type assignment functions $\mathsf{ty}(k)$ and $\mathsf{ty}(op)$ used in the T_Const and T_Op rules. In line with our philosophy (Section 8), the rule for constants gives them base types: $\mathsf{ty}(k) = B$. We require that no constants have, by default, type dynamic or a refinement type. By the same philosophy, the operator type assignment function $\mathsf{ty}(op)$ takes operations to first-order types which ensure totality. If $\mathsf{ty}(op) = T_1 \to \ldots \to T_n \to T$, then the operation's denotation $[\![op]\!]$ is a total function from $(T_1, \ldots, T_n)$ to T. If, for example, division is expressed as the operator div, then $\mathsf{ty}(\mathsf{div}) = \mathsf{Int} \to \{x{:}\mathsf{Int} \mid x \neq 0_{\mathbf{Id}}\} \to \mathsf{Int}$ or some similarly exact type. This property of operator types is critical: we believe that refinement types are designed to help programmers avoid failures of (fundamentally partial) primitive operations.

Pre-values are typed by T_Const and T_Abs; pre-values tagged with the **Id** coercion are typed as values by T_PreVal. T_TagVal types values that are tagged with anything but a refinement type, for which we use a separate rule. We want all values at a refined type to satisfy their refinement—a key property and part of our philosophy of refinement types. The T_TagValRefine rule ensures that values typed at a refinement type actually satisfy their refinement. In our metatheory, the typing rule for active check forms, T_CheckNaive, holds onto a trace of the evaluation of the predicate. If the check succeeds, the trace can then be put directly into a T_TagValRefine derivation. Naturally, none of these rule premises about evaluation are necessary for source programs—they are technicalities for our proofsof preservation (below) and equivalence (Section 6).

The coercions are the essence of this calculus: they represent the step-by-step checks that are done to move values between dynamic, simple, and refinement types. The syntax of coercions in Figure 6 splits coercions into three parts: composite coercions c, primitive coercions d, and tags D. The typing rules for coercions are written in Figure 7; the types of primitive coercions are shown graphically in Figure 8. When it is clear from context whether we mean a composite or a primitive coercion, we will simply call either a "coercion". A composite coercion c is simply a list of primitive coercions. We write the empty coercion—the composite coercion comprising zero primitive coercions—as **Id**. When we write $c; d$ or $d; c$ in a rule and it matches against a coercion with a single primitive coercion—that is, when we match $c; B!$ against $B!$—we let $c = \mathbf{Id}$. This is a slight departure from earlier coercion systems; this construction avoids messing around too much with re-association of coercion composition. We compare our coercions to other formulations in related work (Section 9). There are four kinds of primitive coercions: failures **Fail**, *tag* coercions $D!$, *checking* coercions $D?$, and *functional* coercions $c_1 \mapsto c_2$. (Note that c_1 and c_2 are composite.) Finally, the tags D are a flattening of the type space: each base type B has a corresponding tag (which we also write B); functions have a single tag **Fun**. Intuitively, these are the type tags that are commonly used in

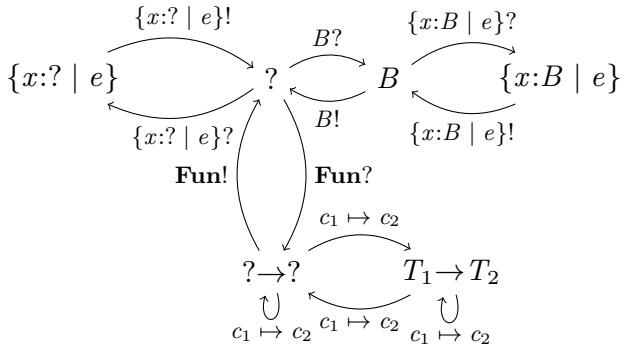

Figure 8: Primitive coercions

dynamically typed languages. We also have refinement tags for both types of refinement, which we write the same as the corresponding types: $\{x{:}B \mid e\}$ for refinements of base types and $\{x{:}? \mid e\}$ for refinements of type dynamic.

Failure coercions are present only for showing the equivalence with the space-efficient calculus; rule F_Fail gives a semantics for **Fail**. (We discuss the operational semantics more fully below, in Section 3.2.) It is worthwhile to contrast our treatment of failure with that of Herman et al. [21]. Whereas Henglein treats mismatched tag/untag operations, such as $B!;\mathbf{Fun}?$, as stuck, we follow Herman et al. [21] in having an explicit failure coercion, **Fail**, which leads to an uncatchable program failure, fail. The F_Fail rule causes the program to fail when a failure coercion appears. (We carefully keep **Fail** out of the tags placed on pre-values and values.) In fact, F_Fail will never apply when evaluating sensible source programs—no sane program will start with **Fail** in it, and no Naive evaluation rule generates **Fail**. Instead, failures arise in the Naive when the other rules with Fail in their name fire. We include F_Fail as a technicality for the soundness proof (Theorem 6.12) in Section 6. In Herman et al.'s calculus, $\langle\mathbf{Fail}\rangle\ v$ is a value—the program won't actually fail until this value reaches an elimination form. While systems with lazy error detection have been proposed [25], we say that $\langle\mathbf{Fail}\rangle\ e$ raises a program-terminating exception immediately. Eager error detecton is more in line with standard error behavior, particularly other calculi where failed casts result in blame [12, 14, 23, 37, 6, 19, 41, 42, 36, 18, 38, 3, 2, 9].

The *tagging* coercions $D!$ and *checking* coercions $D?$ fall into two groups: those which move values into type dynamic and those which deal with refinements (of either base types or type dynamic).

The tags B and **Fun** are used to move values to and from the dynamic type ?. The tagging coercions $B!$ and **Fun**! mark a base value (typed B) or functional value (typed ?→?) as having the dynamic type ?. The checking coercions $B?$ and **Fun**? are the corresponding *untagging* coercions, taking a dynamic value and checking its tag. If the tags match, the original typed value is returned: $\langle\mathsf{Bool}?\rangle\ \mathsf{true}_{B!} \longrightarrow_n \mathsf{true}_{\mathbf{Id}}$. If the tags don't match, the program

fails: $\langle \mathsf{Bool?} \rangle\ 5_{\mathsf{Int!}} \longrightarrow_n^* \mathsf{fail}$.

The tags $\{x{:}B \mid e\}$ and $\{x{:}? \mid e\}$ are used for refinements. The checking coercion $\{x{:}T \mid e\}?$ checks that a value v satisfies the predicate e, i.e., that $e[v/x] \longrightarrow_n \mathsf{true}_{\mathbf{Id}}$; see F_CheckOK and F_CheckFail below. The coercion $\{x{:}T \mid e\}!$ is correspondingly used to 'forget' refinement checks.

Note that in both the dynamic and the refinement cases, the checking coercions are the ones that might fail. Given our philosophy of values starting out simply typed, the two types of coercions differ in that tagging coercions are applied first when moving from simple types to dynamic typing, but checking coercions are applied first when moving to refinements. Any simply typed value is just fine as an appropriately tagged dynamic value, but a simply typed value must be checked to see if it satisfies a refinement.

Finally, while the **Fun**! and **Fun**? coercions injecting and project functions on ?→? into type ?, there is a separate structural coercion that works on typed functions: $c_1 \mapsto c_2$, typed by the rule C_Fun. Note that the C_Fun rule is contravariant; see the F_Fun rule below.

Only certain so-called *value coercions* can appear on values: **Id** appears on *all* values; $B!$ and **Fun**! tag values into the dynamic type; $\{x{:}T \mid e\}?$ marks successful refinement checks; and $c_1 \mapsto c_2$ wraps a value with pending checks, creating a function proxy. We'll revisit the notion of value coercions below, in our space efficient calculus in Section 5.

3.2 Operational semantics

Our rules are adapted from the evaluation contexts used in Herman et al. [21]. F_Beta and F_Op are totally standard CBV rules, as are most of the congruence and exception raising rules (F_AppL, F_AppR, F_OpInner, F_AppRaiseL, F_AppRaiseR, F_OpRaise). Before discussing the coercion evaluation rules, it is worth taking a moment to talk about the denotation of operators. In particular, $[\![op]\!]\,(v_1, \ldots, v_n)$ must (a) be total when applied to correctly typed values and (b) ignore the tags on its inputs. This disallows some potentially useful operators—e.g., projecting the tag from a dynamic value—but greatly simplifies the technicalities relating the two calculi in Section 6. We don't believe that adding such tag-dependent operators would break anything in a deep way, but omit them for simplicity's sake.

Most of the rules for coercions take a term of the form $\langle d; c \rangle\ v$ and somehow apply the primitive coercion d to v. The rest cover more structural uses of coercions. We cover these structural rules first and then explain the "tagging" rules. F_TagId applies when we have used up all of the primitive coercions, in which case we simply drop the coercion form.

The F_Merge and F_CoerceInner rules coordinate coercion merging and congruence. The F_Merge rule simply concatenates two adjacent coercions. This concatenation isn't space efficient—in the space-efficient calculus, we normalize the concatenation to a canonical coercion of bounded size. F_CoerceInner steps congruently inside a coerced term—we are careful to ensure that it can only apply after F_Merge has fired. Carefully staging F_CoerceInner after

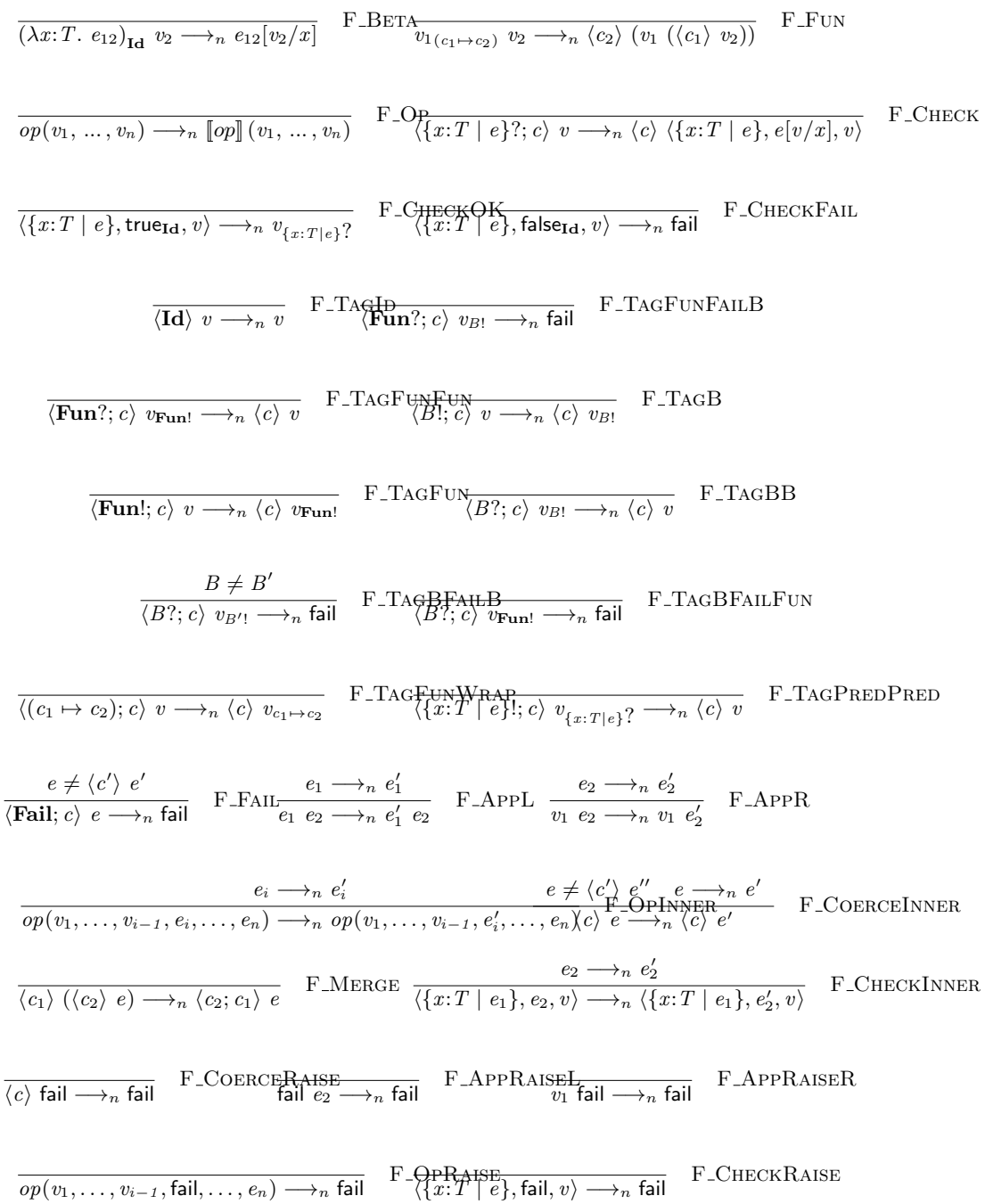

Figure 9: Naive operational semantics

F_Merge helps maintain determinism(Lemma 3.3)—if we didn't force F_Merge to apply first, the number of coercions might grow out of control.[5] Note that in F_Merge and F_Fail, the innermost term need not be a value. If we formulated our semantics as an abstract machine, we could have an explicit stack of coercions; instead, we combine them as they collide.

The remaining coercion rules have Tag in their name and work on a term $\langle d; c\rangle\ v$ by combining d and v. F_TagB and F_TagFun tag base values and functions (of type $?\to?$) into type dynamic, using the tagging coercions $B!$ and **Fun**!, respectively. F_TagBB and F_TagFunFun apply $B?$ and **Fun**? to values that have matching $B!$ and **Fun**! tags; the effect is to simply strip the tag off the value. The F_TagFunFailB, F_TagBFailFun, and F_TagBFailB rules cause the program to fail when it tries to strip a tag off with a checking coercion that doesn't match.

F_Check starts the *active check* for a refinement check. An active check $\langle\{x{:}T \mid e_1\}, e_2, v\rangle$ is a special kind of condition: if $e_2 \longrightarrow_n^* \mathsf{true}_{\mathbf{Id}}$, then it returns $v_{\{x:T|e_1\}?}$ (rule F_CheckOK); if $e_2 \longrightarrow_n^* \mathsf{false}_{\mathbf{Id}}$, then the active check returns fail (rule F_CheckFail). Note that the typing rules for active checks make sure that $e_1[v/x] \longrightarrow_n^* e_2$, i.e., that the active check is actually checking whether or not the value satisfies the predicate. The F_TagPredPred rule is similar the untagging rules F_TagBB and F_TagFunFun, though there is no chance of failure here. Having $\{x{:}T \mid e\}!$ eliminate the tag $\{x{:}T \mid e\}?$ is reminiscent of the coercion normalization rule given in the introduction—which we said occasionally skips checks. But here in the Naive, F_TagPredPred only applies when removing a tag from a value, i.e., when the check has already been done. In our space-efficient semantics in Section 5, our coercion normalization will actually skip checks.

The F_TagFunWrap rule wraps a value in a functional coercion. The F_Fun rule unwinds applications of wrapped values, coercing the wrapped function's argument and result.

Readers particularly familiar with contracts will recognize that these coercions are a lower level account of the steps taken in running casts (see Belo et al. [3] for an account); alternatively, these coercions are a lower level formulation of the projections underlying contracts [11]. See Greenberg et al. [18] for a comparison.

In Figure 10, we translate the cast example from the introduction (Figure 2). It is easy to see that this calculus isn't space efficient, either:, coercions can consume an unbounded amount of space. As the function evaluates, a stack of coercions builds up—here, proportional to the size of the input. Again, we highlight redexes; note that the whole term is highlighted when the outermost coercions merge. The casts of the earlier example match the coercions here, e.g. the cast $\langle ? \Rightarrow \mathsf{Bool}\rangle\ e$ is just like the coercion $\langle \mathsf{Bool}?\rangle\ e$.

[5] Siek and Wadler [38] pointed out that Herman et al.'s evaluation contexts introduce nondeterminism; our explicit congruence rules avoid this problem.

$$
\begin{array}{ll}
 & \mathsf{odd}\ 3_{\mathbf{Id}} \\
\longrightarrow_n & \mathsf{even}_{\mathsf{Int!}\mapsto\mathsf{Bool?}}\ 2_{\mathbf{Id}} \\
\longrightarrow_n & \langle \mathsf{Bool?} \rangle\ (\mathsf{even}\ (\langle \mathsf{Int!} \rangle\ 2_{\mathbf{Id}})) \\
\longrightarrow_n & \langle \mathsf{Bool?} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathbf{Id}})_{\mathsf{Int?}\mapsto\mathsf{Bool!}}\ (2_{\mathbf{Id}})_{\mathsf{Int!}}) \\
\longrightarrow_n & \langle \mathsf{Bool?} \rangle\ (\langle \mathsf{Bool!} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathbf{Id}}\ (\langle \mathsf{Int?} \rangle\ (2_{\mathbf{Id}})_{\mathsf{Int!}}))) \\
\longrightarrow_n & \langle \mathsf{Bool!}; \mathsf{Bool?} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathbf{Id}}\ (\langle \mathsf{Int?} \rangle\ (2_{\mathbf{Id}})_{\mathsf{Int!}})) \\
\longrightarrow_n & \langle \mathsf{Bool!}; \mathsf{Bool?} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathbf{Id}}\ 2_{\mathbf{Id}}) \\
\longrightarrow_n & \langle \mathsf{Bool!}; \mathsf{Bool?} \rangle\ (\mathsf{odd}\ 1_{\mathbf{Id}}) \\
\longrightarrow_n & \langle \mathsf{Bool!}; \mathsf{Bool?} \rangle\ (\mathsf{even}_{\mathsf{Int!}\mapsto\mathsf{Bool?}}\ 0_{\mathbf{Id}}) \\
\longrightarrow_n & \langle \mathsf{Bool!}; \mathsf{Bool?} \rangle\ (\langle \mathsf{Bool?} \rangle\ (\mathsf{even}\ (\langle \mathsf{Int!} \rangle\ 0_{\mathbf{Id}}))) \\
\longrightarrow_n & \langle \mathsf{Bool?}; \mathsf{Bool!}; \mathsf{Bool?} \rangle\ (\mathsf{even}\ (\langle \mathsf{Int!} \rangle\ 0_{\mathbf{Id}})) \\
\longrightarrow_n & \langle \mathsf{Bool?}; \mathsf{Bool!}; \mathsf{Bool?} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathbf{Id}})_{\mathsf{Int?}\mapsto\mathsf{Bool!}}\ (0_{\mathbf{Id}})_{\mathsf{Int!}}) \\
\longrightarrow_n & \langle \mathsf{Bool?}; \mathsf{Bool!}; \mathsf{Bool?} \rangle\ (\langle \mathsf{Bool!} \rangle \\
 & \quad ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathbf{Id}}\ (\langle \mathsf{Int?} \rangle\ (0_{\mathbf{Id}})_{\mathsf{Int!}}))) \\
\longrightarrow_n & \langle \mathsf{Bool!}; \mathsf{Bool?}; \mathsf{Bool!}; \mathsf{Bool?} \rangle \\
 & \quad ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathbf{Id}}\ (\langle \mathsf{Int?} \rangle\ (0_{\mathbf{Id}})_{\mathsf{Int!}})) \\
\longrightarrow_n & \langle \mathsf{Bool!}; \mathsf{Bool?}; \mathsf{Bool!}; \mathsf{Bool?} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathbf{Id}}\ 0_{\mathbf{Id}}) \\
\longrightarrow_n & \langle \mathsf{Bool!}; \mathsf{Bool?}; \mathsf{Bool!}; \mathsf{Bool?} \rangle\ \mathsf{true}_{\mathbf{Id}} \\
\longrightarrow_n & \langle \mathsf{Bool?}; \mathsf{Bool!}; \mathsf{Bool?} \rangle\ (\mathsf{true}_{\mathbf{Id}})_{\mathsf{Bool!}} \\
\longrightarrow_n & \langle \mathsf{Bool!}; \mathsf{Bool?} \rangle\ \mathsf{true}_{\mathbf{Id}} \\
\longrightarrow_n & \langle \mathsf{Bool?} \rangle\ (\mathsf{true}_{\mathbf{Id}})_{\mathsf{Bool!}} \\
\longrightarrow_n & \langle \mathbf{Id} \rangle\ \mathsf{true}_{\mathbf{Id}} \\
\longrightarrow_n & \mathsf{true}_{\mathbf{Id}}
\end{array}
$$

Figure 10: NAIVE reduction

3.3 Proofs

NAIVE is type sound, i.e, it enjoys progress and preservation.

3.1 Lemma [Regularity of coercion typing]:

- If $\vdash c : T_1 \rightsquigarrow T_2$ then $\vdash T_1$ and $\vdash T_2$.
- If $\vdash d : T_1 \rightsquigarrow T_2$ then $\vdash T_1$ and $\vdash T_2$.

Proof: By mutual induction on the typing derivations.

(C_ID) By inversion.

(C_FAIL) By inversion.

(C_COMPOSE) By the IH.

(C_BUNTAG) By WF_DYN on the left; WF_BASE on the right.

(C_BTAG) By WF_BASE on the left; by WF_DYN on the right.

(C_FUNUNTAG) By WF_DYN on the left; by WF_FUN with WF_DYN on the right.

(C_FUNTAG) By WF_FUN with WF_DYN on the left; by WF_DYN on the right.

(C_FUN) By the IH and WF_FUN.

(C_PREDBUNTAG) By inversion, we have $\vdash \{x{:}B \mid e\}$. We are done with WF_BASE.

(C_PREDBTAG) By inversion, we have $\vdash \{x{:}B \mid e\}$. We are done with WF_BASE.

(C_PREDDYNUNTAG) By inversion, we have $\vdash \{x{:}? \mid e\}$. We are done with WF_DYN.

(C_PREDBTAG) By inversion, we have $\vdash \{x{:}B \mid e\}$. We are done with WF_DYN.

□

3.2 Lemma [Regularity]: If $\Gamma \vdash e : T$ then $\vdash T$, and if $\Gamma \vdash u : T$ then $\vdash T$.

Proof: By induction on the typing derivation.

(T_CONST) By assumption.

(T_ABS) By the assumption, the IH, and WF_FUN.

(T_PREVAL) By the IH.

(T_TAGVAL) By regularity of coercion typing (Lemma 3.1).

(T_TAGVALREFINE) By assumption.

(T_VAR) By induction on $\vdash \Gamma$.

(T_FAIL) By assumption.

(T_COERCE) By regularity of coercion typing (Lemma 3.1).

(T_Op) By assumption on operation typing.

(T_App) By the IH

(T_CheckNaive) By assumption.

□

3.3 Lemma [Determinism]: If $e \longrightarrow_n e_1$ and $e \longrightarrow_n e_2$ then $e_1 = e_2$.

Proof: By induction on $e \longrightarrow_n e_1$, observing that in each case the same rule must have applied to find $e \longrightarrow_n e_2$. □

3.4 Lemma [Canonical forms]: If $\emptyset \vdash v : T$, then:

- If $T = \mathsf{Bool}$, then v is either $\mathsf{true}_{\mathbf{Id}}$ or $\mathsf{false}_{\mathbf{Id}}$.
- If $T = T_1 \to T_2$, then v is either $\lambda x{:}T_1.\ e_{12\,\mathbf{Id}}$ or ${v'}_{c_1 \mapsto c_2}$.
- If $T = ?$, then v is either ${v'}_{B!}$ or ${v'}_{\mathbf{Fun}!}$.
- If $T = \{x{:}T' \mid e\}$, then v is ${v'}_{\{x:T'|e\}?}$.

Proof: By induction on the typing derivation $\emptyset \vdash v : T$.

Proof:

(T_PreVal) Constants have base types, so we must only consider the case where $T = \mathsf{Bool}$; if $\mathsf{ty}(k) = \mathsf{Bool}$ then k is either true or false, and we are done.

(T_TagVal) $\emptyset \vdash {v'}_d : T_2$ and $\vdash d : T_1 \rightsquigarrow T_2$. It must be that d is one of $B!$ or $\mathbf{Fun}!$ or $c_1 \mapsto c_2$. In the first two cases, we fulfill the $T_2 = ?$ case. In the latter case, we fulfill the arrow case.

(T_TagValRefine) $\emptyset \vdash {v'}_{\{x:T|e\}?} : \{x{:}T \mid e\}$, which fulfills the refinement type case. □

3.5 Lemma [Progress]: If $\emptyset \vdash e : T$ then either e is a result or $e \longrightarrow_n e'$.

Proof: By induction on the typing derivation.

(T_PreVal) $u_{\mathbf{Id}}$ is a result.

(T_TagVal) v_d is a result.

(T_TagValRefine) $v_{\{x:T|e\}?}$ is a result.

(T_Var) Contradictory—no such typing derivation.

(T_Fail) fail is a result.

(T_Coerce) $\emptyset \vdash \langle c \rangle\ e : T_2$, where $\vdash c : T_1 \rightsquigarrow T_2$ and $\emptyset \vdash e : T_1$. We go by cases on the IH for e:

($e = v$) We go by cases on c.

($c = \mathbf{Id}$) We step by F_TagId.

($c = \mathbf{Fail}; c'$) We step by F_Fail.

($c = B!; c'$) We step by F_TagB.

($c = B?; c'$) By canonical forms, v is either ${v'}_{B'!}$ or ${v'}_{\mathbf{Fun}!}$. In the former case, we step by either F_TagBB or F_TagBFailB, depending on whether $B = B'$. In the latter case, we step by F_TagBFailFun.

($c = \mathbf{Fun}!; c'$) We step by F_TagFun.

($c = \mathbf{Fun}?; c'$) By canonical forms, v is either ${v'}_{B'!}$ or ${v'}_{\mathbf{Fun}!}$. In the former case, we step by F_TagFunFailB. In the latter case, we step by F_TagFunFun.

($c = \{x{:}T \mid e\}?; c'$) We step by F_Check.

($c = \{x{:}T \mid e\}!; c'$) By canonical forms, v must be of the form ${v'}_{\{x:T|e\}?}$, so we step by F_TagPredPred.

($c = (c_1 \mapsto c_2); c'$) We step by F_TagFunWrap.

($e = \mathsf{fail}$) We step by F_CoerceFail.

($e \longrightarrow_n e'$) If $e = \langle c' \rangle\ e'$, we ignore the step from the IH and instead step by F_Merge. Otherwise, we step by F_CoerceInner.

(T_Op) $\emptyset \vdash op(e_1, \ldots, e_n) : T$. We go by cases on the IH of each e_i, from left to right. If any of the e_i steps, we step by F_OpInner. If any of e_i is fail, we step by F_OpRaise. Finally, if all of the e_i are values, we step by F_Op.

(T_App) We have $\emptyset \vdash e_1\ e_2 : T_2$, where $\emptyset \vdash e_1 : T_1 \to T_2$ and $\emptyset \vdash e_2 : T_1$. We go by cases on IH for e_1.

($e_1 = v_1$) We go by cases on the IH for $\emptyset \vdash e_2 : T_1$.

($e_2 = v_2$) By canonical forms, v_1 is either $\lambda x{:}T_1.\ e_{12}{}_{\mathbf{Id}}$ or ${v'_1}_{c_1 \mapsto c_2}$. We step by F_Beta or F_Fun, respectively.

($e_2 = \mathsf{fail}$) We step by F_AppRaiseR.

($e_2 \longrightarrow_n e_2'$) We step by F_AppR.

($e_1 = \mathsf{fail}$) We step by F_AppRaiseL.

(otherwise) By the IH on $\emptyset \vdash e_1 : T_1 \to T_2$, we have $e_1 \longrightarrow_n e_1'$, and we step by F_AppL.

(T_CheckNaive) $\emptyset \vdash \langle \{x{:}T \mid e_1\}, e_2, v \rangle : \{x{:}T \mid e_1\}$. We go by cases on the IH for $\emptyset \vdash e_2 : \mathsf{Bool}$.

($e_2 = v_2$) By canonical forms, v_2 is either $\mathsf{true}_{\mathbf{Id}}$ or $\mathsf{false}_{\mathbf{Id}}$. We step by F_CheckOK or F_CheckFail, respectively.

($e_2 = \mathsf{fail}$) We step by F_CheckRaise.

($e_2 \longrightarrow_n e_2'$) We step by F_CheckInner.

□

3.6 Lemma [Substitution]: If $\Gamma_1, x{:}T, \Gamma_2 \vdash e : T'$ and $\emptyset \vdash v : T$ then $\Gamma_1, \Gamma_2 \vdash e[v/x] : T'$. Similarly, if $\Gamma_1, x{:}T, \Gamma_2 \vdash u : T'$ and $\emptyset \vdash v : T$ then $\Gamma_1, \Gamma_2 \vdash u[v/x] : T'$.

Proof: By induction on the typing derivation, leaving Γ_2 general.

(T_Const) By WF_Const.

(T_Abs) By WF_Abs and the IH.

(T_PreVal) By T_PreVal and the IH.

(T_TagVal) By T_TagVal and the IH.

(T_TagValRefine) By T_TagValRefine and the IH, noting that all terms involved are actually closed.

(T_Var) Either by assumption (if the two variables are the same) or by T_Var and weakening.

(T_Fail) By T_Fail.

(T_Coerce) By T_Coerce and the IH.

(T_Op) By T_Op and the IH.

(T_App) By T_App and the IH.

(T_CheckNaive) By T_CheckNaive, noting that all terms involved are actually closed.

□

3.7 Lemma [Preservation]: If $\emptyset \vdash e : T$ and $e \longrightarrow_n e'$ then $\emptyset \vdash e' : T$.

Proof: By induction on the typing derivation $\emptyset \vdash e : T$.

(T_PreVal) Contradictory—doesn't step.

(T_TagVal) Contradictory—doesn't step.

(T_TagValRefine) Contradictory—doesn't step.

(T_Var) Contradictory—there is no such derivation.

(T_Fail) Contradictory—doesn't step.

(T_Coerce) By cases on the step taken.

(F_Check) $\emptyset \vdash \langle \{x{:}T_1 \mid e\}?; c\rangle\ v : T_2$ and $\langle \{x{:}T_1 \mid e\}?; c\rangle\ v \longrightarrow_n \langle c\rangle\ \langle \{x{:}T_1 \mid e\}, e[v/x], v\rangle$. By inversion, $\vdash c : \{x{:}T_1 \mid e\} \rightsquigarrow T_2$. We are done by T_CheckNaive and T_Coerce.

(F_TagId) $\emptyset \vdash \langle \mathbf{Id} \rangle\ v : T$ and $\langle \mathbf{Id} \rangle\ v \longrightarrow_n v$. By assumption.

(F_TagFunFailB) $\emptyset \vdash \langle \mathbf{Fun}?; c \rangle\ v_{B!} : T$ and $\langle \mathbf{Fun}?; c \rangle\ v_{B!} \longrightarrow_n \mathsf{fail}$. By regularity of coercion typing (Lemma 3.1) we can apply T_Fail.

(F_TagFunFun) $\emptyset \vdash \langle \mathbf{Fun}?; c \rangle\ v_{\mathbf{Fun}!} : T$ and $\langle \mathbf{Fun}?; c \rangle\ v_{\mathbf{Fun}!} \longrightarrow_n \langle c \rangle\ v$. By inversion $\vdash c : (?\to?) \rightsquigarrow T$, and we are done by assumption and T_Coerce.

(F_TagB) $\emptyset \vdash \langle B!; c \rangle\ v : T$ and $\langle B!; c \rangle\ v \longrightarrow_n \langle c \rangle\ v_{B!}$. By inversion, $\vdash c : ? \rightsquigarrow T$ and $\emptyset \vdash v : B$. We finish by T_TagVal and T_Coerce.

(F_TagFun) $\emptyset \vdash \langle \mathbf{Fun}!; c \rangle\ v : T$ and $\langle \mathbf{Fun}!; c \rangle\ v \longrightarrow_n \langle c \rangle\ v_{\mathbf{Fun}!}$. By inversion, $\vdash c : ? \rightsquigarrow T$ and $\emptyset \vdash v : ?\to?$. We finish by T_TagVal and T_Coerce.

(F_TagBB) $\emptyset \vdash \langle B?; c \rangle\ v_{B!} : T$ and $\langle B?; c \rangle\ v_{B!} \longrightarrow_n \langle c \rangle\ v$. By inversion, $\vdash c : B \rightsquigarrow T$ and $\emptyset \vdash v : B$. We finish by T_Coerce.

(F_TagBFailB) $\emptyset \vdash \langle B?; c \rangle\ v_{B'!} : T$ and $\langle B?; c \rangle\ v_{B'!} \longrightarrow_n \mathsf{fail}$. By regularity of coercion typing (Lemma 3.1) and T_Fail.

(F_TagBFailFun) $\emptyset \vdash \langle B?; c \rangle\ v_{\mathbf{Fun}!} : T$ and $\langle B?; c \rangle\ v_{\mathbf{Fun}!} \longrightarrow_n \mathsf{fail}$. By regularity of coercion typing (Lemma 3.1) and T_Fail.

(F_TagFunWrap) $\emptyset \vdash \langle (c_1 \mapsto c_2); c \rangle\ v : T$ and $\langle (c_1 \mapsto c_2); c \rangle\ v \longrightarrow_n \langle c \rangle\ v_{c_1 \mapsto c_2}$. By inversion, $\vdash c : (T_{21}\to T_{22}) \rightsquigarrow T$ and $\emptyset \vdash v : T_{11}\to T_{12}$. By T_TagVal and T_Coerce.

(F_TagPredPred) $\emptyset \vdash \langle \{x{:}T_1 \mid e\}!; c \rangle\ v_{\{x:T_1|e\}?} : T_2$ and $\langle \{x{:}T_1 \mid e\}!; c \rangle\ v_{\{x:T_1|e\}?} \longrightarrow_n \langle c \rangle\ v$. By inversion, $\vdash c : T_1 \rightsquigarrow T_2$ and $\emptyset \vdash v : T_1$. By T_Coerce.

(F_Fail) $\emptyset \vdash \langle \mathbf{Fail}; c \rangle\ e : T$ and $\langle \mathbf{Fail}; c \rangle\ e \longrightarrow_n \mathsf{fail}$. By regularity of coercion typing (Lemma 3.1), $\vdash T$, so we are done by T_Fail.

(F_CoerceInner) $\emptyset \vdash \langle c \rangle\ e : T_2$ and $\langle c \rangle\ e \longrightarrow_n \langle c \rangle\ e'$, where $e \longrightarrow_n e'$. By inversion, $\vdash c : T_1 \rightsquigarrow T_2$ and $\emptyset \vdash e : T_1$. By the IH on $\emptyset \vdash e : T_1$, $\emptyset \vdash e' : T_1$. We are done by T_Coerce.

(F_Merge) $\emptyset \vdash \langle c_1 \rangle\ (\langle c_2 \rangle\ e) : T_3$ and $\langle c_1 \rangle\ (\langle c_2 \rangle\ e) \longrightarrow_n \langle c_2; c_1 \rangle\ e$. By inversion, $\vdash c_1 : T_2 \rightsquigarrow T_3$ and $\vdash c_2 : T_1 \rightsquigarrow T_2$ and $\emptyset \vdash e : T_1$. By C_Compose, $\vdash c_2; c_1 : T_1 \rightsquigarrow T_3$, so we are done by T_Coerce.

(F_CoerceRaise) $\emptyset \vdash \langle c \rangle\ \mathsf{fail} : T$ and $\langle c \rangle\ \mathsf{fail} \longrightarrow_n \mathsf{fail}$. By regularity of coercion typing (Lemma 3.1), we have $\vdash T$. We are done by T_Fail.

(T_Op) We go by cases on the step taken.

(F_Op) By assumption on the denotations of operations.

(F_OpInner) By the IH and T_OpInner.

(F_OpRaise) By regularity (Lemma 3.2) and T_Fail.

(T_App) We go by cases on the step taken.

(F_Beta) $\emptyset \vdash \lambda x{:}T_1.\ e_{12}\mathbf{_{Id}}\ v_2 : T_2$ and $\lambda x{:}T_1.\ e_{12}\mathbf{_{Id}}\ v_2 \longrightarrow_n e_{12}[v_2/x]$. By inversion, $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : T_1\to T_2$; by further inversion, $x{:}T_1 \vdash e_1 : T_2$. By substitution (Lemma 3.6), $\emptyset \vdash e_1[v_2/x] : T_{22}$.

(F_Fun) $\emptyset \vdash v_{1\,c_1 \mapsto c_2}\ v_2 : T_{22}$ and $v_{1\,c_1 \mapsto c_2}\ v_2 \longrightarrow_n \langle c_2 \rangle\ (v_1\ (\langle c_1 \rangle\ v_2))$. By inversion, $\emptyset \vdash v_{1\,c_1 \mapsto c_2} : T_{21} \to T_{22}$ and $\emptyset \vdash v_2 : T_{21}$; by further inversion, $\emptyset \vdash v_1 : T_{11} \to T_{12}$ and $\vdash c_1 : T_{21} \rightsquigarrow T_{11}$ and $\vdash c_2 : T_{12} \rightsquigarrow T_{22}$.
By T_Coerce, $\emptyset \vdash \langle c_1 \rangle\ v_2 : T_{11}$. By T_App, $\emptyset \vdash v_1\ (\langle c_1 \rangle\ v_2) : T_{12}$. By T_Coerce, $\emptyset \vdash \langle c_2 \rangle\ (v_1\ (\langle c_1 \rangle\ v_2)) : T_{22}$, and we are done.

(F_AppL) $\emptyset \vdash e_1\ e_2 : T_2$ and $e_1\ e_2 \longrightarrow_n e_1'\ e_2$ where $e_1 \longrightarrow_n e_1'$. By inversion, $\emptyset \vdash e_1 : T_1 \to T_2$ and $\emptyset \vdash e_2 : T_2$. By the IH on the former derivation, $\emptyset \vdash e_1' : T_1 \to T_2$, and we are done by T_App.

(F_AppR) $\emptyset \vdash v_1\ e_2 : T_2$ and $v_1\ e_2 \longrightarrow_n v_1\ e_2'$ where $e_2 \longrightarrow_n e_2'$. By inversion, $\emptyset \vdash v_1 : T_1 \to T_2$ and $\emptyset \vdash e_2 : T_2$. By the IH on the latter derivation, $\emptyset \vdash e_2' : T_2$, and we are done by T_App.

(F_AppRaiseL) $\emptyset \vdash \mathsf{fail}\ e_2 : T$. By regularity (Lemma 3.2), $\vdash T$, so we can apply T_Fail and be done.

(F_AppRaiseR) $\emptyset \vdash v_1\ \mathsf{fail} : T$. By regularity (Lemma 3.2), $\vdash T$, so we can apply T_Fail and be done.

(T_CheckNaive) We go by cases on the step taken.

(F_CheckOK) $\emptyset \vdash \langle \{x{:}T \mid e\}, \mathsf{true}_{\mathbf{Id}}, v \rangle : \{x{:}T \mid e\}$ and $\langle \{x{:}T \mid e\}, \mathsf{true}_{\mathbf{Id}}, v \rangle \longrightarrow_n v_{\{x:T|e\}?}$. By inversion, $\emptyset \vdash v : T$ and $e[v/x] \longrightarrow_n^* \mathsf{true}_{\mathbf{Id}}$, so we are done by T_TagValRefine.

(F_CheckFail) $\emptyset \vdash \langle \{x{:}T \mid e\}, \mathsf{false}_{\mathbf{Id}}, v \rangle : \{x{:}T \mid e\}$ and $\langle \{x{:}T \mid e\}, \mathsf{false}_{\mathbf{Id}}, v \rangle \longrightarrow_n \mathsf{fail}$. By inversion, $\vdash \{x{:}T \mid e\}$, and we are done by T_Fail.

(F_CheckInner) $\emptyset \vdash \langle \{x{:}T \mid e_1\}, e_2, v \rangle : \{x{:}T \mid e_1\}$ and $\langle \{x{:}T \mid e_1\}, e_2, v \rangle \longrightarrow_n \langle \{x{:}T \mid e_1\}, e_2', v \rangle$, where $e_2 \longrightarrow_n e_2'$. By inversion, we have $\vdash \{x{:}T \mid e_1\}$ and $\emptyset \vdash v : T$ and $\emptyset \vdash e_2 : \mathsf{Bool}$ and $e_1[v/x] \longrightarrow_n^* e_2$. We now have $e_1[v/x] \longrightarrow_n^* e_2'$, and $\emptyset \vdash e_2' : \mathsf{Bool}$ by the IH, so we are done by T_CheckNaive.

(F_CheckRaise) $\emptyset \vdash \langle \{x{:}T \mid e\}, \mathsf{fail}, v \rangle : \{x{:}T \mid e\}$ and $\langle \{x{:}T \mid e\}, \mathsf{fail}, v \rangle \longrightarrow_n \mathsf{fail}$. By inversion, $\vdash \{x{:}T \mid e\}$, so we can apply T_Fail and be done.

□

3.8 Theorem [Type soundness]: If $\emptyset \vdash e : T$ then either $e \longrightarrow_n^* r$ such that $\emptyset \vdash r : T$ or e diverges.

Proof: Using progress (Lemma 3.5) and preservation (Lemma 3.7). Naturally the proof is not constructive! □

Before concluding the section, we establish *coercion congruence*, a property of evaluation that will be critical in relating Naive and Efficient in Section 6. We say e_1 and e_2 coterminate when (a) e_1 diverges iff e_2 diverges, and (b) $e_1 \longrightarrow_n^* v$ iff $e_2 \longrightarrow_n^* v$. Cotermination is obviously reflexive, symmetric, and transitive.

When we say that e diverges, we mean that for all e' such that $e \longrightarrow_n^* e'$, there exists an e'' such that $e' \longrightarrow_n e''$. Combined with the fact that values don't step, we can see that this definition coincides with another standard definition of divergence: e never reduces to a value, i.e., for all e' such that $e \longrightarrow_n^* e'$, it is never the case that e' is a value.

3.9 Lemma: If $e \longrightarrow_n e'$ then e and e' coterminate.

Proof: By soundness (Theorem 3.8), either e diverges or reduces a result.

By assumption, e takes at least one step, so it isn't a value. So if $e \longrightarrow_n^* e''$ to diverge or reduce to a value, then $e \longrightarrow_n e' \longrightarrow_n^* e''$ by determinism (Lemma 3.3). So e' behaves the same way. □

3.10 Corollary: $\langle c_1 \rangle\ (\langle c_2 \rangle\ e)$ and $\langle c_2; c_1 \rangle\ e$ coterminate.

Proof: By Lemma 3.9 and F_Merge. □

3.11 Lemma [Coercion divergence congruence]: If e diverges then $\langle c \rangle\ e$ diverges.

Proof: It is never the case that $e \longrightarrow_n^* v$, and $\langle c \rangle\ e$ is never a value for any c or e. □

3.12 Lemma [Coercion congruence]: If $e \longrightarrow_n^* r$ then $\langle c \rangle\ r$ and $\langle c \rangle\ e$ coterminate.

Proof: We instead prove that there exists an e' such that $\langle c \rangle\ r \longrightarrow^* e'$ and $\langle c \rangle\ e \longrightarrow^* e'$, which implies coterminationby way of Lemma 3.9. We go by induction on $e \longrightarrow_n^* r$.

($r \longrightarrow_n^0 r$) Immediate.

($e \longrightarrow_n e' \longrightarrow_n^* r$) By cases on $e \longrightarrow_n e'$. The F_Beta, F_Fun, F_Op, F_CheckOK, F_CheckFail, F_AppL, F_AppR, F_OpInner, F_CheckInner, F_AppRaiseL, F_AppRaiseR, F_OpRaise, and F_CheckRaise all work by applying F_CoerceInner and the original rule to find $\langle c \rangle\ e \longrightarrow_n \langle c \rangle\ e'$, and then we are done by the IH, since $e \longrightarrow_n e'$ and $e \longrightarrow_n^* r$ implies $e' \longrightarrow_n^* r$ by determinism (Lemma 3.3).

The remaining cases must confront some degree of merging between coercions in e and the coercion c. The general thrust is to observe that if $e = \langle c' \rangle\ e \longrightarrow_n \langle c'' \rangle\ e'$, then $\langle c \rangle\ (\langle c' \rangle\ e) \longrightarrow_n \langle c'; c \rangle\ e \longrightarrow_n \langle c''; c \rangle\ e''$. We can then apply the IH on $\langle c'' \rangle\ e' \longrightarrow_n^* r$ to find an e'' such that $\langle c \rangle\ (\langle c'' \rangle\ e') \longrightarrow_n^* e''$ and $\langle c \rangle\ r \longrightarrow_n^* e''$. Finally, the former term steps by F_Merge to $\langle c''; c \rangle\ e'$, which must in turn reduce to e''—so we are done.

(F_TagId) $\langle \mathbf{Id} \rangle\ e \longrightarrow_n e$, so $\langle c \rangle\ (\langle \mathbf{Id} \rangle\ e) \longrightarrow_n \langle \mathbf{Id}; c \rangle\ e = \langle c \rangle\ e$. By the IH on $e \longrightarrow_n^* r$, we know that there exists an e' such that $\langle c \rangle\ e \longrightarrow_n^* e'$ and $\langle c \rangle\ r \longrightarrow_n^* e'$.

(F_TagFunFailB) $\langle \mathbf{Fun}?; c' \rangle\ v_{0\,B!} \longrightarrow_n \mathsf{fail}$, so $\langle c \rangle\ (\langle \mathbf{Fun}?; c' \rangle\ v_{0\,B!}) \longrightarrow_n \langle \mathbf{Fun}?; c'; c \rangle\ v_{0\,B!} \longrightarrow_n \mathsf{fail}$ by F_Merge and F_TagFunFailB. So $e' = \mathsf{fail}$, since $\langle c \rangle\ \mathsf{fail} \longrightarrow_n \mathsf{fail}$ by F_CoerceRaise.

(F_TagFunFun) $\langle \mathbf{Fun}?; c' \rangle\ v_{0\,\mathbf{Fun}!} \longrightarrow_n \langle c' \rangle\ v_0$, so $\langle c \rangle\ (\langle \mathbf{Fun}?; c' \rangle\ v_{0\,\mathbf{Fun}!}) \longrightarrow_n \langle \mathbf{Fun}?; c'; c \rangle\ v_{0\,\mathbf{Fun}!} \longrightarrow_n \langle c'; c \rangle\ v_0$ by F_Merge and F_TagFunFun. We know that $\langle c' \rangle\ v_0 \longrightarrow_n^* v$, and by the IH there exists an e' such that $\langle c \rangle\ (\langle c' \rangle\ v_0) \longrightarrow_n^* e'$ and $\langle c \rangle\ r \longrightarrow_n^* e'$. We conclude by applying F_Merge on the former term along with Corollary 3.10.

(F_TagB) $\langle B!; c' \rangle\ v_0 \longrightarrow_n \langle c \rangle\ v_{0\,B!}$, so $\langle c \rangle\ (\langle B!; c' \rangle\ v_0) \longrightarrow_n \langle B!; c'; c \rangle\ v_0 \longrightarrow_n \langle c'; c \rangle\ v_{0\,B!}$. Since $\langle c' \rangle\ v_{0\,B!} \longrightarrow_n^* v$, we know that there exists an e' such that $\langle c \rangle\ (\langle c' \rangle\ v_{0\,B!}) \longrightarrow_n^* e'$ and $\langle c \rangle\ r \longrightarrow_n^* e'$ by the IH. We conclude by applying F_Merge on the former term along with Corollary 3.10.

(F_TagFun) As for F_TagB.

(F_TagBB) As for F_TagFunFun.

(F_TagBFailB) As for F_TagFunFailB.

(F_TagBFailFun) As for F_TagBFailFun.

(F_TagFunWrap) $\langle (c_1 \mapsto c_2); c' \rangle\ v_0 \longrightarrow_n \langle c' \rangle\ v_{0\,c_1 \mapsto c_2}$, so $\langle c \rangle\ (\langle (c_1 \mapsto c_2); c' \rangle\ v_0) \longrightarrow_n \langle (c_1 \mapsto c_2); c'; c \rangle\ v_0 \longrightarrow_n \langle c'; c \rangle\ v_{0\,c_1 \mapsto c_2}$. Since $\langle c' \rangle\ v_{0\,c_1 \mapsto c_2} \longrightarrow_n^* v$, we know by the IH that there exists an e' such that $\langle c \rangle\ (\langle c' \rangle\ v_{0\,c_1 \mapsto c_2}) \longrightarrow_n^* e'$ and $\langle c \rangle\ r \longrightarrow_n^* e'$. We conclude by applying F_Merge on the former term along with Corollary 3.10.

(F_TagPredPred) As for F_TagFunFun and F_TagBB.

(F_Check) $\langle \{x{:}T \mid e\}?; c' \rangle\ v_0 \longrightarrow_n \langle c' \rangle\ \langle \{x{:}T \mid e\}, e[v_0/x], v_0 \rangle$, so $\langle c \rangle\ (\langle \{x{:}T \mid e\}?; c' \rangle\ v_0) \longrightarrow_n \langle \{x{:}T \mid e\}?; c'; c \rangle\ v_0 \longrightarrow_n \langle c'; c \rangle\ \langle \{x{:}T \mid e\}, e[v_0/x], v_0 \rangle$ by F_Merge and F_Check. Since $\langle c' \rangle\ \langle \{x{:}T \mid e\}, e[v_0/x], v_0 \rangle \longrightarrow_n^* r$, the IH gives us an e' such that $\langle c \rangle\ (\langle c' \rangle\ \langle \{x{:}T \mid e\}, e[v_0/x], v_0 \rangle) \longrightarrow_n^* e'$ and $\langle c \rangle\ r \longrightarrow_n^* e'$; we are done by applying F_Merge and Corollary 3.10 on the left.

(F_CoerceInner) $\langle c' \rangle\ e \longrightarrow_n \langle c' \rangle\ e'$, so $\langle c \rangle\ (\langle c' \rangle\ e) \longrightarrow_n \langle c'; c \rangle\ e \longrightarrow_n \langle c'; c \rangle\ e'$. Since $\langle c' \rangle\ e' \longrightarrow_n^* r$, we know by the IH that there exists an e'' such that $\langle c \rangle\ (\langle c' \rangle\ e') \longrightarrow_n^* e''$. This last steps immediately by F_Merge to $\langle c'; c \rangle\ e'$, so we know that term goes to e''', too.

(F_Merge) $\langle c_1 \rangle\ (\langle c_2 \rangle\ e) \longrightarrow_n \langle c_2; c_1 \rangle\ e$, so $\langle c \rangle\ (\langle c_1 \rangle\ (\langle c_2 \rangle\ e)) \longrightarrow_n \langle c_1; c \rangle\ (\langle c_2 \rangle\ e) \longrightarrow_n \langle c_2; c_1; c \rangle\ e$. We know that $\langle c_2; c_1 \rangle\ e \longrightarrow_n^* r$, so the IH gives us an e' such that $\langle c \rangle\ (\langle c_2; c_1 \rangle\ e) \longrightarrow_n^* e'$ and $\langle c \rangle\ r \longrightarrow_n^* e'$. But the former term steps by F_Merge to $\langle c_2; c_1; c \rangle\ e$, so we are done by Corollary 3.10.

(F_Fail) $\langle \mathbf{Fail}; c' \rangle\ e \longrightarrow_n \mathsf{fail}$, so $\langle c \rangle\ (\langle \mathbf{Fail}; c' \rangle\ e) \longrightarrow_n \langle \mathbf{Fail}; c'; c \rangle\ e \longrightarrow_n \mathsf{fail}$. We have $e' = r = \mathsf{fail}$, since $\langle c \rangle\ \mathsf{fail} \longrightarrow_n^* \mathsf{fail}$ by F_CoerceRaise.

(F_CoerceRaise) $\langle c' \rangle\ \mathsf{fail} \longrightarrow_n \mathsf{fail}$, so $\langle c \rangle\ (\langle c' \rangle\ \mathsf{fail}) \longrightarrow_n \langle c'; c \rangle\ \mathsf{fail} \longrightarrow_n \mathsf{fail}$, and again $e' = r = \mathsf{fail}$ as in F_Fail, so we are done.

□

Value rules

$$\begin{array}{rcl}
\forall j.\ k_{\mathbf{Id}} \sim^j k_{\mathbf{Id}} : B & \iff & \mathsf{ty}(k) = B \\
v_{11} \sim^j v_{21} : T_1 \to T_2 & \iff & \\
\forall m < j.\ \forall v_{12} \sim^m v_{22} : T_1.\ v_{11}\ v_{12} \simeq^m v_{21}\ v_{22} : T_2 &&
\end{array}$$

$$\begin{array}{rcl}
v_{1\,B!} \sim^j u_{2\,c \Downarrow B!} : ? & \iff & v_1 \sim^j u_{2\,c} : B \\
v_{1\,\mathbf{Fun!}} \sim^j u_{2\,c \Downarrow \mathbf{Fun!}} : ? & \iff & v_1 \sim^j u_{2\,c} : ? \to ?
\end{array}$$

$$\begin{array}{c}
v_{1\,\{x:T \mid e_1\}?} \sim^j u_{c \Downarrow \{x:T \mid e_2\}?} : \{x{:}T \mid e_1\} \\
\iff \\
\forall m < j.\ v_1 \sim^m u_{2\,c} : T \wedge \{x{:}T \mid e_1\} \sim^m \{x{:}T \mid e_2\}
\end{array}$$

Term rules

$$\begin{array}{l}
e_1 \simeq^j e_2 : T \iff \\
e_1 \text{ diverges} \vee \\
\forall m < j. \quad e_1 \longrightarrow_c^m \mathsf{fail} \implies e_2 \longrightarrow_c^* \mathsf{fail} \\
\qquad \wedge \quad e_1 \longrightarrow_c^m v_1 \implies e_2 \longrightarrow_c^* v_2 \wedge v_1 \sim^{(j-m)} v_2 : T
\end{array}$$

Type rules

$$\begin{array}{rcl}
B \sim^j B & & ? \sim^j ? \\
T_{11} \to T_{12} \sim^j T_{21} \to T_{22} & \iff & T_{11} \sim^j T_{21} \wedge T_{12} \sim^j T_{22} \\
\{x{:}T \mid e_1\} \sim^j \{x{:}T \mid e_2\} & \iff & \\
\multicolumn{3}{c}{\forall m < j.\ \forall v_1 \sim^m v_2 : T.\ e_1[v_1/x] \simeq^m e_2[v_2/x] : \mathsf{Bool}}
\end{array}$$

Closing substitutions

$$\begin{array}{rcl}
\Gamma \models^j \delta & \iff & \forall x \in \mathrm{dom}(\Gamma).\ \delta_1(x) \sim^j \delta_2(x) : T \\
\Gamma \vdash e_1 \simeq e_2 : T & \iff & \forall j \geq 0.\ \forall \Gamma \models^j \delta.\ \delta_1(e_1) \simeq^j \delta_2(e_2) : T
\end{array}$$

Figure 11: Relating the Cast and Naive

4 Soundness of Naive with regard to Cast

4.1 Lemma [Preservation for coerce]: Assuming that no refinement tags or active checking forms are present:

1. If $\Gamma \vdash e : T$ then $\mathsf{coerce}\,(\Gamma) \vdash \mathsf{coerce}\,(e) : \mathsf{coerce}\,(T)$.
2. If $\Gamma \vdash u : T$ then $\mathsf{coerce}\,(\Gamma) \vdash \mathsf{coerce}\,(u) : \mathsf{coerce}\,(T)$.
3. If $\vdash T_1$ and $\vdash T_2$ and $\vdash T_1 \parallel T_2$ then $\vdash \mathsf{coerce}(T_1, T_2) : \mathsf{coerce}\,(T_1) \rightsquigarrow \mathsf{coerce}\,(T_2)$.
4. If $\vdash T$ then $\vdash \mathsf{coerce}\,(T)$.
5. If $\vdash \Gamma$ then $\vdash \mathsf{coerce}\,(\Gamma)$.

Proof: By simultaneous induction on the typing/well formedness/similarity derivations.

Translating casts

$$
\begin{array}{rcl}
\mathsf{coerce}(T, T) & = & \mathbf{Id} \\
 & & \text{when } T \neq T_1 \to T_2 \\
\mathsf{coerce}(?, T_1 \to T_2) & = & \mathbf{Fun}?; \mathsf{coerce}(? \to ?, T_1 \to T_2) \\
\mathsf{coerce}(?, B) & = & B? \\
\mathsf{coerce}(B, ?) & = & B! \\
\mathsf{coerce}(T_1 \to T_2, ?) & = & \mathsf{coerce}(T_1 \to T_2, ? \to ?); \mathbf{Fun}! \\
\mathsf{coerce}(T_{11} \to T_{12}, T_{21} \to T_{22}) & = & \mathsf{coerce}(T_{21}, T_{11}) \mapsto \mathsf{coerce}(T_{12}, T_{22}) \\
\mathsf{coerce}(\{x{:}T_1 \mid e\}, T_2) & = & \{x{:}T_1 \mid e\}!; \mathsf{coerce}(T_1, T_2) \\
 & & \text{when } T_2 \neq \{x{:}T_1 \mid e\} \\
\mathsf{coerce}(T_1, \{x{:}T_2 \mid e\}) & = & \mathsf{coerce}(T_1, T_2); \{x{:}T_2 \mid e\}? \\
 & & \text{when } T_1 \neq T_2 \text{ and } T_1 \neq \{x{:}T_1' \mid e'\}
\end{array}
$$

Translating pre-values and terms

$$
\begin{array}{rcl}
\mathsf{coerce}\,(k) & = & k \\
\mathsf{coerce}\,(\lambda x{:}T.\ e) & = & \lambda x{:}\mathsf{coerce}\,(T).\ \mathsf{coerce}\,(e) \\
 & & \\
\mathsf{coerce}\,(x) & = & x \\
\mathsf{coerce}\,(u_{\mathbf{Id}}) & = & \mathsf{coerce}\,(u)_{\mathbf{Id}} \\
\mathsf{coerce}\,(v_{B!}) & = & \mathsf{coerce}\,(v)_{B!} \\
\mathsf{coerce}\,(v_{\mathbf{Fun}!}) & = & \mathsf{coerce}\,(v)_{\mathbf{Fun}!} \\
\mathsf{coerce}\,(v_{\langle T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22} \rangle}) & = & \mathsf{coerce}\,(v)_{(\mathsf{coerce}(T_{21}, T_{11}) \mapsto \mathsf{coerce}(T_{12}, T_{22}))} \\
\mathsf{coerce}\,(v_{\{x:T \mid e\}?}) & = & \mathsf{coerce}\,(v)_{\{x:T \mid \mathsf{coerce}\,(e)\}?} \\
\mathsf{coerce}\,(\langle T_1 \Rightarrow T_2 \rangle\ e) & = & \langle \mathsf{coerce}(T_1, T_2) \rangle\ \mathsf{coerce}\,(e) \\
\mathsf{coerce}\,(e_1\ e_2) & = & \mathsf{coerce}\,(e_1)\ \mathsf{coerce}\,(e_2) \\
\mathsf{coerce}\,(op(e_1, \ldots, e_n)) & = & op(\mathsf{coerce}\,(e_1), \ldots, \mathsf{coerce}\,(e_n)) \\
\mathsf{coerce}\,(\langle \{x{:}T \mid e_1\}, e_2, v \rangle) & = & \langle \mathsf{coerce}\,(\{x{:}T \mid e_1\}), \mathsf{coerce}\,(e_2), \mathsf{coerce}\,(v) \rangle
\end{array}
$$

Translating types and contexts

$$
\begin{array}{rcl}
\mathsf{coerce}\,(?) & = & ? \\
\mathsf{coerce}\,(B) & = & B \\
\mathsf{coerce}\,(T_1 \to T_2) & = & \mathsf{coerce}\,(T_1) \to \mathsf{coerce}\,(T_2) \\
\mathsf{coerce}\,(\{x{:}T \mid e\}) & = & \{x{:}T \mid \mathsf{coerce}\,(e)\} \\
 & & \\
\mathsf{coerce}\,(\emptyset) & = & \emptyset \\
\mathsf{coerce}\,(\Gamma, x{:}T) & = & \mathsf{coerce}\,(\Gamma), x{:}\mathsf{coerce}\,(T)
\end{array}
$$

Figure 12: Translating from Cast to Naive

(WF_Empty) By WF_Empty.

(WF_Extend) By WF_Extend, using IHs (4) and (6).

(WF_Dyn) By WF_Dyn.

(WF_Base) By WF_Base.

(WF_Fun) By WF_Fun and IH (4).

(WF_Refine) By WF_Refine and IH (4).

(T_Const) By WF_Const and IH (6).

(T_Abs) By WF_Abs and IHs (4) and (4).

(T_Var) By WF_Var and IH (6).

(T_PreVal) By WF_PreVal and IH (3).

(T_TagB) By IH (4), $\mathsf{coerce}\,(\Gamma) \vdash \mathsf{coerce}\,(v) : B$. By C_BTag, $\vdash B! : B \rightsquigarrow ?$. We are done by T_TagVal.

(T_TagFun) By IH (4), $\mathsf{coerce}\,(\Gamma) \vdash \mathsf{coerce}\,(v) : ? \to ?$. By C_FunTag, $\vdash \mathbf{Fun!} : (? \to ?) \rightsquigarrow ?$. We are done by T_TagVal.

(T_TagRefine) Contradiction—we assumed that refinement tags do not appear.

(T_Wrap) By IH (4), $\mathsf{coerce}\,(\Gamma) \vdash \mathsf{coerce}\,(v) : \mathsf{coerce}\,(T_{11} \to T_{12})$. By Naive regularity (Lemma 3.2) and IH (4), we have $\vdash \mathsf{coerce}\,(T_{11} \to T_{12})$ and $\vdash \mathsf{coerce}\,(T_{21} \to T_{22})$. Since we also have $\vdash T_{11} \to T_{12} \parallel T_{21} \to T_{22}$, we can apply IH (3) to find that

$$\vdash \mathsf{coerce}(T_{11} \to T_{12},\, T_{21} \to T_{22}) : \mathsf{coerce}\,(T_{11} \to T_{12}) \rightsquigarrow \mathsf{coerce}\,(T_{21} \to T_{22})$$

Recall that $\mathsf{coerce}(T_{11} \to T_{12},\, T_{21} \to T_{22}) = \mathsf{coerce}(T_{21},\, T_{11}) \mapsto \mathsf{coerce}(T_{12},\, T_{22})$. So by WF_TagVal, we are done.

(T_Fail) By WF_Fail and IHs (4) and (6).

(T_Cast) By WF_Coerce, Naive regularity (Lemma 3.2), and IHs (4), (4), and (3).

(T_App) By WF_App and IH (4).

(T_Op) By WF_Op and IH (4).

(T_CheckCast) Contradictory—we assumed that there were no active checks.

(P_Id) Since T can't be a function type, $\mathsf{coerce}(T, T) = \mathbf{Id}$, and we are done by C_Id.

(P_DynL) By C_Compose and IH (3), using C_BUntag, C_FunUntag, or C_Compose and IH (3) with C_PredUntag and IH (4), depending on T.

Types

$$\begin{array}{lcl} T & ::= & B \mid T_1 \to T_2 \mid \{x{:}B \mid e\} \mid ? \mid \{x{:}? \mid e\} \\ B & ::= & \mathsf{Bool} \mid \mathsf{Int} \mid \ldots \end{array}$$

Coercions, primitive coercions, and type tags

$$\begin{array}{lcl} c & ::= & d_1; \ldots; d_n \mid \mathbf{Fail} \\ d & ::= & D! \mid D? \mid c_1 \mapsto c_2 \\ D & ::= & B \mid \mathbf{Fun} \mid \{x{:}B \mid e\} \mid \{x{:}? \mid e\} \end{array}$$

Terms, values, pre-values, and results

$$\begin{array}{lcl} e & ::= & x \mid r \mid op(e_1, \ldots, e_n) \mid e_1\ e_2 \mid \langle c \rangle\ e \mid \\ & & \langle \{x{:}T \mid e_1\}, e_2, v \rangle \\ r & ::= & v \mid \mathsf{fail} \\ v & ::= & u_c \\ u & ::= & k \mid \lambda x{:}T.\ e \end{array}$$

Typing contexts

$$\Gamma ::= \emptyset \mid \Gamma, x{:}T$$

Figure 13: Updated syntax for Efficient

$$\frac{\Gamma \vdash u : T_1 \quad \vdash c : T_1 \rightsquigarrow T_2 \quad c \neq \mathbf{Fail} \quad c \neq c'; \{x{:}T \mid e\}?}{\Gamma \vdash u_c : T_2} \ \text{T\_Val}$$

$$\frac{\vdash \Gamma \quad \emptyset \vdash u_c : T \quad \vdash \{x{:}T \mid e\}? : T \rightsquigarrow \{x{:}T \mid e\} \quad e[u_c/x] \longrightarrow^* \mathsf{true}_{\mathbf{Id}}}{\Gamma \vdash u_{c;\{x:T \mid e\}?} : \{x{:}T \mid e\}} \ \text{T\_ValRefine}$$

$$\frac{\vdash \Gamma \quad \vdash \{x{:}T \mid e_1\} \quad \emptyset \vdash v : T \quad \emptyset \vdash e_2 : \mathsf{Bool} \quad e_1[v/x] \longrightarrow^* e_2}{\Gamma \vdash \langle \{x{:}T \mid e_1\}, e_2, v \rangle : \{x{:}T \mid e_1\}} \ \text{T\_Check}$$

Figure 14: Updated typing rules for Efficient

(P_DynR) By C_BTag, C_Compose and IH (3) with C_FunTag, or C_Compose and IH (3) with C_PredTag and IH (4), depending on T. (4).

(P_Fun) By C_Fun and IH (3).

(P_RefineL) By C_Compose and IH (3), with C_PredTag and IH (4).

(P_RefineR) By C_Compose and IH (3), with C_PredUntag and IH (4).

□

5 A space-efficient coercion calculus

Having developed the naïve semantics in Naive, we now turn to space efficiency. In this section, we will define Efficient, a space-efficient coercion calculus.

There are two loci of inefficiency: coercion merges and function proxies (functional coercions). When F_Merge applies, it merely concatenates two coercions: $\langle c_1 \rangle\ (\langle c_2 \rangle\ e) \longrightarrow_n \langle c_2; c_1 \rangle\ e$. Efficient's semantics combines c_2 and c_1 to eliminate redundant checks. To bound the number of function proxies, we'll make sure that coercion merging combines adjacent functional coercions and change the tagging scheme on values—while Naive allows an arbitrary stack of tags values, Efficient will keep the size of value tags bounded. The solution to both of these problems lies in *canonical coercions* and our *merge* algorithm. Before we describe them below in Section 5.1, we discuss changes to the syntax of values and to the typing rules.

We make changes to both the syntax (Figure 13) and typing rules (Figure 14) of Naive from Section 3; we define an entirely new operational semantics for Efficient in Figure 17. Throughout the new typing rules, we assume that coercions are canonical (see below).

In Naive, tagging is stacked: a value is either a pre-value tagged with **Id** or a value tagged with a single primitive coercion. In Efficient, we collapse this stack: values are pre-values tagged with a composite coercion, u_c. Naïve, stacked values were typed using T_PreVal, T_TagVal, and T_TagValRefine; we now use rules T_Val and T_ValRefine to type values.

We also change the structure of coercions slightly, treating **Fail** as a composite coercion. We do this because coercion normalization doesn't allow composite coercions with **Fail** at the top level—such coercions are normalized to just **Fail** itself.

The changes to the typing rules aren't major: T_Val and T_ValRefine account for flattened values: T_Val will apply to u_c unless the coercion c ends in $\{x{:}T \mid e\}?$, in which case the typing derivation for the value will be T_ValRefine around T_Val. We separate the two rules to make sure we have value inversion, as outlined in our philosophy (Section 8). That is, we want to ensure that if a value has a refinement check tag on it, it satisfies that refinement. The T_Check rule changes to use the space-efficient semantics, but it remains a technical rule for supporting the evaluation of programs. Finally, we change all of the typing rules to require that coercions appearing in the program source are *canonical*. Before discussing the new evaluation rules, we discuss our space-efficient coercions and what we mean by a canonical coercion.

5.1 Space-efficient coercions

We define a set of *canonical coercions*, further subdivided into value coercions for constants and for functions. We list these coercions in Table 1 below; we prove that they are in fact *the* normal coercions for a standard set of rewrite rules given in Figure 15in Lemma 5.5 and Lemma 5.7. Next, we define a set of rules for merging coercions, proving that merging two well typed canonical coercions yields a well typed canonical coercion—no bigger than the previous two combined. This is how we will show space efficiency: where a naïve implementation would accumulate and discharge all checks, Efficient will keep its coercions in canonical form for which we have a bounded size (see Table 1).

$$
\begin{array}{rll}
\mathbf{Fail}; c & \longrightarrow \mathbf{Fail} & \text{(F\_Fail)} \\
c; \mathbf{Fail} & \longrightarrow \mathbf{Fail} & \\
(c_{11} \mapsto c_{12}); (c_{21} \mapsto c_{22}) & \longrightarrow (c_{21}; c_{11}) \mapsto (c_{12}; c_{22}) & \\
B!; B? & \longrightarrow \mathbf{Id} & \text{(F\_TagBB)} \\
B!; B'? & \longrightarrow \mathbf{Fail} \text{ when } B \neq B' & \\
 & & \text{(F\_TagBFailB)} \\
\mathbf{Fun}!; \mathbf{Fun}? & \longrightarrow \mathbf{Id} & \text{(F\_TagFunFun)} \\
B!; \mathbf{Fun}? & \longrightarrow \mathbf{Fail} & \text{(F\_TagBFailFun)} \\
\mathbf{Fun}!; B? & \longrightarrow \mathbf{Fail} & \text{(F\_TagFunFailB)} \\
\{x{:}T \mid e\}?; \{x{:}T \mid e\}! & \longrightarrow \mathbf{Id} & \text{(F\_TagPredPred)} \\
\{x{:}T \mid e\}!; \{x{:}T \mid e\}? & \longrightarrow \mathbf{Id} &
\end{array}
$$

$$
\frac{d_{i-1}; d_i \longrightarrow c}{d_1; \ldots; d_{i-1}; d_i; \ldots; d_n \longrightarrow d_1; \ldots; c; \ldots; d_n} \quad \textsc{Compat}
$$

$$
\frac{c_1 \longrightarrow c_1'}{d_1; \ldots; (c_1 \mapsto c_2); \ldots; d_n \longrightarrow d_1; \ldots; (c_1' \mapsto c_2); \ldots; d_n} \quad \textsc{FunDom}
$$

$$
\frac{c_2 \longrightarrow c_2'}{d_1; \ldots; (c_1 \mapsto c_2); \ldots; d_n \longrightarrow d_1; \ldots; (c_1 \mapsto c_2'); \ldots; d_n} \quad \textsc{FunCod}
$$

Figure 15: Rewriting rules

Henglein and Herman et al. define coercions with slightly different structure: for us, **Id** is notation for the empty composite coercion, but for them it is a coercion in its own right. Henglein and Herman et al.'s systems work by taking a single rewrite rule, the so-called ϕ rule:

$$B!; B? \longrightarrow \mathbf{Id} \qquad (\phi)$$

They then define a term rewriting system modulo an equational theory obtained by completing the following rules with reflexivity, symmetry, transitivity, and compatibility:

$$
\begin{array}{rcl}
\mathbf{Id}; c & = & c \\
c; \mathbf{Id} & = & c \\
c_{11} \mapsto c_{12}; c_{21} \mapsto c_{22} & = & (c_{21}; c_{11}) \mapsto (c_{12}; c_{22}) \\
(c_1; c_2); c_3 & = & c_1; (c_2; c_3)
\end{array}
$$

We, however, directly define a rewrite system in Figure 15, where we lift the two-coercion rewrite rules to composite coercions with the rules Compat, FunDom, and FunCod. Recall that $c = \mathbf{Id}; c = c; \mathbf{Id}$.

Note that many of these rules correspond to reduction rules in Naive's operational semantics; we've written the reduction rule names next to such rewrite rules. The rules for predicates over base types and type dynamic are new. Just as $B?$ takes a less specific type, ?, to a more specific type B while

performing a check (C_BUntag), we have $\{x{:}B \mid e\}?$ take a less specific type, B to a more specific type $\{x{:}B \mid e\}$ while performing a check (C_PredUntag). By the same analogy, $\{x{:}B \mid e\}!$ takes a more specific type to a less specific one. The rules for refinements don't have just a ϕ rule—they must have what Henglein calls a ψ rule:

$$\{x{:}T \mid e\}?; \{x{:}T \mid e\}! \longrightarrow \mathbf{Id}$$

Consider the rule F_TagPredPred from the naïve operational semantics (Figure 9): we must have a ψ rule if $\{x{:}T \mid e\}!$ is going to untag $v_{\{x:T|e\}?}$. But if $\{x{:}T \mid e\}?; \{x{:}T \mid e\}! \longrightarrow \mathbf{Id}$ on tags, it must also hold for coercions on the stack, if we want space efficiency. Swapping the meaning of the coercions ending ? and ! won't change anything: the reduction rule corresponding to F_TagPredPred will still eliminate some checking coercions no matter what. That is, space-efficient refinement checking *must* drop some checks on the floor. The ϕ rule for refinements is an optimization, unnecessary for soundness and space efficiency.

5.1 Lemma [Preservation for rewriting]: If $\vdash c : T_1 \rightsquigarrow T_2$ and $c \longrightarrow c'$ then $\vdash c' : T_1 \rightsquigarrow T_2$.

Proof: By case analysis on the reduction step taken. □

5.2 Lemma [Confluence]: Given a well typed coercion c_1, if $c_1 \longrightarrow c_2$ and $c_1 \longrightarrow c_2'$ then $c_2 \longrightarrow^* c_3$ and $c_2' \longrightarrow^* c_3$.

Proof: By case analysis on the critical pairs. If the reductions take place in non-overlapping parts of the term, then we simply make the appropriate, other, reduction step in both terms.

If the reduction takes place in an overlapping critical pair, we must reason more carefully. If one of the steps is any of the **Id** or **Fail** cases, we are done: simply take the appropriate step in the other term.

The remaining critical pairs are multiple functional coercions in a row and overlapping refinement coercions.

Suppose we have $c_1 = c; (c_{11} \mapsto c_{12}); (c_{21} \mapsto c_{22}); (c_{31} \mapsto c_{32}); c'$ and

$$\begin{array}{lll} c_1 & \longrightarrow & c_2 = c; ((c_{21}; c_{11}) \mapsto (c_{12}; c_{22})); (c_{31} \mapsto c_{32}); c' \\ c_1 & \longrightarrow & c_2' = c; (c_{11} \mapsto c_{12}); ((c_{31}; c_{21}) \mapsto (c_{22}; c_{32})); c'. \end{array}$$

We can simply reduce each side once more to find $c; (c_{31}; c_{21}; c_{11}) \mapsto (c_{12}; c_{22}; c_{32}); c'$ on both sides.

Suppose we have $c_1 = c; \{x{:}T \mid e\}?; \{x{:}T \mid e\}!; \{x{:}T \mid e\}?; c'$ and

$$\begin{array}{lll} c_1 & \longrightarrow & c_2 = c; \mathbf{Id}; \{x{:}T \mid e\}?; c' \\ c_1 & \longrightarrow & c_2' = c; \{x{:}T \mid e\}?; \mathbf{Id}; c'. \end{array}$$

Note that the refinements must be the same to have a well typed critical pair where the either reduction rule could fire. In this case, we can simply reduce the **Id** on either side to find confluence. The case for $c_1 = c; \{x{:}T \mid e\}!; \{x{:}T \mid e\}?; \{x{:}T \mid e\}!; c'$ is symmetric.

□

5.3 Lemma: The rewrite system of Figure 15 is normalizing.

Proof: We define coercion size straightforwardly, as follows:

$$\begin{array}{rcl} \mathrm{size}(\mathbf{Id}) = \mathrm{size}(\mathbf{Fail}) & = & 1 \\ \mathrm{size}(D!) = \mathrm{size}(D?) & = & 1 \\ \mathrm{size}(d_1; \ldots; d_n) & = & \sum \mathrm{size}(d_i) \\ \mathrm{size}(c_1 \mapsto c_2) & = & 1 + \mathrm{size}(c_1) + \mathrm{size}(c_2) \end{array}$$

We show that if $c_1 \longrightarrow c_2$, then $\mathrm{size}(c_1) > \mathrm{size}(c_2)$ (by inspection). The only subtle case is $(c_{11} \mapsto c_{12}); (c_{21} \mapsto c_{22}) \longrightarrow (c_{21}; c_{11}) \mapsto (c_{12}; c_{22})$, wherein the size reduces by exactly 1, by eliminating one of the functional coercion constructors. □

5.4 Lemma: The rewrite system of Figure 15 is strongly normalizing on well typed coercions.

Proof: The rewrite system is normalizing (Lemma 5.3) and confluent (Lemma 5.2). □

5.5 Lemma: The canonical coercions are normal.

Proof: By inspection. □

5.6 Lemma: If $d_1; \ldots; d_n$ is canonical, then any prefix $d_1; \ldots$ or suffix $\ldots; d_n$ is also canonical.

Proof: By inspection of Figure 8. □

5.7 Lemma: If $\vdash c : T_1 \rightsquigarrow T_2$ and c is normal, then c is canonical.

Proof: By induction on the derivation of $\vdash c : T_1 \rightsquigarrow T_2$. Consulting Table 1 will speed up the case analysis, since the possible typings quickly circumscribe the possibilities.

(C_Id) **Id** is canonical.

(C_Fail) **Fail** is canonical.

(C_Compose) We have $\vdash d; c : T_1 \rightsquigarrow T_2$; by inversion we have $\vdash d : T_1 \rightsquigarrow T'$ and $\vdash c : T' \rightsquigarrow T_2$. Moreover, if $d; c$ is normal, then so is c—so by the IH, we know that c is canonical.

We will consider various possibilities for c, but in all cases we may rule out **Id**—because each of the primitive coercions is canonical—and **Fail**—because $d; \mathbf{Fail}$ is not normal. So in the following cases, we are looking at a coercion $d; c$ where c is non-empty. We go by cases on the former derivation:

(C_BUntag) $d = B?$. It must be the case that $T' = B$, so c is one of $B!$ or $B!; \{x{:}? \mid e\}?$ or $\{x{:}B \mid e\}?$. In all cases, we have immediately that $d; c$ is canonical.

(C_BTag) $d = B!$. It must be the case that $T' = ?$. It cannot be the case that $c = \mathbf{Fun}?; c'$, for then $d; c$ wouldn't be normal. By the same token, we can't have that $c = B?; c'$. The only remaining possibility is that $c = \{x{:}? \mid e\}?$, and $B!; \{x{:}? \mid e\}?$ is canonical.

(C_FunUntag) $d = \mathbf{Fun}?$. It must be the case that $T' = (?\to?)$, so c is one of **Fun**! or $\mathbf{Fun}!; \{x{:}? \mid e\}?$ or one of the $(c_1 \mapsto c_2); c'$ coercions. In all cases, $d; c$ is canonical.

(C_FunTag) $d = \mathbf{Fun}!$. It must be the case that $T' = ?$. It cannot be the case that $c = \mathbf{Fun}?; c'$ or $c = B!; c'$, because then $d; c$ wouldn't be normal. We can therefore conclude that $c = \{x{:}? \mid e\}?$, and $\mathbf{Fun}!; \{x{:}? \mid e\}?$ is canonical.

(C_Fun) $d = c_1 \mapsto c_2$. It must be the case that c_1 and c_2 are normal, so by the IH they are also canonical. It can't be the case that $c = (c_1' \mapsto c_2'); c'$—then $d; c$ wouldn't be normal.

So the only possibility is that (epending on the types of c_1 and c_2) the coercion c could be **Fun**! or $\mathbf{Fun}!; \{x{:}? \mid e\}?$. In both of these cases, $d; c$ is canonical.

(C_PredUntag) $d = \{x{:}T \mid e\}?$. The only possibility is that $c = \{x{:}T \mid e\}!; c'$, but this would not be normal—a contradiction. (That is, the only canonical coercion beginning $\{x{:}T \mid e\}?$ is exactly $\{x{:}T \mid e\}?$.)

(C_PredTag, $T = ?$) $d = \{x{:}? \mid e\}!$. So c must come from ?. If $\{x{:}? \mid e'\}?$, we must have $e \neq e'$ for $d; c$ to be normal, but then $d; c$ is canonical. If $c = B?; c'$, then $\{x{:}? \mid e\}!; B?; c'$ is canonical. The same is true when $c = \mathbf{Fun}?; c'$.

(C_PredTag, $T = B$) $d = \{x{:}B \mid e\}!$, so c comes from B.

So c is either $B!$ or $B!; \{x{:}? \mid e'\}?$ or $\{x{:}B \mid e'\}?$. For the first two, we have that $d; c$ is canonical. In the last case, $d; c$ is normal iff $e \neq e'$, in which case $d; c$ is canonical.

□

5.8 Corollary: All well typed coercions rewrite to a canonical coercion.

Proof: By strong normalization (Lemma 5.4), preservation (Lemma 5.1), and the fact well typed normal coercions are canonical (Lemma 5.7). □

Having developed the rewrite rules for our (somewhat relaxed) coercions, we define a merge algorithm in Figure 16 that takes two coercions and merges them from the edges. We will only ever merge *canonical* coercions, greatly simplifying the algorithm. Our merging relation implements the ϕ rule in N_BB and N_FunFun. We implement failure rules in N_BFailB, N_BFailFun, and N_FunFailB. We relate our coercion systems to others in Section 9.

Looking at Table 1, we can see why T_ValRefine only needs to check the last coercion on a tagged pre-value: if $\{x{:}T \mid e\}?$ appears in a canonical coercion, it appears at the end. Similarly, the observation that certain coercions

Coercion		Type		
Id	:	T	$\rightsquigarrow$	T
Fail	:	T	$\rightsquigarrow$	T'
$\{x{:}? \mid e\}?$	:	$?$	$\rightsquigarrow$	$\{x{:}? \mid e\}$
$B?$	:	$?$	$\rightsquigarrow$	B
$B?; B!$	:	$?$	$\rightsquigarrow$	$?$
$B?; B!; \{x{:}? \mid e\}?$	:	$?$	$\rightsquigarrow$	$\{x{:}? \mid e\}$
$B?; \{x{:}B \mid e\}?$	:	$?$	$\rightsquigarrow$	$\{x{:}B \mid e\}$
$\mathbf{Fun}?$	:	$?$	$\rightsquigarrow$	$?\to?$
$\mathbf{Fun}?; \mathbf{Fun}!$	:	$?$	$\rightsquigarrow$	$?$
$\mathbf{Fun}?; \mathbf{Fun}!; \{x{:}? \mid e\}?$	:	$?$	$\rightsquigarrow$	$\{x{:}? \mid e\}$
$\mathbf{Fun}?; c_1 \mapsto c_2$	:	$?$	$\rightsquigarrow$	$T_{21}\to T_{22}$
$\mathbf{Fun}?; c_1 \mapsto c_2; \mathbf{Fun}!$	:	$?$	$\rightsquigarrow$	$?$
$\mathbf{Fun}?; c_1 \mapsto c_2; \mathbf{Fun}!; \{x{:}? \mid e\}?$	:	$?$	$\rightsquigarrow$	$\{x{:}? \mid e\}$
$B!$	:	B	$\rightsquigarrow$	$?$
$B!; \{x{:}? \mid e\}?$	:	B	$\rightsquigarrow$	$\{x{:}? \mid e\}$
$\{x{:}B \mid e\}?$	:	B	$\rightsquigarrow$	$\{x{:}B \mid e\}$
$\mathbf{Fun}!$	:	$(?\to?)$	$\rightsquigarrow$	$?$
$\mathbf{Fun}!; \{x{:}? \mid e\}?$	:	$(?\to?)$	$\rightsquigarrow$	$\{x{:}? \mid e\}$
$c_1 \mapsto c_2$	:	$(T_{11}\to T_{12})$	$\rightsquigarrow$	$(T_{21}\to T_{22})$
$c_1 \mapsto c_2; \mathbf{Fun}!$	:	$(T_{11}\to T_{12})$	$\rightsquigarrow$	$?$
$c_1 \mapsto c_2; \mathbf{Fun}!; \{x{:}? \mid e\}?$	:	$(T_{11}\to T_{12})$	$\rightsquigarrow$	$\{x{:}? \mid e\}$
$\{x{:}? \mid e\}!$	:	$\{x{:}? \mid e\}$	$\rightsquigarrow$	$?$
$\{x{:}? \mid e\}!; \{x{:}? \mid e'\}?$	:	$\{x{:}? \mid e\}$	$\rightsquigarrow$	$?$ where $e \neq e'$
$\{x{:}? \mid e\}!; B?$	:	$\{x{:}? \mid e\}$	$\rightsquigarrow$	B
$\{x{:}? \mid e\}!; B?; B!$	:	$\{x{:}? \mid e\}$	$\rightsquigarrow$	$?$
$\{x{:}? \mid e\}!; B?; B!; \{x{:}? \mid e'\}?$	:	$\{x{:}? \mid e\}$	$\rightsquigarrow$	$\{x{:}? \mid e'\}$
$\{x{:}? \mid e\}!; B?; \{x{:}B \mid e'\}?$	:	$\{x{:}? \mid e\}$	$\rightsquigarrow$	$\{x{:}B \mid e'\}$
$\{x{:}? \mid e\}!; \mathbf{Fun}?$	:	$\{x{:}? \mid e\}$	$\rightsquigarrow$	$(?\to?)$
$\{x{:}? \mid e\}!; \mathbf{Fun}?; \mathbf{Fun}!$	:	$\{x{:}? \mid e\}$	$\rightsquigarrow$	$?$
$\{x{:}? \mid e\}!; \mathbf{Fun}?; \mathbf{Fun}!; \{x{:}? \mid e'\}?$	:	$\{x{:}? \mid e\}$	$\rightsquigarrow$	$\{x{:}? \mid e'\}$
$\{x{:}? \mid e\}!; \mathbf{Fun}?; c_1 \mapsto c_2$	:	$\{x{:}? \mid e\}$	$\rightsquigarrow$	$T_{21}\to T_{22}$
$\{x{:}? \mid e\}!; \mathbf{Fun}?; c_1 \mapsto c_2; \mathbf{Fun}!$	:	$\{x{:}? \mid e\}$	$\rightsquigarrow$	$?$
$\{x{:}? \mid e\}!; \mathbf{Fun}?; c_1 \mapsto c_2; \mathbf{Fun}!; \{x{:}? \mid e'\}?$	:	$\{x{:}? \mid e\}$	$\rightsquigarrow$	$\{x{:}? \mid e'\}$
$\{x{:}B \mid e\}!$	:	$\{x{:}B \mid e\}$	$\rightsquigarrow$	B
$\{x{:}B \mid e\}!; B!$	:	$\{x{:}B \mid e\}$	$\rightsquigarrow$	$?$
$\{x{:}B \mid e\}!; B!; \{x{:}? \mid e'\}?$	:	$\{x{:}B \mid e\}$	$\rightsquigarrow$	$\{x{:}? \mid e'\}$
$\{x{:}B \mid e\}!; \{x{:}B \mid e'\}?$	:	$\{x{:}B \mid e\}$	$\rightsquigarrow$	$\{x{:}B \mid e'\}$ where $e \neq e'$

Rows with a blue background are *value coercions*, and are the only coercions that can appear as tags on pre-values. Horizontal rules mark a change of initial primitive coercion.

Table 1: Canonical coercions

$$\frac{c_1; c_2 \text{ is canonical}}{c_1 * c_2 \Rightarrow (c_1; c_2)}\ \text{N\_Canonical} \qquad \frac{c_{21} * c_{11} \Rightarrow c_{31} \quad c_{12} * c_{22} \Rightarrow c_{32} \quad c_1 * (c_{31} \mapsto c_{32}); c_2 \Rightarrow c}{c_1; (c_{11} \mapsto c_{12}) * (c_{21} \mapsto c_{22}); c_2 \Rightarrow c}\ \text{N\_Fun}$$

$$\frac{}{\mathbf{Fail} * c \Rightarrow \mathbf{Fail}}\ \text{N\_FailL} \qquad \frac{c \neq \mathbf{Fail}}{c * \mathbf{Fail} \Rightarrow \mathbf{Fail}}\ \text{N\_FailR} \qquad \frac{c_1 * c_2 \Rightarrow c}{c_1; B! * B?; c_2 \Rightarrow c}\ \text{N\_BB} \qquad \frac{c_1 * c_2 \Rightarrow c}{c_1; \mathbf{Fun}! * \mathbf{Fun}?; c_2 \Rightarrow c}\ \text{N\_FunFun}$$

$$\frac{B \neq B'}{c_1; B! * B'?; c_2 \Rightarrow \mathbf{Fail}}\ \text{N\_BFailB} \qquad \frac{}{c_1; B! * \mathbf{Fun}?; c_2 \Rightarrow c}\ \text{N\_BFailFun} \qquad \frac{}{c_1; \mathbf{Fun}! * B?; c_2 \Rightarrow \mathbf{Fail}}\ \text{N\_FunFailB}$$

$$\frac{c_1 * c_2 \Rightarrow c}{c_1; \{x{:}T \mid e\}? * \{x{:}T \mid e\}!; c_2 \Rightarrow c}\ \text{N\_PredPred} \qquad \frac{c_1 * c_2 \Rightarrow c}{c_1; \{x{:}T \mid e\}! * \{x{:}T \mid e\}?; c_2 \Rightarrow c}\ \text{N\_PredSame}$$

Figure 16: Merging coercions

are *value coercions*, i.e., are the only coercions that can be applied to values, can be made based on typing: if constants are typed at simple types and lambdas are assigned functional types, then all value coercions must come from B or $T_1 \to T_2$.

We write $c_1 \Downarrow c_2$ (read "merge c_1 and c_2") for canonical coercions c_1 and c_2 to mean the coercion c such that $c_1 * c_2 \Rightarrow c$. This notation is justified by Lemma 5.11, which shows that merging is an operator on canonical coercions.

5.9 Lemma [Preservation for merge]: If $\vdash c_1 : T_1 \rightsquigarrow T_2$ and $\vdash c_2 : T_2 \rightsquigarrow T_3$ and $c_1 * c_2 \Rightarrow c_3$ then $\vdash c_3 : T_1 \rightsquigarrow T_3$.

Proof: By induction on the derivation of $c_1 * c_2 \Rightarrow c_3$.

(N_Canonical) By induction on the length of c_1, using C_Compose.

(N_FailL) We have $\mathbf{Fail} * c_2 \Rightarrow \mathbf{Fail}$. By regularity (Lemma 3.1), $\vdash T_3$. By inversion we have $\vdash T_1$, so by C_Fail we have $\vdash \mathbf{Fail} : T_1 \rightsquigarrow T_3$.

(N_FailR) We have $c_1 * \mathbf{Fail} \Rightarrow \mathbf{Fail}$. By regularity (Lemma 3.1), $\vdash T_1$. By inversion we have $\vdash T_3$, so by C_Fail we have $\vdash \mathbf{Fail} : T_1 \rightsquigarrow T_3$.

(N_BB) We have $\vdash c_1; B! : T_1 \rightsquigarrow ?$ and $\vdash B?; c_2 : ? \rightsquigarrow T_3$. By inversion, $\vdash c_1 : T_1 \rightsquigarrow B$ and $\vdash c_2 : B \rightsquigarrow T_3$. We can now apply the IH to find that $\vdash c : T_1 \rightsquigarrow T_3$.

(N_BFailB) We have $\vdash c_1; B! : T_1 \rightsquigarrow ?$ and $\vdash B'?; c_2 : ? \rightsquigarrow T_3$. By regularity (Lemma 3.1), we know that $\vdash T_1$ and $\vdash T_3$. So we can conclude by C_Fail that $\vdash \mathbf{Fail} : T_1 \rightsquigarrow T_3$.

(N_BFailFun) We have $\vdash c_1; B! : T_1 \rightsquigarrow ?$ and $\vdash \mathbf{Fun}?; c_2 : ? \rightsquigarrow T_3$. By regularity (Lemma 3.1), we know that $\vdash T_1$ and $\vdash T_3$. So we can conclude by C_Fail that $\vdash \mathbf{Fail} : T_1 \rightsquigarrow T_3$.

(N_FunFun) We have $\vdash c_1; \mathbf{Fun}! : T_1 \rightsquigarrow ?$ and $\vdash \mathbf{Fun}?; c_2 : ? \rightsquigarrow T_3$. By inversion, $\vdash c_1 : T_1 \rightsquigarrow (?\to?)$ and $\vdash c_2 : (?\to?) \rightsquigarrow T_3$. We can now apply the IH to find that $\vdash c : T_1 \rightsquigarrow T_3$.

(N_FunFailB) We have $\vdash c_1; \mathbf{Fun}! : T_1 \rightsquigarrow ?$ and $\vdash B?; c_2 : ? \rightsquigarrow T_3$. By regularity (Lemma 3.1), we know that $\vdash T_1$ and $\vdash T_3$. So we can conclude by C_Fail that $\vdash \mathbf{Fail} : T_1 \rightsquigarrow T_3$.

(N_PredPred) We have $\vdash c_1; \{x{:}T \mid e\}? : T_1 \rightsquigarrow \{x{:}T \mid e\}$ and $\vdash \{x{:}T \mid e\}!; c_2 : \{x{:}T \mid e\} \rightsquigarrow T_3$, where T is either B or ?. By inversion, $\vdash c_1 : T_1 \rightsquigarrow T$ and $\vdash c_2 : T \rightsquigarrow T_3$. We can now apply the IH to find that $\vdash c : T_1 \rightsquigarrow T_3$.

(N_PredSame) We have $\vdash c_1; \{x{:}T \mid e\}! : T_1 \rightsquigarrow T$ and $\vdash \{x{:}T \mid e\}!; c_2 : T \rightsquigarrow T_3$, where T is either B or ?. By inversion, $\vdash c_1 : T_1 \rightsquigarrow \{x{:}T \mid e\}$ and $\vdash c_2 : \{x{:}T \mid e\} \rightsquigarrow T_3$. We can now apply the IH to find that $\vdash c : T_1 \rightsquigarrow T_3$.

(N_Fun) We have $\vdash c_1; (c_{11} \mapsto c_{12}) : T_1 \rightsquigarrow (T_{21}\to T_{22})$ and $\vdash (c_{21} \mapsto c_{22}); c_2 : (T_{21}\to T_{22}) \rightsquigarrow T_3$. By inversion, we know that:

$$\begin{array}{ll}
\vdash c_1 : & T_1 \rightsquigarrow (T_{11}\to T_{12}) \\
\vdash c_{11} : & T_{21} \rightsquigarrow T_{11} \\
\vdash c_{12} : & T_{12} \rightsquigarrow T_{22} \\
\vdash c_{21} : & T_{31} \rightsquigarrow T_{21} \\
\vdash c_{22} : & T_{22} \rightsquigarrow T_{32} \\
\vdash c_2 : & (T_{31}\to T_{32}) \rightsquigarrow T_3
\end{array}$$

We have that $c_{21} * c_{11} \Rightarrow c_{31}$ and $c_{12} * c_{22} \Rightarrow c_{32}$, so by the IH we have that $\vdash c_{31} : T_{31} \rightsquigarrow T_{11}$ and $\vdash c_{32} : T_{12} \rightsquigarrow T_{32}$. By C_Fun, $\vdash c_{31} \mapsto c_{32} : (T_{11}\to T_{12}) \rightsquigarrow (T_{31}\to T_{32})$. We now have enough typing to apply the IH on $c_1 * (c_{31} \mapsto c_{32}); c_2 \Rightarrow c$ and find that $\vdash c : T_1 \rightsquigarrow T_3$.

□

5.10 Lemma [Merge is a function]: Given canonical coercions c_1 and c_2, if $c_1 * c_2 \Rightarrow c_3$ and $c_1 * c_2 \Rightarrow c_3'$, then $c_3 = c_3'$.

Proof: By induction on the derivation of $c_1 * c_2 \Rightarrow c_3$, observing in each case that whatever rule applied to form the derivation must be the one used to form $c_1 * c_2 \Rightarrow c_3'$. The only tricky case is N_Canonical. But if $c_1; c_2$ is canonical, then none of the N_... rules can apply. □

5.11 Lemma [Merge is an operator]: Given canonical coercions $\vdash c_1 : T_1 \rightsquigarrow T_2$ and $\vdash c_2 : T_2 \rightsquigarrow T_3$, then there exists a unique canonical coercion c such that $c_1 * c_2 \Rightarrow c$.

Proof: By induction on $\text{size}(c_1) + \text{size}(c_2)$, with a long case analysis. Uniqueness is by Lemma 5.10; we use Lemma 7.1 to apply the IH when merging two functional coercions. We can use Table 1 and types to narrow the search.

First, we can rule out cases where either coercion is **Id** (we just get the other coercion, which is already canonical) or **Fail** (we just get **Fail**, which is canonical). Now we go by analysis on the left coercion c_1. In many cases we will reduce out a few primitive coercions and then use the fact that prefixes and suffixes of canonical coercions are canonical (Lemma 5.6) to apply the IH.

($\{x{:}? \mid e\}?$) The only coercions that can apply are of the form $\{x{:}? \mid e\}!; c_2'$. In all cases, we will first apply N_PredPred, leaving us with **Id** on the left and c_2' on the right. We will have just c_2', which is canonical since it is a suffix of a canonical coercion (Lemma 5.6).

($B?$) The only canonical coercions that can apply are:

1. $B!$, where $B?; B!$ is canonical;
2. $B!; \{x{:}? \mid e\}?$, where $B?; B!; \{x{:}? \mid e\}?$ is canonical; and
3. $\{x{:}B \mid e\}?$, where $B?; \{x{:}B \mid e\}?$ is canonical.

($B?; B!$) Here c_2 can be any canonical coercion from ?. The possibilities are:

1. $\{x{:}? \mid e\}?$, where $B?; B!; \{x{:}? \mid e\}?$ is canonical;
2. $B?; c_2'$, with $B? * c_2' \Rightarrow c_3$ by the IH and Lemma 5.6, and we can then apply N_PredPred;
3. $B'?; c_2'$, where $B \neq B'$ and we have **Fail**;
4. $\mathbf{Fun}?; c_2'$, where we have **Fail**.

($B?; B!; \{x{:}? \mid e\}?$) Here c_2 can be any canonical coercion from $\{x{:}? \mid e\}?$. All of these are of the form $\{x{:}? \mid e\}!; c_2'$. By the IH, $B?; B! * c_2' \Rightarrow c$ for some canonical c; then we can apply N_PredPred.

($B?; \{x{:}B \mid e\}?$) Here c_2 can be any canonical coercion from $\{x{:}B \mid e\}$. All of these are of the form $\{x{:}B \mid e\}!; c_2'$. By the IH, $B? * c_2' \Rightarrow c$ for some canonical c; then we can apply N_PredPred.

(**Fun**?) Here c_2 can be any canonical coercion from ?→?, to which is either of the form $\mathbf{Fun}!; c_2'$ or $c_1 \mapsto c_2; c_2'$. In either case, $\mathbf{Fun}?; c_2$ is already canonical, so by N_Canonical we are done.

(**Fun**?; **Fun**!) Here c_2 can be any canonical coercion from ?. The possibilities are:

1. $\{x{:}? \mid e\}?$, where $\mathbf{Fun}?; \mathbf{Fun}!; \{x{:}? \mid e\}?$ is canonical;
2. $B?; c_2'$, where we have **Fail**;
3. $\mathbf{Fun}?; c_2'$, where we can step once by N_PredPred after observing that $\mathbf{Fun}? * c_2' \Rightarrow c$ for some canonical c by the IH and Lemma 5.6.

(**Fun**?; **Fun**!; $\{x{:}? \mid e\}?$) Here c_2 comes from $\{x{:}? \mid e\}$. These are all of the form $\{x{:}? \mid e\}!; c_2'$. If we are to step by N_PredPred, we must consider $\mathbf{Fun}?; \mathbf{Fun}!$ and c_2' (canonical by Lemma 5.6). Since their combined size is smaller than our original coercions, we know by the IH that $\mathbf{Fun}?; \mathbf{Fun}! * c_2' \Rightarrow c$ for some canonical c, and can step.

($\mathbf{Fun}?; c_{11} \mapsto c_{12}$) The coercion c_2 must come from a functional type that agrees with the type of $\vdash c_{11} \mapsto c_{12} : (T_{11} \to T_{12}) \rightsquigarrow (T_{21} \to T_{22})$, i.e., $\vdash c_2 : (T_{21} \to T_{22}) \rightsquigarrow T_3$. These coercions either begin $\mathbf{Fun}!; c_2'$ or $c_{21} \mapsto c_{22}; c_2'$. In the first case, $\mathbf{Fun}?; c_{11} \mapsto c_{12}; \mathbf{Fun}!; c_2'$ is canonical if $\mathbf{Fun}!; c_2'$ is canonical. In the second case, we will step by N_FunFun: we first apply the IH on the smaller coercions $\mathbf{Fun}?$ and $c_{31} \mapsto c_{32}; c_2'$ (using Lemma 7.1) to find a canonical c.

($\mathbf{Fun}?; c_{11} \mapsto c_{12}; \mathbf{Fun}!$) Here c_2 comes from type ?. If it is of the form $B?; c_2'$, we have **Fail**, which is canonical. If $c_2 = \{x{:}? \mid e\}?$, then $\mathbf{Fun}?; c_{11} \mapsto c_{12}; \mathbf{Fun}!; \{x{:}? \mid e\}?$ is canonical. If it is of the form $\mathbf{Fun}?; c_2'$, then we can step by N_FunFun: we must show how $\mathbf{Fun}?; c_{11} \mapsto c_{12}$ and c_2' merge. The latter is canonical by Lemma 5.6. Since both are canonical and their combined size is smaller than the original pair of coercions (by Lemma 7.1), we can apply the IH to find a canonical c that they merge to.

($\mathbf{Fun}?; c_{11} \mapsto c_{12}; \mathbf{Fun}!; \{x{:}? \mid e\}?$) Here c_2 comes from $\{x{:}? \mid e\}?$, so it must be a canonical coercion of the form $\{x{:}? \mid e\}!; c_2'$. We can combine them by N_PredPred, using Lemma 5.6 and the IH.

($B!$) In this case, c_2 comes from ?. If it is $\{x{:}? \mid e\}?$, we are done—$B!; \{x{:}? \mid e\}?$ is canonical. If it begins $\mathbf{Fun}!; c_2'$, we get **Fail**, which is canonical. If it begins $B?; c_2'$, we get c_2', which is canonical by Lemma 5.6.

($B!; \{x{:}? \mid e\}?$) Here $c_2 = \{x{:}? \mid e\}!; c_2'$, where c_2' is canonical by Lemma 5.6. We can apply the IH to find $B! * c_2' \Rightarrow c$ for some canonical c, stepping by N_PredPred.

($\{x{:}B \mid e\}?$) Here $c_2 = \{x{:}B \mid e\}!; c_2'$, where c_2' is canonical by Lemma 5.6. By N_PredPred and N_Canonical, we find that $\{x{:}B \mid e\}? * \{x{:}B \mid e\}!; c_2' \Rightarrow c_2'$.

($\mathbf{Fun}!$) In this case, c_2 comes from ?. If it is $\{x{:}? \mid e\}?$, we are done—$\mathbf{Fun}!; \{x{:}? \mid e\}?$ is canonical. If it begins $B!; c_2'$, we get **Fail**, which is canonical. If it begins $\mathbf{Fun}?; c_2'$, we get c_2', which is canonical by Lemma 5.6.

($\mathbf{Fun}!; \{x{:}? \mid e\}?$) Here $c_2 = \{x{:}? \mid e\}!; c_2'$, where c_2' is canonical by Lemma 5.6. We can apply the IH to find $\mathbf{Fun}! * c_2' \Rightarrow c$ for some canonical c.

($c_{11} \mapsto c_{12}$) The coercion c_2 must come from a functional type that agrees with the type of $\vdash c_{11} \mapsto c_{12} : (T_{11} \to T_{12}) \rightsquigarrow (T_{21} \to T_{22})$, i.e., $\vdash c_2 : (T_{21} \to T_{22}) \rightsquigarrow T_3$. These coercions either begin $\mathbf{Fun}!; c_2'$ or $c_{21} \mapsto c_{22}; c_2'$. In the first case, $c_{11} \mapsto c_{12}; \mathbf{Fun}!; c_2'$ is canonical if $\mathbf{Fun}!; c_2'$ is canonical.

In the second case, we use the IH to find that $c_{21} * c_{11} \Rightarrow c_{31}$ and $c_{12} * c_{22} \Rightarrow c_{32}$. If $c_{21} \mapsto c_{22}; c_2'$ was canonical, so must be $c_{31} \mapsto c_{32}; c_2'$, which is what we are left with after stepping by N_Fun and by N_Canonical.

($c_{11} \mapsto c_{12}; \mathbf{Fun}!$) In this case, c_2 comes from ?. If it is $\{x{:}? \mid e\}?$, we are done—$\mathbf{Fun}!; \{x{:}? \mid e\}?$ is canonical. If it begins $B!; c_2'$, we get **Fail**, which is canonical. If it begins $\mathbf{Fun}?; c_2'$, we can apply the IH (since c_2' is canonical by Lemma 5.6) to show that $c_{11} \mapsto c_{12} * c_2' \Rightarrow c$ for some canonical c, stepping by N_FunFun.

($c_{11} \mapsto c_{12}; \mathbf{Fun}!; \{x{:}? \mid e\}?$) Here $c_2 = \{x{:}? \mid e\}!; c_2'$, where c_2' is canonical by Lemma 5.6. We can apply the IH to find $c_{11} \mapsto c_{12}; \mathbf{Fun}! * c_2' \Rightarrow c$ for some canonical c, stepping by N_PredPred.

($\{x{:}? \mid e\}!$) Here c_2 must come from ?.

If it is $\{x{:}? \mid e\}?$, then we step by N_PredSame and are left with **Id**. If it is $\{x{:}? \mid e'\}?$ for $e \neq e'$, then $\{x{:}? \mid e\}!; \{x{:}? \mid e\}?$ is canonical.

If it is $B!; c_2'$ or $\mathbf{Fun}!; c_2'$, then again $\{x{:}? \mid e\}!; c_2$ is canonical.

($\{x{:}? \mid e\}!; \{x{:}? \mid e'\}?$) The coercion c_2 must be of the form $\{x{:}? \mid e''\}!; c_2'$, where c_2' is canonical by Lemma 5.6. So by the IH, $\{x{:}? \mid e\}! * c_2' \Rightarrow c$ for some canonical c, and we can step by N_PredPred.

($\{x{:}? \mid e\}!; B?$) Here c_2 must be either $B!; c_2'$ or $\{x{:}B \mid e'\}?$. In either case, the concatenation of the two is canonical.

($\{x{:}? \mid e\}!; B?; B!$) In this case, c_2 comes from ?. If it is $\{x{:}? \mid e'\}?$, we are done—$\{x{:}? \mid e\}!; B?; B!; \{x{:}? \mid e'\}?$ is canonical. If it begins $\mathbf{Fun}!; c_2'$, we get **Fail**, which is canonical. If it begins $B?; c_2'$, we can apply the IH to find $\{x{:}? \mid e\}!; B? * c_2' \Rightarrow c$, since c_2' is canonical by Lemma 5.6. We can then apply N_BB.

($\{x{:}? \mid e\}!; B?; B!; \{x{:}? \mid e'\}?$) Here $c_2 = \{x{:}? \mid e'\}!; c_2'$, where c_2' is canonical by Lemma 5.6. We can apply the IH to find $\{x{:}? \mid e\}!; B?; B! * c_2' \Rightarrow c$ for some canonical c, stepping by N_PredPred.

($\{x{:}? \mid e\}!; B?; \{x{:}B \mid e'\}?$) Here $c_2 = \{x{:}B \mid e'\}!; c_2'$, where c_2' is canonical by Lemma 5.6. We can apply the IH to find $\{x{:}? \mid e\}!; B? * c_2' \Rightarrow c$ for some canonical c, stepping by N_PredPred.

($\{x{:}? \mid e\}!; \mathbf{Fun}?$) Here c_2 can be any canonical coercion from ?→?, to which is either of the form $\mathbf{Fun}!; c_2'$ or $c_1 \mapsto c_2; c_2'$. In either case, $\{x{:}? \mid e\}!; \mathbf{Fun}?; c_2$ is already canonical, so by N_Canonical we are done.

($\{x{:}? \mid e\}!; \mathbf{Fun}?; \mathbf{Fun}!$) In this case, c_2 comes from ?. If it is $\{x{:}? \mid e\}?$, we are done—$\{x{:}? \mid e\}!; \mathbf{Fun}?; \mathbf{Fun}!; \{x{:}? \mid e\}?$ is canonical. If it begins $B!; c_2'$, we get **Fail**, which is canonical. If it begins $\mathbf{Fun}?; c_2'$, we can apply the IH (since c_2' is canonical by Lemma 5.6) to show that $\{x{:}? \mid e\}!; \mathbf{Fun}? * c_2' \Rightarrow c$ for some canonical c, stepping by N_FunFun.

($\{x{:}? \mid e\}!; \mathbf{Fun}?; \mathbf{Fun}!; \{x{:}? \mid e'\}?$) Here $c_2 = \{x{:}? \mid e'\}!; c_2'$, where c_2' is canonical by Lemma 5.6. We can apply the IH to find $\{x{:}? \mid e\}!; \mathbf{Fun}?; \mathbf{Fun}! * c_2' \Rightarrow c$ for some canonical c, stepping by N_PredPred.

($\{x{:}? \mid e\}!; \mathbf{Fun}?; c_{11} \mapsto c_{12}$) The coercion c_2 must come from a functional type that agrees with the type of $\vdash c_{11} \mapsto c_{12} : (T_{11} \to T_{12}) \rightsquigarrow (T_{21} \to T_{22})$, i.e., $\vdash c_2 : (T_{21} \to T_{22}) \rightsquigarrow T_3$. These coercions either begin $\mathbf{Fun}!; c_2'$ or $c_{21} \mapsto c_{22}; c_2'$. In the first case, $c_{11} \mapsto c_{12}; \mathbf{Fun}!; c_2'$ is canonical if $\mathbf{Fun}!; c_2'$ is canonical.

In the second case, the IH gives us that $c_{21} * c_{11} \Rightarrow c_{31}$ and $c_{12} * c_{22} \Rightarrow c_{32}$ (using Lemma 7.1 for the size argument). If $c_{21} \mapsto c_{22}; c_2'$ was canonical, so must be $(c_{31} \mapsto c_{32}); c_2'$. By the IH again, $\{x{:}? \mid e\}!; \mathbf{Fun}? * (c_{31} \mapsto c_{32}); c_2' \Rightarrow c$ for some canonical c, so we can step by N_Fun.

($\{x{:}? \mid e\}!; \mathbf{Fun}?; c_{11} \mapsto c_{12}; \mathbf{Fun}!$) In this case, c_2 comes from ?. If it is $\{x{:}? \mid e'\}?$, we are done—$\{x{:}? \mid e\}!; \mathbf{Fun}?; c_{11} \mapsto c_{12}; \mathbf{Fun}!; \{x{:}? \mid e'\}?$ is canonical. If it begins $B!; c_2'$, we get **Fail**, which is canonical. If it begins $\mathbf{Fun}?; c_2'$, we can apply the IH (since c_2' is canonical by Lemma 5.6) to show that $\{x{:}? \mid e\}!; \mathbf{Fun}?; c_{11} \mapsto c_{12} * c_2' \Rightarrow c$ for some canonical c, stepping by N_FunFun.

$; \mathbf{Fun}?; c_{11} \mapsto c_{12}; \mathbf{Fun}!; \{x{:}? \mid e'\}?$) Here $c_2 = \{x{:}? \mid e'\}!; c_2'$, where c_2' is canonical by Lemma 5.6. We can apply the IH to find $\{x{:}? \mid e\}!; \mathbf{Fun}?; c_{11} \mapsto c_{12}; \mathbf{Fun}! * c_2' \Rightarrow c$ for some canonical c, stepping by N_PredPred.

($\{x{:}B \mid e\}!$) Here $c_2 = \{x{:}B \mid e\}?; c_2'$. We step to c_2' by N_PredSame and N_Canonical. We have that c_2' is canonical by Lemma 5.6.

($\{x{:}B \mid e\}!; B!$) Here c_2 comes from ?. If it is $\{x{:}? \mid e'\}?$, we are done—$\{x{:}B \mid e\}!; B!; \{x{:}? \mid e'\}?$ is canonical. If it begins $\mathbf{Fun}!; c_2'$, we get **Fail**, which is canonical. If it begins $B?; c_2'$, we can use the IH to show that $\{x{:}B \mid e\}! * c_2' \Rightarrow c$ for a canonical c (since c_2' is canonical by Lemma 5.6). Then we can step by N_BB.

($\{x{:}B \mid e\}!; B!; \{x{:}? \mid e'\}?$) Here $c_2 = \{x{:}? \mid e'\}!; c_2'$. We know that c_2' is canonical (Lemma 5.6), so by the IH we have $\{x{:}B \mid e\}!; B! * c_2' \Rightarrow c$ for some canonical c. We step by N_PredPred.

($\{x{:}B \mid e\}!; \{x{:}B \mid e'\}?$) Here $c_2 = \{x{:}B \mid e'\}!; c_2'$. We know that c_2' is canonical (Lemma 5.6), so by the IH we have $\{x{:}B \mid e\}! * c_2' \Rightarrow c$ for some canonical c. We step by N_PredPred.

□

5.12 Lemma: For all canonical coercions c_1 and c_2, if $c_1 * c_2 \Rightarrow c_3$ then $c_1; c_2 \longrightarrow^* c_3$.

Proof: By induction on the derivation of $c_1 * c_2 \Rightarrow c_3$.

(N_Canonical) Since $c_3 = c_1; c_2$, we are done immediately by reflexivity.

(N_FailL) We have $\mathbf{Fail} * c_2 \Rightarrow \mathbf{Fail}$. By induction on the length of c_2, $\mathbf{Fail}; c_2 \longrightarrow^* \mathbf{Fail}$.

(N_FailR) We have $c_1 * \mathbf{Fail} \Rightarrow \mathbf{Fail}$. By induction on the length of c_1, $c_1; \mathbf{Fail} \longrightarrow^* \mathbf{Fail}$.

(N_BB) We have $c_1 * c_2 \Rightarrow c_3$. We have $c_1; B!; B?; c_2 \longrightarrow c_1; c_2$, so we are done by the IH.

(N_BFailB) We have $c_1; B!; B'?; c_2 \longrightarrow c_1; \mathbf{Fail}; c_2$. By induction on the length of c_1 and c_2, we can conclude that $c_1; \mathbf{Fail}; c_2 \longrightarrow^* \mathbf{Fail}$.

(N_BFailFun) We have $c_1; B!; \mathbf{Fun}?; c_2 \longrightarrow c_1; \mathbf{Fail}; c_2$. By induction on the length of c_1 and c_2, we know that $c_1; \mathbf{Fail}; c_2 \longrightarrow^* \mathbf{Fail}$.

(N_FunFun) We have $c_1 * c_2 \Rightarrow c_3$ and $c_1; \mathbf{Fun}!; \mathbf{Fun}?; c_2 \longrightarrow c_1; c_2$, so we are done by the IH.

(N_FunFailB) We have $c_1; \mathbf{Fun}!; B?; c_2 \longrightarrow c_1; \mathbf{Fail}; c_2$. By induction on the length of c_1 and c_2, we know that $c_1; \mathbf{Fail}; c_2 \longrightarrow^* \mathbf{Fail}$.

(N_PredPred) We have $c_1 * c_2 \Rightarrow c_3$ and $c_1; \{x{:}T \mid e\}?; \{x{:}T \mid e\}!; c_2 \longrightarrow c_1; c_2$, so we have $c_1; c_2 \longrightarrow^* c_3$ by the IH.

(N_PredSame) We have $c_1 * c_2 \Rightarrow c_3$ and $c_1; \{x{:}T \mid e\}!; \{x{:}T \mid e\}?; c_2 \longrightarrow c_1; c_2$, so we have $c_1; c_2 \longrightarrow^* c_3$ by the IH.

(N_Fun) We know that $c_{21} * c_{11} \Rightarrow c_{31}$ and $c_{12} * c_{22} \Rightarrow c_{32}$ and $c_1 * (c_{31} \mapsto c_{32}); c_2 \Rightarrow c_3$.

We can rewrite $c_1; (c_{11} \mapsto c_{12}); (c_{21} \mapsto c_{22}); c_2 \longrightarrow c_1; ((c_{21}; c_{11}) \mapsto (c_{12}; c_{22})); c_2$. By the IH, we know that $c_{21}; c_{11} \longrightarrow^* c_{31}$ and $c_{12}; c_{22} \longrightarrow^* c_{32}$, so we can rewrite to $c_1; (c_{31} \mapsto c_{32}); c_2$. But we know by the IH that this rewrites to c_3, so we are done.

□

5.13 Lemma [Merge is associative]: $c_1 \Downarrow (c_2 \Downarrow c_3) = (c_1 \Downarrow c_2) \Downarrow c_3$ for all canonical coercions c_1, c_2, and c_3.

Proof: Consider the term $c_1; c_2; c_3$. On the one hand, we can say that $c_1; c_2; c_3 \longrightarrow^* c_1; c_2 \Downarrow c_3 \longrightarrow^* c_1 \Downarrow c_2 \Downarrow c_3$ by Lemma 5.12. On the other hand, we also have $c_1; c_2; c_3 \longrightarrow^* c_1 \Downarrow c_2; c_3 \longrightarrow^* c_1 \Downarrow c_2 \Downarrow c_3$.

Recall that results of merges are canonical forms (Lemma 5.11), which are normal (Lemma 5.5). Since rewriting is confluent (Lemma 5.2), it must be that case that $c_1 \Downarrow (c_2 \Downarrow c_3) = (c_1 \Downarrow c_2) \Downarrow c_3$. □

5.2 Operational semantics

We give the changed operational semantics in Figure 17. The biggest change to our operational semantics is that E_Merge explicitly merges the two coercions. Herman et al. simply say that they keep their coercions in normal form—that is, we should interpret normalization happening automatically when E_Merge applies, even though they write E_Merge as directly concatenating the two coercions into $c_2; c_1$. Our semantics explicitly normalizes the coercions (rule E_Merge), possibly stopping the program on the next step (the no-longer-useless rule E_Fail).

Otherwise, the rules are largely the same as the naïve semantics, though we're now able to use merges to distill the tag rules into a few possibilities:

$$\frac{}{(\lambda x{:}T.\ e_{12})_{\mathbf{Id}}\ v_2 \longrightarrow e_{12}[v_2/x]}\ \text{E\_Beta} \qquad \frac{}{u_{1\,(c_1 \mapsto c_2)}\ v_2 \longrightarrow \langle c_2 \rangle\ (u_{1\,\mathbf{Id}}\ (\langle c_1 \rangle\ v_2))}\ \text{E\_Fun}$$

$$\frac{}{op(v_1, \ldots, v_n) \longrightarrow [\![op]\!]\,(v_1, \ldots, v_n)}\ \text{E\_Op} \qquad \frac{}{\langle \{x{:}T \mid e\}?; c \rangle\ v \longrightarrow \langle c \rangle\ \langle \{x{:}T \mid e\}, e[v/x], v \rangle}\ \text{E\_Check}$$

$$\frac{}{\langle \{x{:}T \mid e\}, \mathsf{true}_{\mathbf{Id}}, u_c \rangle \longrightarrow u_{c \Downarrow \{x:T \mid e\}?}}\ \text{E\_CheckOK} \qquad \frac{}{\langle \{x{:}T \mid e\}, \mathsf{false}_{\mathbf{Id}}, v \rangle \longrightarrow \mathsf{fail}}\ \text{E\_CheckFail}$$

$$\frac{d_1 \neq \{x{:}T \mid e\}? \quad c * d_1 \Rightarrow c' \quad c' \neq \mathbf{Fail}}{\langle d_1; c_2 \rangle\ u_c \longrightarrow \langle c_2 \rangle\ u_{c'}}\ \text{E\_Tag} \qquad \frac{d_1 \neq \{x{:}T \mid e\}? \quad c * d_1 \Rightarrow \mathbf{Fail}}{\langle d_1; c_2 \rangle\ u_c \longrightarrow \mathsf{fail}}\ \text{E\_TagFail}$$

$$\frac{}{\langle \mathbf{Id} \rangle\ v \longrightarrow v}\ \text{E\_TagId} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\ \text{E\_AppL} \qquad \frac{e_2 \longrightarrow e_2'}{v_1\ e_2 \longrightarrow v_1\ e_2'}\ \text{E\_AppR}$$

$$\frac{e_i \longrightarrow e_i'}{op(v_1, \ldots, v_{i-1}, e_i, \ldots, e_n) \longrightarrow op(v_1, \ldots, v_{i-1}, e_i', \ldots, e_n)}\ \text{E\_OpInner} \qquad \frac{}{\langle c_2 \rangle\ (\langle c_2 \rangle\ e) \longrightarrow \langle c_2 \Downarrow c_1 \rangle\ e}\ \text{E\_Merge}$$

$$\frac{e \neq \langle c' \rangle\ e'' \quad c \neq \mathbf{Fail} \quad e \longrightarrow e'}{\langle c \rangle\ e \longrightarrow \langle c \rangle\ e'}\ \text{E\_CoerceInner} \qquad \frac{e_2 \longrightarrow e_2'}{\langle \{x{:}T \mid e_1\}, e_2, v \rangle \longrightarrow \langle \{x{:}T \mid e_1\}, e_2', v \rangle}\ \text{E\_CheckInner}$$

$$\frac{e \neq \langle c \rangle\ e'}{\langle \mathbf{Fail} \rangle\ e \longrightarrow \mathsf{fail}}\ \text{E\_Fail} \qquad \frac{}{\langle c \rangle\ \mathsf{fail} \longrightarrow \mathsf{fail}}\ \text{E\_CoerceRaise}$$

$$\frac{}{\mathsf{fail}\ e_2 \longrightarrow \mathsf{fail}}\ \text{E\_AppRaiseL} \qquad \frac{}{v_1\ \mathsf{fail} \longrightarrow \mathsf{fail}}\ \text{E\_AppRaiseR}$$

$$\frac{}{op(v_1, \ldots, v_{i-1}, \mathsf{fail}, \ldots, e_n) \longrightarrow \mathsf{fail}}\ \text{E\_OpRaise} \qquad \frac{}{\langle \{x{:}T \mid e\}, \mathsf{fail}, v \rangle \longrightarrow \mathsf{fail}}\ \text{E\_CheckRaise}$$

Figure 17: Operational semantics

$$
\begin{array}{ll}
 & \mathsf{odd}\ 3_{\mathbf{Id}} \\
\longrightarrow & \mathsf{even}_{\mathsf{Int!}\mapsto\mathsf{Bool?}}\ 2_{\mathbf{Id}} \\
\longrightarrow & \langle \mathsf{Bool?} \rangle\ (\mathsf{even}\ (\langle \mathsf{Int!} \rangle\ 2_{\mathbf{Id}})) \\
\longrightarrow & \langle \mathsf{Bool?} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathsf{Int?}\mapsto\mathsf{Bool!}}\ 2_{\mathsf{Int!}}) \\
\longrightarrow & \langle \mathsf{Bool?} \rangle\ (\langle \mathsf{Bool!} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathbf{Id}}\ (\langle \mathsf{Int?} \rangle\ 2_{\mathsf{Int!}}))) \\
\longrightarrow & \langle \mathbf{Id} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathbf{Id}}\ (\langle \mathsf{Int?} \rangle\ 2_{\mathsf{Int!}})) \\
\longrightarrow & \langle \mathbf{Id} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathbf{Id}}\ 2_{\mathbf{Id}}) \\
\longrightarrow & \langle \mathbf{Id} \rangle\ (\mathsf{odd}\ 1_{\mathbf{Id}}) \\
\longrightarrow & \langle \mathbf{Id} \rangle\ (\mathsf{even}_{\mathsf{Int!}\mapsto\mathsf{Bool?}}\ 0_{\mathbf{Id}}) \\
\longrightarrow & \langle \mathbf{Id} \rangle\ (\langle \mathsf{Bool?} \rangle\ (\mathsf{even}\ (\langle \mathsf{Int!} \rangle\ 0_{\mathbf{Id}}))) \\
\longrightarrow & \langle \mathsf{Bool?} \rangle\ (\mathsf{even}\ (\langle \mathsf{Int!} \rangle\ 0_{\mathbf{Id}})) \\
\longrightarrow & \langle \mathsf{Bool?} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathsf{Int?}\mapsto\mathsf{Bool!}}\ 0_{\mathsf{Int!}}) \\
\longrightarrow & \langle \mathsf{Bool?} \rangle\ (\langle \mathsf{Bool!} \rangle\ (\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathbf{Id}}\ (\langle \mathsf{Int?} \rangle\ 0_{\mathsf{Int!}})) \\
\longrightarrow & \langle \mathbf{Id} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathbf{Id}}\ (\langle \mathsf{Int?} \rangle\ 0_{\mathsf{Int!}})) \\
\longrightarrow & \langle \mathbf{Id} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathbf{Id}}\ 0_{\mathbf{Id}}) \\
\longrightarrow & \langle \mathbf{Id} \rangle\ \mathsf{true}_{\mathbf{Id}} \\
\longrightarrow & \mathsf{true}_{\mathbf{Id}}
\end{array}
$$

Figure 18: Space-efficient reduction

E_Tag replaces all of the successful F_Tag* rules; E_TagFail replaces all of the failing F_Tag* rules. E_CheckOK is essentially F_CheckOK, though it uses a merge instead of concatenation (though the typing rules mean that the merge will apply N_Canonical every time).

We can finally observe that our reduction is space efficient: the coercions in Figure 18 don't grow with the size of the input like the coercions in Figure 10 or the casts in Figure 2. We discuss this claim in more detail in Section 7.

5.3 Proofs

Efficient enjoys type soundness; we show as much using standard syntactic methods.

We assume that $\mathsf{ty}(k)$ and $\mathsf{ty}(op)$ are always well formed.

5.14 Lemma [Regularity]: 1. If $\Gamma \vdash e : T$, then $\vdash T$.

2. If $\Gamma \vdash u : T$, then $\vdash T$.

3. If $\vdash \Gamma$, then $\vdash T$ for all $x{:}T \in \Gamma$.

Proof: By mutual induction on the typing derivations.

(T_Var) By the IH from (3).

(T_Val) By the IH from (2).

(T_ValRefine) By Lemma 3.1 on $\vdash \{x{:}T \mid e\}? : T \rightsquigarrow \{x{:}T \mid e\}$.

(T_Op) By assumption.

(T_App) By inversion of the IH from (1) on $\Gamma \vdash e_1 : T_1 \to T_2$.

(T_Coerce) By Lemma 3.1 on $\vdash c : T_1 \rightsquigarrow T_2$.

(T_Fail) By assumption.

(T_Check) By assumption.

(T_Const) By assumption.

(T_Abs) By WF_Fun on the assumption that $\vdash T_1$ and the IH from (1) on $\Gamma, x{:}T_1 \vdash e_{12} : T_2$.

(WF_Empty) Immediate.

(WF_Extend) By assumption and the IH from (3).

□

5.15 Lemma [Determinism]: If $e \longrightarrow e_1$ and $e \longrightarrow e_2$, then $e_1 = e_2$.

Proof: By induction on $e \longrightarrow e_1$, observing that in each case the same rule must have applied to find $e \longrightarrow e_2$. We use the uniqueness of canonical merges (Lemma 5.11) in the E_Merge case. □

5.16 Lemma [Canonical forms]: If $\emptyset \vdash v : T$ then:

- If $T = \mathsf{Bool}$, then $v = \mathsf{true}_{\mathbf{Id}}$ or $v = \mathsf{false}_{\mathbf{Id}}$.
- If $T = T_1 \to T_2$, then $v = \lambda x{:}T_1'.\ e_{\,c}$ where $c = \mathbf{Id}$ or $c = c_1 \mapsto c_2$.

Proof: By cases on the typing derivation. We observe that the only coercions from B to B are **Id** and **Fail**, and $\mathsf{true}_{\mathbf{Fail}}$ isn't well formed.

In the function case, the only coercions from $T_{11} \to T_{12}$ to $T_{21} \to T_{22}$ are **Id** and **Fail** and $c_1 \mapsto c_2$. Since $u_{\mathbf{Fail}}$ isn't well formed, c must be either the identity or a functional coercion. □

5.17 Lemma [Progress]: If $\emptyset \vdash e : T$ then either e is a result, or there exists an e' such that $e \longrightarrow e'$.

Proof: By induction on the typing derivation $\emptyset \vdash e : T$.

(T_Var) Contradictory—no such derivation.

(T_Val) u_c is a value result.

(T_ValRefine) $u_{c;\{x:T|e\}?}$ is a value result.

(T_Op) Either one of the e_i steps, or we step by E_Op.

(T_App) Either e_1 or e_2 step, or both are results. If either is fail, then we step by E_AppRaiseL or E_AppRaiseR. This leaves us with the case that they are both values. We have $\emptyset \vdash v_1\ v_2 : T_2$. By inversion, $\emptyset \vdash v_1 : T_1 \to T_2$ and $\emptyset \vdash v_2 : T_1$. By canonical forms (Lemma 5.16), $v_1 = \lambda x{:}T_1'.\ e_1{}_c$, where c is either $\mathbf{Id}$ or $c_1 \mapsto c_2$. We step by E_Beta or E_Fun.

(T_Coerce) We have $\emptyset \vdash \langle c \rangle\ e : T_2$, where $\vdash c : T_1 \rightsquigarrow T_2$ and $\emptyset \vdash e : T_1$. If the inner term is of the form $\langle c' \rangle\ e'$, we step by E_Merge (in which case we are well typed by Lemma 5.9). If $c = \mathbf{Fail}$, we step E_Fail (in which case we are well typed by Lemma 5.14 and T_Fail). If not, it either steps by E_CoerceInner or e is a result. If $e = \mathsf{fail}$, we step by E_CoerceRaise. If $e = u_{c'}$, we step by E_Tag, E_TagFail, or E_Check, depending on what the leftmost coercion in c is. If there is no leftmost coercion, i.e., we have $\langle \mathbf{Id} \rangle\ v$, then we step by E_TagId.

(T_Fail) fail is a result.

(T_Check) If e_2 steps, then we step by E_CheckInner. If not, it is a result. If $e_2 = \mathsf{fail}$, we step by E_CheckRaise.

By inversion, $\emptyset \vdash e_2 : \mathsf{Bool}$. By canonical forms (Lemma 5.16), if e_2 is a value, then it is either $\mathsf{true}_{\mathbf{Id}}$ or $\mathsf{false}_{\mathbf{Id}}$. In the former case, we step by E_CheckOK; in the latter case, we step by E_CheckFail.

□

5.18 Lemma [Weakening]: If $\Gamma_1, \Gamma_2 \vdash e : T$ and $x \notin \mathrm{dom}(\Gamma)$ and $\vdash T'$, then $\Gamma_1, x{:}T', \Gamma_2 \vdash e : T$.

Proof: By induction on e. □

5.19 Lemma [Substitution]: If $\emptyset \vdash v : T'$:

- if $\Gamma_1, x{:}T', \Gamma_2 \vdash e : T$ then $\Gamma_1, \Gamma_2 \vdash e[v/x] : T$.
- if $\Gamma_1, x{:}T', \Gamma_2 \vdash u : T$ then $\Gamma_1, \Gamma_2 \vdash u[v/x] : T$.

Proof: By induction on the typing derivation of e.

(T_Var) We have $\Gamma_1, x{:}T', \Gamma_2 \vdash y : T$. If $x = y$, then $T = T'$, and we have our result by weakening (Lemma 5.18). If $x \neq y$, then we have $y{:}T \in \Gamma_1, \Gamma_2$ and are done by T_Var.

(T_Val) By the IH and T_Val.

(T_ValRefine) By the IH and T_ValRefine, noting that all terms involved are actually closed.

(T_Op) By the IH and T_Op.

(T_App) By the IHs and T_App.

(T_Coerce) By the IHs and T_Coerce.

(T_Fail) Immediate, by T_Fail.

(T_Check) Immediate, by T_Check—the context is unused in the premises, and the terms involved are closed.

(T_Const) Immediate, by T_Const.

(T_App) By the IH and T_Abs.

□

5.20 Lemma [Preservation]: If $\emptyset \vdash e : T$ and $e \longrightarrow e'$, then $\emptyset \vdash e' : T$.

Proof: By induction on the typing derivation.

(T_Var) Contradictory—no such typing derivation exists.

(T_Val) Contradictory—doesn't step.

(T_ValRefine) Contradictory—doesn't step.

(T_Op) By assumption.

(T_App) We go by cases on the step taken:

(E_AppL) By the IH.

(E_AppR) By the IH.

(E_AppRaiseL) By T_Fail and regularity (Lemma 5.14).

(E_AppRaiseR) By T_Fail and regularity (Lemma 5.14).

(E_Beta) By substitution.

(E_Fun) By T_Coerce, T_App, and T_Val using C_Id for T_Val.

(T_Coerce) We go by cases on the step taken:

(E_CoerceInner) By the IH.

(E_Merge) By T_Coerce and Lemma 5.9.

(E_Fail) By T_Fail and regularity (Lemma 5.14).

(E_CoerceRaise) By T_Fail and regularity (Lemma 5.14).

(E_Tag) By T_Coerce, T_Val, and C_Compose, knowing that the result of the merge isn't a fail or $\{x{:}T \mid e\}$? for T_Val.

(E_TagFail) By T_Fail and regularity (Lemma 5.14).

(E_TagId) By the typing assumption on the value.

(E_Check) By T_Coerce and T_Check.

(T_Fail) Contradictory—doesn't step.

(T_Check) We go by cases on the step taken:

(E_CheckInner) By the IH.

(E_CheckOK) By T_ValRefine and C_Compose, knowing that the merge can't be a fail—predicates never introduce failures. We can take the evaluation we need directly from the T_Check derivation.

(E_CheckFail) By T_Fail and regularity (Lemma 5.14).

(E_CheckRaise) By T_Fail and regularity (Lemma 5.14).

□

In Section 3, we define cotermination and show that if $e \longrightarrow_n^* r$ then $\langle c\rangle\ e$ and $\langle c\rangle\ r$ coterminate, and if e diverges so does $\langle c\rangle\ e$ (Lemma 3.12). Unfortunately, neither of these exactly hold in Efficient For the former, consider $e = \langle\{x{:}\mathsf{Bool} \mid \mathsf{false}_{\mathbf{Id}}\}?\rangle\ \mathsf{true}_{\mathbf{Id}} \longrightarrow^* \mathsf{fail}$. The term $\langle\{x{:}\mathsf{Bool} \mid \mathsf{false}_{\mathbf{Id}}\}!\rangle\ e \longrightarrow^* \mathsf{true}_{\mathbf{Id}}$. For the latter, suppose that $\mathsf{diverge}$ is a closed, divergent term, such as $(\lambda x{:}(?\to?).\ x\ (\langle\mathbf{Fun}!\rangle\ x))_{\mathbf{Id}}\ (\langle\mathbf{Fun}? \mapsto \mathbf{Id}\rangle\ (\lambda x{:}(?\to?).\ x\ (\langle\mathbf{Fun}!\rangle\ x))_{\mathbf{Id}})$. On the one hand

$$\begin{array}{ll} & \langle\{x{:}\mathsf{Bool} \mid \mathsf{diverge}\}!\rangle\ (\langle\{x{:}\mathsf{Bool} \mid \mathsf{diverge}\}?\rangle\ \mathsf{true}_{\mathbf{Id}}) \\ \longrightarrow & \langle\mathbf{Id}\rangle\ \mathsf{true}_{\mathbf{Id}} \\ \longrightarrow & \mathsf{true}_{\mathbf{Id}} \end{array}$$

but $\langle\{x{:}\mathsf{Bool} \mid \mathsf{diverge}\}!\rangle\ \mathsf{true}_{\mathbf{Id}}$ diverges. The theorem doesn't hold for when e diverges or evaluates to fail, but it does for values.

5.21 Lemma: If $e \longrightarrow^* v$ and $\langle c\rangle\ v \longrightarrow^* v'$ then $\langle c\rangle\ e \longrightarrow^* v'$.

Proof: By induction on $e \longrightarrow^* v$, and then by cases on the step taken. (The base case ($e = v$) is immediate.)

If the step doesn't have an exposed coercion, we can just reapply the step taken: E_Beta, E_Fun, E_Op, E_CheckOK, E_CheckFail, E_AppL, E_AppR, E_OpInner, E_CheckInner, E_AppRaiseL, E_AppRaiseR, E_OpRaise, and E_CheckRaise all fit this rubric. The remaining cases must merge coercions in e with c. The general thrust is as for Lemma 3.12: we will merge and normalize, take some small tag manipulation step (determined by analyzing how the canonical merge went), and then reproduce that for a stepped e.

(E_TagId) We step by E_Merge (using N_Canonical) and apply the IH.

(E_Tag) We have $e_2 = \langle d_1; c_2\rangle\ u_{c_2'} \longrightarrow \langle c_2\rangle\ u_{c_2' \Downarrow d_1}$ with $d_1 \neq \{x{:}T \mid e\}?$.

Instead we step to $\langle d_1; c_2 \Downarrow c\rangle\ u_{c_2'}$ by E_Merge.

If d_1 remains unaffected by the merge, we are done easily: we step by E_Tag and can apply the IH on $\langle c\rangle\ (\langle c_2\rangle\ u_{c'})$.

If d_1 is affected, then all of $d_1; c_2$ must have disappeared. If $d_1; c_2 \Downarrow c = c$, then we are done by assumption. So instead it must be the case that $d_1; c_2 \Downarrow c$ is some suffix of c.

By the IH, $\langle c\rangle\ (\langle c_2\rangle\ u_{c_2'\Downarrow d_1}) \longrightarrow^*\ v'$, which steps to $\langle c_2 \Downarrow c\rangle\ u_{c_2'\Downarrow d_1}$. Whatever coercion was left in $c_2 \Downarrow c$ that eliminated d_1 must now be exposed, so we can step by E_Tag to find $\langle d_1; c_2 \Downarrow c\rangle\ u_{c_2'} \longrightarrow^*\ v'$.

(E_TagFail) Contradictory—fail isn't a value.

(E_Check) We have $e_2 = \langle\{x{:}T \mid e\}?; c'\rangle\ v_2' \longrightarrow \langle c'\rangle\ \langle\{x{:}T \mid e\}, e[v_2'/x], v_2'\rangle$; since this steps to some value v', it must be the case that $e[v_2'/x] \longrightarrow^* \mathsf{true}_{\mathbf{Id}}$.

The combiend term steps by E_Merge to $\langle\{x{:}T \mid e\}?; c' \Downarrow c\rangle\ v_2'$.

Now, the result of $\{x{:}T \mid e\}?; c' \Downarrow c$ either has the same refinement checking coercion on the front or it doesn't. If it does, we can use the evaluation we found above to step to $\langle c' \Downarrow c\rangle\ v_2'$ (using Lemma 5.13).

If it doesn't, we're already at $\langle c' \Downarrow c\rangle\ v_2'$. In either case, that term is equivalent to $\langle c\rangle\ (\langle c'\rangle\ v_2')$, and we can apply the IH on $\langle c'\rangle\ v_2' \longrightarrow^*\ v_2$.

(E_CoerceInner) We have $e_2 = \langle c_1\rangle\ (\langle c_2\rangle\ e_2') \longrightarrow \langle c_2 \Downarrow c_1\rangle\ e_2'$. We'll step by E_Merge twice to find $\langle c_2 \Downarrow c_1 \Downarrow c\rangle\ e_2'$. By associativity of merges, that term is equal to $\langle c_1 \Downarrow c_2 \Downarrow c\rangle\ e_2$.

By the IH we know that $\langle c\rangle\ (\langle c_2 \Downarrow c_1\rangle\ e_2') \longrightarrow^*\ v'$, so we are done.

(E_Fail) Contradictory—fail isn't a value.

(E_Merge) We step by E_Merge and then by E_Merge again to find the same result, and we are done.

□

6 Soundness of Efficient with regard to Naive

We will never get an exact semantic match between the naïve and space-efficient semantics: the ψ rule for refinements in the space-efficient semantics mean that some checks will happen in the naïve semantics that won't happen in the space-efficient semantics. All we can hope for is that *if* Naive produces a value, then Efficient will produce a similar one.

We adapt the asymmetric logical relations from Greenberg et al. [18] to show that the two calculi behave mostly the same, with Naive diverging and failing more often. We are forced into a *step-indexed* logical relation because of the type ?. In particular, the more common, fixpoint-on-types definition of our wouldn't work for the function tag case, since we need to take values at ? and relate them at the type ?→?.[6] We define the step-indexed logical relation in Figure 19; when we say $e \longrightarrow_n^m e'$, we mean that e steps to e' in *exactly* m steps. The definitions begin by defining a relation $v_1 \precsim^j v_2 : T$ for closed values and a relation $e_1 \precapprox^j e_2 : T$ for closed terms as a fixpoint on the index j.[7] We

[6] An early draft was buggy, using an ill defined type-indexed logical relation.

[7] The definition for terms is just fancy notation for a proposition involving the relation on values; this striated definition conveniently separates the definition of related values and evaluation, while avoiding the need for ⊤⊤-closure [33].

Value rules

$$\begin{array}{rcl}
\forall j.\ k_{\mathbf{Id}} \gtrsim^j k_{\mathbf{Id}} : B & \iff & \mathsf{ty}(k) = B \\
v_{11} \gtrsim^j v_{21} : T_1 \to T_2 & \iff & \\
\forall m < j.\ \forall v_{12} \gtrsim^m v_{22} : T_1.\ v_{11}\ v_{12} \gtrapprox^m v_{21}\ v_{22} : T_2 &&
\end{array}$$

$$\begin{array}{rcl}
v_{1\,B!} \gtrsim^j u_{2\,c \Downarrow B!} : ? & \iff & v_1 \gtrsim^j u_{2\,c} : B \\
v_{1\,\mathbf{Fun}!} \gtrsim^j u_{2\,c \Downarrow \mathbf{Fun}!} : ? & \iff & v_1 \gtrsim^j u_{2\,c} : ? \to ?
\end{array}$$

$$\begin{array}{c}
v_{1\,\{x:T \mid e_1\}?} \gtrsim^j u_{c \Downarrow \{x:T \mid e_2\}?} : \{x{:}T \mid e_1\} \\
\iff \\
\forall m < j.\ v_1 \gtrsim^m u_{2\,c} : T \wedge \{x{:}T \mid e_1\} \gtrsim^m \{x{:}T \mid e_2\}
\end{array}$$

Term rules

$$\begin{array}{l}
e_1 \gtrapprox^j e_2 : T \iff \\
e_1 \text{ diverges} \vee \\
\forall m < j. \quad e_1 \longrightarrow_n^m \mathsf{fail} \\
\qquad \vee \quad e_1 \longrightarrow_n^m v_1 \implies e_2 \longrightarrow^* v_2 \wedge v_1 \gtrsim^{(j-m)} v_2 : T
\end{array}$$

Type rules

$$\begin{array}{rcl}
B \gtrsim^j B \qquad ? \gtrsim^j ? & & \\
T_{11} \to T_{12} \gtrsim^j T_{21} \to T_{22} & \iff & T_{11} \gtrsim^j T_{21} \wedge T_{12} \gtrsim^j T_{22} \\
\{x{:}T \mid e_1\} \gtrsim^j \{x{:}T \mid e_2\} & \iff & \\
\forall m < j.\ \forall v_1 \gtrsim^m v_2 : T.\ e_1[v_1/x] \gtrapprox^m e_2[v_2/x] : \mathsf{Bool} &&
\end{array}$$

Closing substitutions

$$\begin{array}{rcl}
\Gamma \models^j \delta & \iff & \forall x \in \mathrm{dom}(\Gamma).\ \delta_1(x) \gtrsim^j \delta_2(x) : T \\
\Gamma \vdash e_1 \gtrapprox e_2 : T & \iff & \forall j \geq 0.\ \forall \Gamma \models^j \delta.\ \delta_1(e_1) \gtrapprox^j \delta_2(e_2) : T
\end{array}$$

Figure 19: Relating the naïve and space-efficient semantics

lift the definitions to open terms by defining *dual closing value substitutions* δ; if $\Gamma \models^j \delta$ and $x{:}T \in \Gamma$, then $\delta_1(x) \gtrsim^j \delta_2(x) : T$.

Naïve terms are on the left of the relation, while space-efficient terms are on the right. We require that both sides be well typed. We obtain the asymmetry we seek by saying that *when* the naïve semantics yields a value, then the space-efficient yields a similar one—but otherwise, the naïve semantics will fail or diverge. This definition still allows Efficient to diverge or to fail, but then the naïve semantics must also diverge or fail—but note that it's possible for the left-hand side of the relation to diverge and the right-hand side to fail, and vice versa. This is possible because the naïve semantics could run a check diverges, while the space-efficient semantics skips that check and instead runs a failing one.

Step-indexed logical relations are commonly asymmetric, for a separate reason: only one side needs the index; for us, it is particularly convenient to put the step-index on the naïve side, allowing us to skip reasoning about how step indices and merges interact.

We could try to be more specific in our relation: either you get the same thing, or you get divergence or a failure—that can be traced back exactly to a check that happened in NAIVE but not in EFFICIENT. We omit this more precise tracking for simplicity's sake.

The value relation $v_1 \gtrsim^j v_2 : T$ is subtler than usual for logical relation: the definitions at ? and $\{x{:}T \mid e\}$ must shuffle some tags around. In particular, the rule for type ? is split into cases by the underlying tag of values. The case for refinements $\{x{:}T \mid e\}$ requires that the values be related at the underlying type T (recalling that $T = B$ or $T = ?$) and also that the values be tagged as satisfying the predicate (or a related predicate in EFFICIENT).

Our proof works by showing that a well-typed naïve term e is related to its translation into the space-efficient term $\mathsf{canonical}(e)$. We define $\mathsf{canonical}$ in Figure 20, omitting most of the cases since they are homomorphic. We give the cases for coercions and for values because they are the most interesting. In particular, $\mathsf{canonical}(v)$ must unfold the stacked tags on a NAIVE value and merge them into a single coercion.

Before we continue, we must justify that $\mathsf{canonical}$ is a function—in particular, is the case for composite coercions valid?

6.1 Lemma: $\mathsf{canonical}$ is a well defined function that produces canonical coercions.

Proof: By induction on c, using Lemma 5.11 to show that $c_1 \Downarrow c_2$ is a unique canonical coercion when c_1 and c_2 are canonical. □

In order to put $\mathsf{canonical}(e)$ in the relation, we must know that it is well typed. Since some of the runtime typing terms require facts about derivations that will be hard to translate—in particular, T_TAGVALREFINE and T_CHECKNAIVE—we'll exclude them from our proof. Normal "source" terms shouldn't have any of these, anyway. Later on, when we've proved that naïve terms are logically related to their canonical translations, we'll have the evaluation derivations after all.

6.2 Lemma [Preservation for $\mathsf{canonical}$]: Assuming that no refinement tags or active checking forms are present:

1. If $\vdash c : T_1 \rightsquigarrow T_2$ then $\vdash \mathsf{canonical}(c) : \mathsf{canonical}(T_1) \rightsquigarrow \mathsf{canonical}(T_2)$.
2. If $\vdash d : T_1 \rightsquigarrow T_2$ then $\vdash \mathsf{canonical}(d) : \mathsf{canonical}(T_1) \rightsquigarrow \mathsf{canonical}(T_2)$.
3. If $\Gamma \vdash u : T$ then $\mathsf{canonical}(\Gamma) \vdash \mathsf{canonical}(u) : \mathsf{canonical}(T)$.
4. If $\Gamma \vdash e : T$ then $\mathsf{canonical}(\Gamma) \vdash \mathsf{canonical}(e) : \mathsf{canonical}(T)$.
5. If $\vdash T$ then $\vdash \mathsf{canonical}(T)$.
6. If $\vdash \Gamma$ then $\vdash \mathsf{canonical}(\Gamma)$.

Proof: By simultaneous induction on the derivations, using Lemma 5.9 when merging coercions.

Composite coercions

$$\begin{aligned}
\mathsf{canonical}(\mathbf{Id}) &= \mathbf{Id} \\
\mathsf{canonical}(d_1; \ldots; d_n) &= \\
&\qquad \mathsf{canonical}(d_1) \Downarrow \mathsf{canonical}(d_2; \ldots; d_n)
\end{aligned}$$

Primitive coercions

$$\begin{aligned}
\mathsf{canonical}(\mathbf{Fail}) &= \mathbf{Fail} \\
\mathsf{canonical}(D!) &= \mathsf{canonical}(D)! \\
\mathsf{canonical}(D?) &= \mathsf{canonical}(D)? \\
\mathsf{canonical}(c_1 \mapsto c_2) &= \mathsf{canonical}(c_1) \mapsto \mathsf{canonical}(c_2)
\end{aligned}$$

Tags

$$\begin{aligned}
\mathsf{canonical}(B) &= B \\
\mathsf{canonical}(\mathbf{Fun}) &= \mathbf{Fun} \\
\mathsf{canonical}(\{x{:}T \mid e\}) &= \{x{:}T \mid \mathsf{canonical}(e)\}
\end{aligned}$$

Pre-values

$$\begin{aligned}
\mathsf{canonical}(k) &= k \\
\mathsf{canonical}(\lambda x{:}T.\ e) &= \lambda x{:}\mathsf{canonical}(T).\ \mathsf{canonical}(e)
\end{aligned}$$

Expressions

$$\begin{aligned}
\mathsf{canonical}(x) &= x \\
\mathsf{canonical}(u_{\mathbf{Id}}) &= \mathsf{canonical}(u)_{\mathbf{Id}} \\
\mathsf{canonical}(v_d) &= u_{c \Downarrow d} \\
&\qquad \text{where } \mathsf{canonical}(v) = u_c \\
\mathsf{canonical}(\mathsf{fail}) &= \mathsf{fail} \\
\mathsf{canonical}(op(e_1, \ldots, e_n)) &= op(\mathsf{canonical}(e_1), \ldots, \mathsf{canonical}(e_n)) \\
\mathsf{canonical}(e_1\ e_2) &= \mathsf{canonical}(e_1)\ \mathsf{canonical}(e_2) \\
\mathsf{canonical}(\langle c \rangle\ e) &= \langle \mathsf{canonical}(c) \rangle\ \mathsf{canonical}(e) \\
\mathsf{canonical}(\langle \{x{:}T \mid e_1\}, e_2, v \rangle) &= \langle \mathsf{canonical}(\{x{:}T \mid e_1\}), \mathsf{canonical}(e_2), \mathsf{canonical}(v) \rangle
\end{aligned}$$

Types

$$\begin{aligned}
\mathsf{canonical}(B) &= B \\
\mathsf{canonical}(T_1 \to T_2) &= \mathsf{canonical}(T_1) \to \mathsf{canonical}(T_2) \\
\mathsf{canonical}(?) &= ? \\
\mathsf{canonical}(\{x{:}T \mid e\}) &= \{x{:}T \mid \mathsf{canonical}(e)\}
\end{aligned}$$

Contexts

$$\begin{aligned}
\mathsf{canonical}(\emptyset) &= \emptyset \\
\mathsf{canonical}(\Gamma, x{:}T) &= \mathsf{canonical}(\Gamma), x{:}\mathsf{canonical}(T)
\end{aligned}$$

Figure 20: Canonicalizing naïve terms

(C_Id) By C_Id.

(C_Compose) By IH (2) and IH (1) and Lemma 5.9.

(C_Fail) By C_Fail.

(C_BUntag) By C_BUntag.

(C_BTag) By C_BTag.

(C_FunUntag) By C_FunUntag.

(C_FunTag) By C_FunTag.

(C_Fun) By IH (1) and C_Fun.

(C_PredUntag) By IH (4) and C_PredUntag.

(C_PredTag) By IH (4) and C_PredTag.

(T_Const) By T_Const and IH (6).

(T_Abs) By IH (5) and IH (4), we can reapply T_Abs.

(T_Var) By IH (6) and T_Var.

(T_PreVal) By IH (3) and T_Val.

(T_TagVal) By IH (4), Lemma 5.9, and T_Val.

(T_TagValRefine) Contradictory—we assumed these weren't present.

(T_Op) By IH (4) and T_Op.

(T_App) By IH (4) and T_App.

(T_Coerce) By IH (1) and IH (4) and T_Coerce.

(T_Fail) By IH (6) and IH (5), we can reapply T_Fail.

(T_CheckNaive) Contradictory—we assumed these weren't present.

(WF_Base) By WF_Base.

(WF_Dyn) By WF_Dyn.

(WF_Fun) By IH (5) and WF_Fun.

(WF_Refine) By IH (4) and WF_Refine, noting that T is either B or ?, so $\mathsf{canonical}(T) = T$.

(WF_Empty) Immediate.

(WF_Extend) By IH (6) and IH (5), we can reapply WF_Extend.

□

Our ultimate goal is soundness: if $\Gamma \vdash e : T$ then $\Gamma \vdash e \gtrsim\!\!\approx \mathsf{canonical}(e) : T$. Our proof works in a few stages: first we define relations $\vdash c$ **ignorable** (coercions which are equivalent to **Id** or **Fail**) and $\vdash c$ **failable** (coercions which are equivalent to **Fail**). We define these relations in Figure 21. We then prove lemmas that allow us to easily work with ignorable and failable coercions (Lemma 6.7 and Lemma 6.8, respectively). Then we relate non-canonical coercions to canonical ones (using a separate inductive relation $\vdash c_1 \equiv c_2$, defined in Figure 21). We show that such related coercions are logically related on logically related values. We then prove a separate lemma showing that related coercions are logically related on related terms—this not a trivial extension of the similar lemma for values, due to coercion merges. With those lemmas to hand, we show soundness. Don't worry—we explain the proof less tersely as we go.

6.3 Lemma [Expansion]: If $e_1 \longrightarrow_n^* e_1'$ and $e_2 \longrightarrow^* e_2'$ then $e_1' \gtrsim\!\!\approx^j e_2 : T$ implies $e_1 \gtrsim\!\!\approx^j e_2 : T$.

Proof: Let $m < j$. If e_1' diverges, so does e_1 by determinism (Lemma 3.3) and we are done.

Otherwise, we have $e_1' \longrightarrow_n^m r$. So there exists an m' such that $e_1 \longrightarrow_n^{m'} r$. If $m' > j$, then we are done vacuously; if $m' < j$ then we are done by assumption. □

6.4 Lemma [Contraction]: If $e_1 \longrightarrow_n^* e_1'$ and $e_2 \longrightarrow^* e_2'$ then $e_1 \gtrsim\!\!\approx^j e_2 : T$ implie $e_1' \gtrsim\!\!\approx^j e_2 : T$.

Proof: Let $m < j$. If e_1 diverges, so does e_1' by determinism (Lemma 3.3), and we are done.

Otherwise, we have $e_1 \longrightarrow_n^m r$. But since $e_1 \longrightarrow_n^* e_1'$, we know that $e_1' \longrightarrow_n^{m'} r$ for $m' < m$ by determinism (Lemma 3.3), so we are done by assumption. □

6.5 Lemma [Expansion and contraction]: If $e_1 \longrightarrow_n^* e_1'$ and $e_2 \longrightarrow^* e_2'$ then $e_1 \gtrsim\!\!\approx^j e_2 : T$ iff $e_1' \gtrsim\!\!\approx^j e_2 : T$.

Proof: By Lemma 6.3 and Lemma 6.4. □

6.6 Lemma: Forall indices j, if $e_1 \longrightarrow_n^m \mathsf{fail}$ then $e_1 \gtrsim\!\!\approx^j e_2 : T$.

Proof: If e_1 goes to fail in $m \geq j$ steps, then we are done vacuously. If $m < j$, then are done by definition. □

Ignorable coercions can be freely added or removed to naïve terms while preserving logical relation to space-efficient terms. The I_Id rule obviously fits this bill; I_Fail is also acceptable, when we realize that Naive can fail more often than Efficient. I_BB captures the case of an injection of a base-value into type dynamic, with some possible extra coercions in the middle.

Ignorable coercions

$$\frac{}{\vdash \mathbf{Id}\,\mathbf{ignorable}}\ \text{I\_Id} \qquad \frac{}{\vdash \mathbf{Fail}\,\mathbf{ignorable}}\ \text{I\_Fail} \qquad \frac{\vdash c\,\mathbf{ignorable}}{\vdash B!; c; B?\,\mathbf{ignorable}}\ \text{I\_BB}$$

$$\frac{\vdash c\,\mathbf{ignorable}}{\vdash \mathbf{Fun}!; c; \mathbf{Fun}?\,\mathbf{ignorable}}\ \text{I\_FunFun} \qquad \frac{}{\vdash \{x{:}T \mid e\}?; \{x{:}T \mid e\}!\,\mathbf{ignorable}}\ \text{I\_PredPred}$$

$$\frac{\vdash c\,\mathbf{ignorable}}{\vdash \{x{:}T \mid e\}!; c; \{x{:}T \mid e\}?\,\mathbf{ignorable}}\ \text{I\_PredSame} \qquad \frac{\vdash c_1\,\mathbf{ignorable} \quad \vdash c_2\,\mathbf{ignorable}}{\vdash c_1; c_2\,\mathbf{ignorable}}\ \text{I\_Concat}$$

Failable coercions

$$\frac{}{\vdash \mathbf{Fail}\,\mathbf{failable}}\ \text{L\_Fail} \qquad \frac{B \neq B' \quad \vdash c\,\mathbf{ignorable}}{\vdash B!; c; B'?\,\mathbf{failable}}\ \text{L\_BB}$$

$$\frac{\vdash c\,\mathbf{ignorable}}{\vdash B!; c; \mathbf{Fun}?\,\mathbf{failable}}\ \text{L\_BFun} \qquad \frac{\vdash c\,\mathbf{ignorable}}{\vdash \mathbf{Fun}!; c; B?\,\mathbf{failable}}\ \text{L\_FunB}$$

Relating coercions

$$\frac{\vdash c_i'\,\mathbf{ignorable} \quad \vdash c_i \equiv d_i}{\vdash c_0'; c_1; c_1'; c_2; \ldots; c_{n-1}'; c_n; c_n' \equiv d_1; \ldots; d_n}\ \text{R\_Composite} \qquad \frac{\vdash c\,\mathbf{ignorable}}{\vdash c \equiv \mathbf{Id}}\ \text{R\_Id}$$

$$\frac{\begin{array}{c}\vdash c_i'\,\mathbf{ignorable} \\ \vdash (c_{1\,n\,1}; \ldots; c_{111}) \mapsto (c_{112}; \ldots; c_{1\,n\,2}) \equiv c_{21} \mapsto c_{22} \quad \vdash c_1\,\mathbf{failable}\end{array}}{\vdash (c_{111} \mapsto c_{112}); c_1'; \ldots; c_n'; (c_{1\,n\,1} \mapsto c_{1\,n\,2}) \equiv c_{21} \mapsto c_{22}; c_1'; c_2'}\ \text{R\_Fun} \qquad \text{R\_Fail}$$

$$\frac{\vdash D \equiv D}{\vdash D! \equiv D!}\ \text{R\_Tag} \qquad \frac{\vdash D \equiv D}{\vdash D? \equiv D?}\ \text{R\_Check} \qquad \frac{}{\vdash B \equiv B}\ \text{R\_DB} \qquad \frac{}{\vdash \mathbf{Fun} \equiv \mathbf{Fun}}\ \text{R\_DFun}$$

$$\frac{}{\vdash \{x{:}T \mid e_1\} \equiv \{x{:}T \mid \mathsf{canonical}(e_1)\}}\ \text{R\_DPredCanonical} \qquad \frac{x{:}\mathsf{Bool} \vdash e_1 \gtrsim e_2 : \mathsf{Bool}}{\vdash \{x{:}T \mid e_1\} \equiv \{x{:}T \mid e_2\}}\ \text{R\_DPredLR}$$

Figure 21: Relating coercions

I_FunFun and I_PredSame are similar. Note that I_PredPred can't have extra coercions in the middle—the coercion typing rules ensure that the only non-**Fail** coercion that can come after $\{x{:}T \mid e\}?$ is $\{x{:}T \mid e\}!$. The I_* rules try to capture the logic of similarly named N_* rules.

6.7 Lemma: If $\langle c_1; c_2\rangle\ v_1 \gtrsim^j e_2 : T$ and $\vdash c_1$ **ignorable** then $\langle c_2\rangle\ v_1 \gtrsim^j e_2 : T$.

Proof: By induction on $\vdash c_1$ **ignorable**.
Let $m < j$.

(I_Id) By definition, $\mathbf{Id}; c_2 = c_2$, so we are done immediately.

(I_Fail) $\langle \mathbf{Fail}; c_2\rangle\ v_1 \longrightarrow_n \mathsf{fail}$, and $\mathsf{fail} \gtrsim^j e_2 : T$ by Lemma 6.6.

(I_BB) $\langle B!; c_1'; B?; c_2\rangle\ v_1 \longrightarrow_n \langle c_1'; B?; c_2\rangle\ v_{1\,B!}$ by F_TagB; the reduced term is then related to e_2 at T (Lemma 6.4). Since $\vdash c_1'$ **ignorable**, we can apply the IH and find that $\langle B?; c_2\rangle\ v_{1\,B!} \gtrsim^j e_2 : T$. By F_TagBB and Lemma 6.4, we have $\langle c_2\rangle\ v_1 \gtrsim^j e_2 : T$¿

(I_FunFun) $\langle \mathbf{Fun}!; c_1'; \mathbf{Fun}?; c_2\rangle\ v_1 \longrightarrow_n \langle c_1'; \mathbf{Fun}?; c_2\rangle\ v_{1\,\mathbf{Fun!}}$ by F_TagFun; the reduced term is then related to e_2 at T (Lemma 6.4). Since $\vdash c_1'$ **ignorable**, we can apply the IH and find that $\langle \mathbf{Fun}?; c_2\rangle\ v_{1\,B!} \gtrsim^j e_2 : T$. By F_TagFunFun and Lemma 6.4, we have $\langle c_2\rangle\ v_1 \gtrsim^j e_2 : T$¿

(I_PredPred) $\langle \{x{:}T \mid e\}?; \{x{:}T \mid e\}!; c_2\rangle\ v_1 \longrightarrow_n \langle \{x{:}T \mid e\}!; c_2\rangle\ \langle \{x{:}T \mid e\}, e[v_1/x], v_1\rangle$ by F_Check. By type soundness (Theorem 3.8), we know that $e[v_1/x]$ either reduces a to value, reduces to fail, or diverges. In either of the last two cases, we are done by the definition of the logical relation or Lemma 6.6, respectively.

Suppose that $e[v_1/x] \longrightarrow_n^* v$; the value v must be either $\mathsf{true}_{\mathbf{Id}}$ or $\mathsf{false}_{\mathbf{Id}}$, since $\emptyset \vdash e[v_1/x] : \mathsf{Bool}$. In the latter case we are done by Lemma 6.6, just like for when the whole term reduced to fail.

So then $\langle \{x{:}T \mid e\}!; c_2\rangle\ \langle \{x{:}T \mid e\}, \mathsf{true}_{\mathbf{Id}}, v_1\rangle \longrightarrow_n \langle \{x{:}T \mid e\}!; c_2\rangle\ v_{1\,\{x:T|e\}}?$ by F_CheckOK. We can then step by F_TagPredPred and apply Lemma 6.4 to find that $\langle c_2\rangle\ v_1 \gtrsim^j e_2 : T$.

(I_PredSame) $\langle \{x{:}T \mid e\}!; c_1'; \{x{:}T \mid e\}?; c_2\rangle\ v_1 \gtrsim^j e_2 : T$. By inversion of the typing of v_1, we know that it must be of the form $v_{1\,\{x:T|e\}}'?$, and furthermore $e[v_1/x] \longrightarrow_n^* \mathsf{true}_{\mathbf{Id}}$.

Lemma 6.4 and F_PredPred give us $\langle c_1'; \{x{:}T \mid e\}?; c_2\rangle\ v_1' \gtrsim^j e_2 : T$, so we can apply the IH on $\vdash c_1'$ **ignorable** to find $\langle \{x{:}T \mid e\}?; c_2\rangle\ v_1' \gtrsim^j e_2 : T$. We then step by F_TagPredPred, then by F_Check. Since we know that $e[v_1'/x] \longrightarrow_n^* \mathsf{true}_{\mathbf{Id}}$, we can step the whole term using that relation and F_CheckOK to find $\langle c_2\rangle\ v_{1\,\{x:T|e\}}'? = \langle c_2\rangle\ v_1 \gtrsim^j e_2 : T$ by Lemma 6.4.

(I_Concat) $\langle c_{11}; c_{12}; c_2\rangle\ v_1 \gtrsim\!\!\approx^j e_2 : T$; by the IH on $\vdash c_{11}$ **ignorable**, we find that $\langle c_{12}; c_2\rangle\ v_1 \gtrsim\!\!\approx^j e_2 : T$. By the IH on $\vdash c_{12}$ **ignorable**, we find that $\langle c_2\rangle\ v_1 \gtrsim\!\!\approx^j e_2 : T$.

□

We prove a similar lemma that *failable* coercions always fail. The L_* rules try to capture the logic of similarly named N_Fail* rules.

6.8 Lemma: If $\vdash c_1$ **failable**, then $\langle c_1'; c_1; c_2'\rangle\ v_1 \gtrsim\!\!\approx^j e_2 : T$.

Proof: We begin by using type soundness (Theorem 3.8) to find that either $\langle c_1'; c_1; c_2'\rangle\ v_1 \longrightarrow_n^* \langle c_1; c_2'\rangle\ v_1'$, the whole left-hand side reduces to fail (and we are done by Lemma 6.6), or the whole left-hand side diverges (and we are done by definition). In the last two cases we are done, so we consider the first case.

We proceed by induction on $\vdash c_1$ **failable**.

(L_Fail) We step by F_Fail, and have $\mathsf{fail} \gtrsim\!\!\approx^j e_2 : T$ by Lemma 6.6.

(L_BB) We step by F_TagB followed by F_TagBFailB, and then we have $\mathsf{fail} \gtrsim\!\!\approx^j e_2 : T$ by definition; we finish by expansion (Lemma 6.3).

(L_BFun) We step by F_TagB followed by F_TagBFailFun, and we have $\mathsf{fail} \gtrsim\!\!\approx^j e_2 : T$ by Lemma 6.6; we finish by expansion (Lemma 6.3).

(L_FunB) We step by F_TagFun followed by F_TagFunFailB, and we have $\mathsf{fail} \gtrsim\!\!\approx^j e_2 : T$ by Lemma 6.6; we finish by expansion (Lemma 6.3)

□

With ignorable and failable coercions, we can characterize *all* non-canonical coercions, relating them to canonical coercions. The relation $\vdash c_1 \equiv c_2$ relates a non-canonical coercion c_1 to a canonical coercion c_2. Note that this inductively defined relation isn't the same thing as the logical relation. First we show that well typed coercions c in Naive are related to $\mathsf{canonical}(c)$ in Efficient. Then we'll use this general relation to relate *in the logical relation* how coercion forms work on logically related values. We define two rules for relating the tag $D = \{x{:}T \mid e\}$, one which uses $\mathsf{canonical}$ (rule R_DPredCanonical) and one which uses the logical relation (rule R_DPredLR). We'll use the former rule in the first lemma, and the latter rule in the second lemma; the IH on term sizes in the soundness theorem will let us convert the derivations using the first rule to use the second one.

6.9 Lemma [Characterizing non-canonical coercions]: If $\vdash c : T_1 \rightsquigarrow T_2$ then $\vdash c \equiv \mathsf{canonical}(c)$ (using R_DPredCanonical).

Proof: By induction on $\vdash c : T_1 \rightsquigarrow T_2$

(C_Id) By R_Id.

(C_Fail) By R_Fail, L_Fail, and I_Id.

(C_Compose) We know that $\vdash d_1 \equiv \mathsf{canonical}(d_1)$ and $\vdash d_2; \ldots; d_n \equiv \mathsf{canonical}(d_2; \ldots; d_n)$. It remains to show that concatenation on the left is related to merging on the right.

We go by cases on d_1. Throughout the analysis, we will examine the rule used to find $\vdash d_2; \ldots; d_n \equiv \mathsf{canonical}(d_2; \ldots; d_n)$—when the rule was R_Fail, we are immediately done, since the merge will produce **Fail** on the right and the left will always satisfy R_Fail.

($d_1 = B!$) Either $\mathsf{canonical}(d_1) \Downarrow \mathsf{canonical}(d_2; \ldots; d_n)$ begins with $B!$ or it doesn't. If it does, then we invert $\vdash d_2; \ldots; d_n \equiv \mathsf{canonical}(d_2; \ldots; d_n)$:

(R_Id) We are done by R_Composite.

(R_Fail) We are done by R_Fail.

(R_Composite) We are done by R_Composite, with $c_0' = \mathbf{Id}$ and $c_1 = B!$ and $c_1'; \ldots; c_n' = d_2; \ldots; d_n$.

If it doesn't, then we go by inversion of the derivation of $\mathsf{canonical}(d_1) * \mathsf{canonical}(d_2; \ldots; d_n) \Rightarrow \ldots$. We exclude obviously contradictory cases (N_Canonical, N_FailL, N_FunFun, N_FunFailB, N_PredPred, N_PredSame, N_Fun).

(N_FailR) The only relation rule that could have applied is R_Fail, so we are done again by R_Fail.

(N_BB) By R_Composite, since $\vdash B!; c; B?$ **ignorable**.

(N_BFailB) By R_Fail and L_BB.

(N_BFailFun) By R_Fail and L_BFun.

($d_1 = \mathbf{Fun}!$) As for $B!$.

($d_1 = \{x{:}T \mid e\}!$) If d_1 isn't at the beginning of the coercion, then it must be that $\{x{:}T \mid e\}?$ begins $\mathsf{canonical}(d_2; \ldots; d_n)$.

If d_1 is at the beginning, then we go by R_Composite (if R_Id or R_Composite was used to find $\vdash d_2; \ldots; d_n \equiv \mathsf{canonical}(d_2; \ldots; d_n)$) or by R_Fail (if R_Fail was used).

We go by cases on the derivation of $\vdash d_2; \ldots; d_n \equiv \mathsf{canonical}(d_2; \ldots; d_n)$.

(R_Id) Contradictory—we assumed that the final merge result began with $\{x{:}T \mid e\}?$.

(R_Fail) By R_Fail.

(R_Composite) By R_Composite, noting that $\vdash \{x{:}T \mid e\}!; c_0'; \{x{:}T \mid e\}?$ **ignorable** by I_PredSame.

($d_1 = B?$) It must be the case that the merged coercion starts with $B?$, since no merge eliminates $B?$ on the left.

If R_Id or R_Composite was used to find $\vdash d_2; \ldots; d_n \equiv \mathsf{canonical}(d_2; \ldots; d_n)$, we are done by R_Composite. If R_Fail was used, we're done immediately.

($d_1 = \mathbf{Fun}?$) As for $B?$.

($d_1 = \{x{:}T \mid e\}?$) The rule applying in $\{x{:}T \mid \mathsf{canonical}(e)\}? \Downarrow \mathsf{canonical}(d_2; \ldots; d_n)$ must be either N_FailR, N_Canonical, or N_PredPrd. In the first two cases, we are done (by R_Composite/R_Tag/R_DPredCanonical or by R_Fail). In the third case, we are done by the IH with R_Composite and R_Tag, with $c_0' = \mathbf{Id}$. In the fourth case, we are done by the IH (since $\vdash \{x{:}T \mid e\}?; \{x{:}T \mid e\}! \, \mathbf{ignorable}$, and we can use either R_Id or R_Composite with I_Concat).

($d_1 = c_{11} \mapsto c_{12}$) The rule applying in $\mathsf{canonical}(c_{11}) \mapsto \mathsf{canonical}(c_{12}) \Downarrow \mathsf{canonical}(d_2; \ldots; d_n)$ must be one of N_FailR, N_Canonical, or N_Fun.

In the first two cases we are done (by R_Composite/R_Fun or by R_Fail). In the third case, we are done by the IH with R_Composite/R_Fun. In the fourth case, we are done by the IH, noticing that the relation on the right comes from R_Composite with an initial R_Fun, so we can fold $c_{11} \mapsto c_{12}$ into that R_Fun derivation and reconstruct a new R_Composite derivation.

□

We now show that *any* coercions $\vdash c_1 \equiv c_2$ yield related results when applied to related values. We defined the relation $\vdash c_1 \equiv c_2$ because this lemma is easier to prove on the relation than on the $\mathsf{canonical}$ function itself.

6.10 Lemma [Relating canonical coercions]:
If $v_1 \gtrsim^j v_2 : T_1$ and $\vdash c_1 : T_1 \rightsquigarrow T_2$ and $\vdash c_1 \equiv c_2$ (using R_DPredLR), then $\langle c_1 \rangle \; v_1 \gtrapprox^j \langle c_2 \rangle \; v_2 : T_2$.

Proof: By induction on $\vdash c_1 \equiv c_2$ using ignorability (Lemma 6.7) and failability (Lemma 6.8) extensively.

(R_Id) Observe that $T_1 = T_2$. By ignorability (Lemma 6.7), $\langle c_1 \rangle \; v_1 \longrightarrow_n^* \langle \mathbf{Id} \rangle \; v_1$; by F_TagId, $\langle \mathbf{Id} \rangle \; v_1 \longrightarrow_n v_1$. On the right-hand side, E_TagId steps $\langle \mathbf{Id} \rangle \; v_2 \longrightarrow v_2$, and we have $v_1 \gtrsim^j v_2 : T_1$ by assumption. We are done by expansion (Lemma 6.3).

(R_Fail) By failability (Lemma 6.8).

(R_Composite) We want to show $\langle c_0'; c_1; c_1'; c_2; \ldots; c_{n-1}'; c_n; c_n' \rangle \; v_1 \gtrapprox^j \langle d_1; \ldots; d_n \rangle \; v_2 : T_2$ given that $\vdash c_i' \; \mathbf{ignorable}$ and $\vdash c_i \equiv d_i$.

We go by induction on n. The $n = 0$ case is already covered by R_Id above.

So first we use ignorability (Lemma 6.7) to reduce our proof goal to $\langle c_1; c_1'; c_2; \ldots; c_{n-1}'; c_n; c_n' \rangle \; v_1 \gtrapprox^j \langle d_1; \ldots; d_n \rangle \; v_2 : T_2$. We now go by cases on $\vdash c_1 \equiv d_1$ for the first hypothesis.

(R_Fail) By failability (Lemma 6.8).

(R_Tag) By well typing of the LR, the expressions must have the same type $T_1 = B$ (if the tag is B!), $T_1 = ?\to?$ (if the tag is **Fun**!), or $T_1 = \{x{:}T_1 \mid e\}$ (if the tag is $\{x{:}T_1 \mid e\}$!).

Whatever the case, we know by $v_1 \gtrsim^j v_2 : T_1$ that v_1 and v_2 are similarly tagged, so we can step each side by F_TagB/E_Tag, F_TagFun/E_Tag, or F_PredPred/E_Tag. We then finish by the first IH and expansion (Lemma 6.3).

(R_Check) By well typing of the LR, the expressions must have the same type $T_1 = ?$ (if the tag is $B?$ or **Fun**?), $T_1 = B$ (if the tag is $\{x{:}B \mid e\}?$), or $T_1 = ?$ (if the tag is $\{x{:}? \mid e\}?$).

In the first case, we step by F_TagBB/E_Tag, F_TagFunFun/E_Tag, F_TagBFailB/E_TagFail, F_TagBFailFun/E_TagFail, or F_TagFunFailB/E_TagFail on either side, and are done by value relation we have and the first IH and expansion (Lemma 6.3), or by having fail on the left (Lemma 6.6).

In the last two cases, both sides step to checking forms, which we know coterminate at *all* indices (according to our assumption relating refinements using R_DPredLR), and we finish by the first IH.

(R_Fun) We prove by a separate induction that: If

* $v_1 \gtrsim^j u_{1\,\mathbf{Id}} : T_1 \to T_2$,
* $\vdash ((c_{1\,n\,1}; \ldots; c_{111}) \mapsto (c_{112}; \ldots; c_{1\,n\,2})) : (T_1 \to T_2) \rightsquigarrow (T_1' \to T_2')$, and
* $\vdash ((c_{1\,n\,1}; \ldots; c_{111}) \mapsto (c_{112}; \ldots; c_{1\,n\,2})) \equiv c_{21} \mapsto c_{22}$

then $v_{1\,(c_{111} \mapsto c_{112}) \ldots (c_{1\,n\,1} \mapsto c_{1\,n\,2})} \gtrsim^j u_{1\,c_{21} \mapsto c_{22}} : T_1' \to T_2'$. We prove it by induction on n, repeatedly unwrapping the tags on the left-hand side, applying F_Merge, and then the outer IH.

We step through the ignorable coercions (Lemma 6.7), eventually yielding a value such that the second IH relates the stacked function tags on the left to the merged ones on the right. We then finish by expansion (Lemma 6.3).

□

In Greenberg et al. [18], a similar characterization of casts is sufficient. In a standard lambda calculus, we would be able to use the logical relation to reduce e_1 and e_2 to values and then directly apply Lemma 6.10. But that strategy won't work here: reducing those terms may put new coercions on the outside; these extra coercions will merge into c_1 and c_2 (by F_Merge or E_Merge), possibly disrupting $\vdash c_1 \equiv c_2$. Consider the T_Merge case of our proof of soundness. Even if we have $\langle c_1 \rangle\ v_1 \gtrsim^j \langle c_2 \rangle\ v_2 : T$ and $e_1 \longrightarrow_n^* v_1$ and $e_2 \longrightarrow^* v_2$, we can't just put the derivations in and be done: what if e_1 or e_2 produce terms like $\langle c_1' \rangle\ e_1'$ or $\langle c_2' \rangle\ e_2'$ as they evaluate? Then F_Merge or E_Merge will fire, and we won't know anything about the related coercions, nor do we know how many extra steps may have been taken. Accounting for the steps, it turns out, is not particularly hard: if too many new steps are added, the terms are vacuously

in the relation; if not, then we merely need to account for the extra merged coercions.

Having Lemma 6.10 on values doesn't immediately tell us anything about how coercions work on terms; we must prove that separately.

6.11 Lemma [Relating coercions with merges]: If $e_1 \gtrsim\!\!\approx^j e_2 : T_1$ and $\vdash c_1 : T_1 \rightsquigarrow T_2$ and $\vdash c_1 \equiv c_2$ (using R_DPredLR), then $\langle c_1 \rangle\ e_1 \gtrsim\!\!\approx^j \langle c_2 \rangle\ e_2 : T_2$.

Proof: First, we can ignore the cases where $e_1 \longrightarrow_n^m \mathsf{fail}$ or e_1 diverges—those are immediately related, since $\langle c_1 \rangle\ e_1$ also diverges or goes to fail(by Lemma 3.12).

So $e_1 \longrightarrow_n^m v_1$ and $e_2 \longrightarrow^* v_2$, and by definition, $v_1 \gtrsim\!\!\approx^j v_2 : T_1$. By coercion congruence(Lemma 3.12 and Lemma 5.21), there exist e_1' and e_2' such that (a) $\langle c_1 \rangle\ e_1 \longrightarrow_n^* e_1'$ and $\langle c_1 \rangle\ v_1 \longrightarrow_n^* e_1'$, and (b) $\langle c_2 \rangle\ e_2 \longrightarrow^* e_2'$ and $\langle c_2 \rangle\ v_2 \longrightarrow^* e_2'$. But we know that $\langle c_1 \rangle\ v_1 \gtrsim\!\!\approx^j \langle c_2 \rangle\ v_2 : T_2$ by Lemma 6.10, so we are done by contraction and expansion(Lemma 6.4 and Lemma 6.3). □

6.12 Theorem [Soundness]:

- If $\Gamma \vdash v : T$ then $\Gamma \vdash v \gtrsim\!\!\approx \mathsf{canonical}(v) : T$.
- If $\Gamma \vdash e : T$ then $\Gamma \vdash e \gtrsim\!\!\approx \mathsf{canonical}(e) : T$.
- If $\vdash T$ then $\forall j.\ T \gtrsim^j \mathsf{canonical}(T)$.

Proof: By lexicographic induction on the typing derivation and the size of the term (v or e, respectively), using Lemma 6.11 in the T_Coerce case. We use Lemma 6.9 and Lemma 6.10 in the T_TagVal and T_TagValRefine cases.

Let a j be given. In all cases, we begin by letting $\Gamma \models^j \delta$, so we must show that $\delta_1(e) \gtrsim\!\!\approx^j \delta_2(\mathsf{canonical}(e)) : T$.

(T_Var) By definition, $\delta_1(x) \gtrsim\!\!\approx^j \delta_2(x) : T$.

(T_PreVal) By cases on $\Gamma \vdash u : T$:

(T_Const) Immediate—$k_{\mathbf{Id}} \gtrsim^j k_{\mathbf{Id}} : B$ for any j.

(T_Abs) We must show that $\delta_1(\lambda x{:}T_1.\ e_{1\,\mathbf{Id}}) \gtrsim^j \delta_2(\lambda x{:}\mathsf{canonical}(T_1).\ \mathsf{canonical}(e_1)_{\mathbf{Id}}) : T_1 \to T_2$, given that $\Gamma, x{:}T_1 \vdash e_1 : T_2$. Let $m < j$, and let $v_1 \gtrsim^m v_2 : T_1$. We have $\delta_1(\lambda x{:}T_1.\ e_{1\,\mathbf{Id}})\ v_1 \longrightarrow_n \delta_1(e_1)[v_1/x]$ and $\delta_2(\lambda x{:}\mathsf{canonical}(T_1).\ \mathsf{canonical}(e_1)_{\mathbf{Id}})\ v_2 \longrightarrow \delta_2(\mathsf{canonical}(e_1))[v_2/x]$.

We must now show that these two terms are related at m. We can apply the IH on $\Gamma, x{:}T_1 \vdash e_1 : T_2$ at the index m using the closing substitution $\Gamma, x{:}T_1 \models^m \delta[v_1, v_2/x]$.

(T_TagVal) We must show that $\delta_1(v_{1\,d}) \gtrsim^j \delta_2(u_{1\,c \Downarrow \mathsf{canonical}(d)}) : T_2$, where $\mathsf{canonical}(v_1) = u_{1\,c}$ given that $\delta_1(v_1) \gtrsim\!\!\approx^j \delta_2(\mathsf{canonical}(u_1)_c) : T_1$, knowing that $d \neq \{x{:}T_1 \mid e\}$? and $\vdash d : T_1 \rightsquigarrow T_2$

We can apply Lemma 6.9 to find that $\vdash d \equiv \mathsf{canonical}(d)$. By the IH on size we can convert the relation from R_DPredCanonical to

R_DPredLR. We can then apply Lemma 6.10 to find that $\langle d\rangle\ \delta_1(v_1) \gtrsim^j \langle \mathsf{canonical}(d)\rangle\ \delta_2(\mathsf{canonical}(u_1)_c) : T_2$. Since we originally had $v_{1\,d}$, we know that d must be a value tag, we can step both sides (by F_TagB, F_TagFun, F_TagFunWrap or F_TagPredPred on the left; E_Tag or E_Fun on the right) to find that $\langle d\rangle\ \delta_1(v_1) \longrightarrow_n^* \delta_1(v_{1\,d})$ and $\langle \mathsf{canonical}(d)\rangle\ \delta_2(\mathsf{canonical}(u_1)_c) \longrightarrow^* \delta_2(\mathsf{canonical}(u_1)_{c \Downarrow \mathsf{canonical}(d)})$. We are then done by expansion (Lemma 6.3).

(T_TagValRefine) As for the previous case, stepping through related checking forms to find related results. (Or, possibly, by finding fail on the left and ignoring the right entirely, by Lemma 6.6.)

(T_Op) By the IHs and assumption on the behavior of operations.

(T_App) By the IHs and the definition of the logical relation at function types.

(T_Coerce) By Lemma 6.11.

(T_Fail) Immediate by definition.

(T_Check) By the IHs.

(WF_Dyn) Immediate

(WF_Base) Immediate

(WF_Fun) By the IHs.

(WF_Refine) We have $T \gtrsim^j T$ immediately. As the IH on $x{:}T \vdash e : \mathsf{Bool}$, we have $x{:}T \vdash e \gtrapprox \mathsf{canonical}(e) : \mathsf{Bool}$. Let $m < j$ and $v_1 \gtrsim^m v_2 : T$ be given; we can then find $e[v_1/x] \gtrapprox^m \mathsf{canonical}(e)[v_2/x] : \mathsf{Bool}$ by instantiating the IH at m.

□

The definition of $\emptyset \vdash e_1 \gtrapprox e_2 : T$ gives us our approximate observational equivalence: either e diverges, e reduces to fail, or $e \longrightarrow_n^* v_1$ and $\mathsf{canonical}(e) \longrightarrow^* v_2$ such that $v_1 \gtrsim^j v_2 : T$ for arbitrary j. (We write this result without indices because the definition of $\Gamma \vdash e_1 \gtrapprox e_2 : T$ quantifies over all indices.) Note that for base values, we have exactly the same result on both sides.

7 Space efficiency

The structure of our space-efficiency proof is largely the same as in prior work. Coercion size is broken down by the order of the types involved; the maximum size of any coercion is $|\text{largest coercion}| \cdot 2^{\text{tallest type}}$. Inspecting the canonical coercions, the largest is $\{x{:}? \mid e\}!; \mathbf{Fun}?; c_1 \mapsto c_2; \mathbf{Fun}!; \{x{:}? \mid e'\}?$, with a size of 5. The largest possible canonical coercion therefore has size $M = 5 \cdot 2^h$.

Formally, observe that merging canonical coercions c_1 and c_2 either produces a smaller coercion or $c_1; c_2$ is canonical(and has size $\text{size}(c_1) + \text{size}(c_2)$).

7.1 Lemma [Merge reduces size]: If $c_1 * c_2 \Rightarrow c_3$, then either $\text{size}(c_1) + \text{size}(c_2) > \text{size}(c_3)$, or $c_3 = c_1; c_2$ is canonical.

Proof: By induction on the derivation of $c_1 * c_2 \Rightarrow c_3$. Note that in either case, $\text{size}(c_1) + \text{size}(c_2) \geq \text{size}(c_3)$.

(N_Canonical) $c_1; c_2$ is canonical.

(N_FailL) We have $\mathbf{Fail} * c_2 \Rightarrow \mathbf{Fail}$, with $1 + \text{size}(c_2) > 1$.

(N_FailR) We have $c_1 * \mathbf{Fail} \Rightarrow \mathbf{Fail}$, with $\text{size}(c_1) + 1 > 1$.

(N_BB) We have $\vdash c_1; B! : T_1 \rightsquigarrow ?$ and $\vdash B?; c_2 : ? \rightsquigarrow T_3$. By the IH, $\text{size}(c_1) + \text{size}(c_2) \geq \text{size}(c_3)$, so we immediately have $\text{size}(c_1) + 1 + \text{size}(c_2) + 1 > \text{size}(c_3)$.

(N_BFailB) We have $\vdash c_1; B! : T_1 \rightsquigarrow ?$ and $\vdash B'?; c_2 : ? \rightsquigarrow T_3$. It is immediate that size(**Fail**) is smaller.

(N_BFailFun) We have $\vdash c_1; B! : T_1 \rightsquigarrow ?$ and $\vdash \mathbf{Fun}?; c_2 : ? \rightsquigarrow T_3$. It is immediate that size(**Fail**) is smaller.

(N_FunFun) We have $\vdash c_1; \mathbf{Fun}! : T_1 \rightsquigarrow ?$ and $\vdash \mathbf{Fun}?; c_2 : ? \rightsquigarrow T_3$. By the IH, $\text{size}(c_1) + \text{size}(c_2) \geq \text{size}(c_3)$, so we immediately have $\text{size}(c_1) + 1 + 1 + \text{size}(c_2) \geq \text{size}(c_3)$.

(N_FunFailB) We have $\vdash c_1; \mathbf{Fun}! : T_1 \rightsquigarrow ?$ and $\vdash B?; c_2 : ? \rightsquigarrow T_3$. It is immediate that size(**Fail**) is smaller.

(N_PredPred) We have $\vdash c_1; \{x{:}T \mid e\}? : T_1 \rightsquigarrow \{x{:}T \mid e\}$ and $\vdash \{x{:}T \mid e\}!; c_2 : \{x{:}T \mid e\} \rightsquigarrow T_3$, where T is either B or $?$. By the IH, $\text{size}(c_1) + \text{size}(c_2) \geq \text{size}(c_3)$, so we immediately have $\text{size}(c_1) + 1 + 1 + \text{size}(c_2) \geq \text{size}(c_3)$.

(N_PredSame) We have $\vdash c_1; \{x{:}T \mid e\}! : T_1 \rightsquigarrow T$ and $\vdash \{x{:}T \mid e\}!; c_2 : T \rightsquigarrow T_3$, where T is either B or $?$. By the IH, $\text{size}(c_1) + \text{size}(c_2) \geq \text{size}(c_3)$, so we immediately have $\text{size}(c_1) + 1 + 1 + \text{size}(c_2) \geq \text{size}(c_3)$.

(N_Fun) We have $\vdash c_1; (c_{11} \mapsto c_{12}) : T_1 \rightsquigarrow (T_{21} \to T_{22})$ and $\vdash (c_{21} \mapsto c_{22}); c_2 : (T_{21} \to T_{22}) \rightsquigarrow T_3$. By the IH:

$$\begin{array}{rcl} \text{size}(c_{21}) + \text{size}(c_{11}) & \geq & \text{size}(c_{31}) \\ \text{size}(c_{12}) + \text{size}(c_{22}) & \geq & \text{size}(c_{33}) \\ & \text{so} & \\ \text{size}(c_{11} \mapsto c_{12}) + \text{size}(c_{21} \mapsto c_{22}) & = & 1 + \text{size}(c_{11}) + \text{size}(c_{12}) + \\ & & 1 + \text{size}(c_{21}) + \text{size}(c_{22}) \\ & > & 1 + \text{size}(c_{31}) + \text{size}(c_{32}) \end{array}$$

Finally, $\text{size}(c_1) + \text{size}((c_{31} \mapsto c_{32}); c_2) \geq \text{size}(c_3)$. So we finally conclude that $\text{size}(c_1; (c_{11} \mapsto c_{12})) + \text{size}((c_{21} \mapsto c_{22}); c_2) > \text{size}(c_3)$.

□

Rules with merges (and E_Merge in particular) don't increase the size of the largest coercion in the program. Applying this lemma across an evaluation $e \longrightarrow^* e'$, we can see that no coercion ever exceeds the size of the largest coercion in e. If M is the size of the largest coercion, then there is at most an M-fold space overhead of coercions. But this size bound is galactic; we find it hard to believe that this overhead is observable in practice. A much more interesting notion of space efficiency—not studied here—is to determine implementation schemes for space-efficient layout of coercions in memory and time-efficient merges of coercions. We believe that explicitly enumerating the canonical coercions is a step towards this goal: the canonical coercions in Table 1 are exactly those which must be represented.

8 Design philosophy

Our design philosophy has three principles:

1. Base values have simple types; e.g., all integers are typed at Int.
2. We give operations types precise enough to guarantee totality; e.g., division has a type at least as precise as $\mathsf{Int}{\to}\{x{:}\mathsf{Int} \mid x \neq 0\}{\to}\mathsf{Int}$.
3. Values satisfy their refinement types; e.g., if $\emptyset \vdash v : \{x{:}T \mid e\}$, then $e[v/x] \longrightarrow^* \mathsf{true}$.

Giving base values simple types by avoids some of the technical problems seen in previous refinement type systems that gave constants most specific types [32, 14, 27, 26, 18]. These other systems set $\mathsf{ty}(k)$ such that if $e[k/x] \longrightarrow^* \mathsf{true}$, then $\mathsf{ty}(k) <: \{x{:}T \mid e\}$. We wish to avoid subtyping—and its concomitant circularities [18, 26, 3]. Beyond technical issues, simple types are a de facto default in functional programming; Hindley-Milner is a sweet spot in the design space. We believe that taking this sweet spot as the default programming paradigm will increase the usability and practicality of our designs.

There is one exception to assigning values simple types: it makes sense to give null or $\mathsf{undefined}$ a dynamic type; attempting to coerce such a value to a simple type would result in an error. We omit such values—it's not clear that they're a good idea [24].

To us, keeping operations total is the main goal of refinement types. In fact, the original use of refinement types by Freeman and Pfenning [16] was to make partial pattern matching total. Our calculi don't have abstract datatypes, but partial operations are just as undesirable as partial pattern matches. This point of view is nice for implementations: primitive partial operations can be exposed at types strong enough to guarantee safety; the coercion algorithm will insert appropriate checks which may or may not be optimized away. (This is *hybrid type checking* [14].)

The flipside of using refinements to protect partial operations is that refinement checks that don't end up protecting partial operations aren't as important as those that do. In Efficient in Section 5, we occasionally skip checking

whether or not a value satisfies a refinement. We only do this when the program's next step would be to untag the value as having satisfied the refinement—why bother checking if we don't care about the result? This lenient approach to refinement type checking means that our naïve and space-efficient calculi don't behave exactly the same: some programs raise errors in the former but not the latter, precisely because we skip checks.

One might ask: if it's okay to skip refinement type checks that are going to be immediately ignored, why not do the same for checks when moving into and out of dynamic types? There are two differences, one technical and one philosophical. As a technical difference, we don't skip a **Fun**?; **Fun**! step (read "untag from dynamic as a function, retag to dynamic as a function") because $u_{B!}$ and $u_{\mathbf{Fun}!}$ are really different values operationally, and there are evaluation rules that look at the tags; the tags for refinement types are a technical device for our calculus, and need not be implemented: no evaluation rule meaningfully looks at refinement tags, and the type system guarantees that any value tagged at $\{x{:}T \mid e\}?$ satisfies its refinement. As a philosophical difference, we hold the line at the structure of values, i.e., simple types. If simple types are the default paradigm of our language, we want to avoid errors in this fragment as much as possible. As Siek and Wadler [38] say:

> [the] boundary between static and dynamic typing regions require[s] certain run-time checks to maintain the integrity of the static region.

The refined world has the structure of the simply typed world plus a predicate—it has more type information. On the other hand, dynamic types have less information, so we must be stricter with them. Put another way, simple types are about the broad structure of values, while refinements are about the safety of operations. If we never end up running the operation whose safety is ensured by a refinement type, no problem; the programmer's specification may be too tight. Polymorphism aside, there is no such thing as a simple type that is "too tight".

Our third principle is a compromise between stringency and lenience. If a value is typed at a refinement type, then it satisfies its predicate. Value inversion is a compromise because some checks can be skipped (lenience), but once the program reduces to a value it must actually inhabit its type (stringency). This *value inversion* principle is valuable, and one that we have used in practice when programming with refinement types. In call-by-value (CBV) languages (such as both of our calculi), value inversion lets programmers reason easily about functions which take refined inputs.

Value inversion is the *least* that a refinement type can mean in a sound system: any less and type soundness won't hold. While every sound refinement type system has value inversion, we find it desirable to be able to directly invert the typing derivation to find that a value satisfies the predicate of its refinement type, rather than applying a more complicated or general reasoning principle. We believe that having value inversion directly is easier for programmers to understand and use.

9 Related work

There are two related threads of work: a more recent line of work on gradual types, refinement types, and full-spectrum programming languages; and an older, more general line of work on coercions, which may or may not have runtime semantics. Space efficiency and representation have been studied in both settings.

Space efficiency, gradual typing, and refinement types

In Siek and Taha's seminal work on gradual typing [37], space efficiency is already a concern—they point out that the canonical forms lemma has implications for which values can be unboxed (the typed ones). Herman, Tomb, and Flanagan [21] compiled a language like Siek and Taha's into a calculus with Henglein's coercions [20], proving a space-efficiency result with a galactic bound similar to ours. Herman et al. stop at proving that their compilation is type preserving without proving soundness of their compilation. (We compare our system to Herman et al.'s in greater detail below.) Siek, Garcia, and Taha [36] explore the design space around Herman et al.'s result, this time with an observational equivalence theorem exactly relating two coercion semantics.

Siek and Wadler [38] study an alternative, cast-based formulation of space efficiency, proving tighter bounds than Herman et al. [21] and an exact observational equivalence. Their insight is that casts can be factored not merely as a "twosome" $\langle S \Rightarrow T \rangle$, but rather as a threesome: $\langle S \overset{R}{\Rightarrow} T \rangle$. They maintain the invariant that S downcasts to R, and R upcasts to T; merging casts amounts to calculating a greatest lower bound. They come up with an elegant theory of merging casts, with a detailed accounting for blame. While the mathematics is beautiful, we believe that their algorithm is overkill: Herman et al.'s journal article [22] cleanly enumerates the recursive structure of the canonical coercions for dynamic and simple types, with only 17 possible structures at the top level. Siek and Wadler's theory is the theory of these 17 structures. Many of the solutions can be simply pre-computed and looked up in a table at runtime. We have 37 canonical coercions. We don't study the question here, but we believe that a careful analysis would allow for very compact representations with very fast merges—by pointer comparison and table lookup when functional coercions aren't involved. We discuss this issue further in future work (Section 10).

Before considering other full-spectrum languages, we compare our work to the most closely related work: Herman, Tomb, Flanagan [21, 22] and Henglein [20]. Henglein is trying to reason carefully about programs written in a dynamic style, rather than thinking about multi-paradigm programming (though it is clear that he knows that his work applies to "dynamic typing in a static language"). His theory of coercions has no **Fail** coercion and treats **Id** slightly differently at function types. Herman et al. adapt his calculus to match the setting of gradual types, though they never rebuild his theory. Henglein develops a general theory *characterizing* canonical coercions, but we *enumerate* them.

Perhaps the biggest difference is that Henglein and Herman et al. formulate

coercions as having arbitrary composition: $c_1; c_2$ is a coercion that can be used freely. As a consequence, it is somewhat difficult to reason directly about coercions in the calculus: what should $\langle (c_1 \mapsto c_2; \mathbf{Fun!}); \{x{:}? \mid e\}? \rangle\ v$ do? Their solution is to work with coercions up to an equivalence relation that includes associativity of coercion composition; coercions normalize in a term rewriting system modulo this equivalence relation. Henglein studies some algorithmic rewriting systems. But Herman et al. don't develop the rewriting system at all, never showing that their rewriting system is strongly normalizing, and even when they enumerate canonical coercions in their journal version [22], they do so without proof. We feel that term rewriting modulo equational theories is somewhat insufficient for guiding an implementation of a coercion calculus: the compiler needs a concrete representation for coercions and a concrete algorithm for merging them. We accordingly adopt a constrained form of coercion composition out of a desire to aid implementation, but also out of expedience: we don't need to worry about associativity at all. We don't believe that free composition buys anything, anyway: we don't expect programmers to be writing coercions by hand, so ease of expression in the coercion language isn't particularly important.

Finally, Herman et al.'s calculus is a little odd: the value $\langle \mathbf{Id} \rangle\ v$ takes a step to v. Our approach makes a clearer distinction between terms and the results that they produce. Siek and Wadler noticed a separate problem with nondeterminism at cast merges, most likely due to a mistake in defining evaluation contexts.

The work discussed so far consisted of calculi devised expressly for space-efficient gradual typing. Findler et al. [13] discuss space efficiency from the perspective of an implementation in PLT Racket (then PLT Scheme). Their setting—latent contracts, no type system—is rather different from the foregoing systems; they address datatypes, while the foundational calculi omit datatypes. We believe it would be very interesting future work to try to combine how our refinement types protect partial operations (space efficiently) with classic refinements of datatypes [16] protecting partial matches (also space efficiently).

Considering the world of full-spectrum programming languages more broadly, we summarize existing solutions. None of the following are space efficient; we are the first to combine space efficiency, gradual types, and refinement types. Ou et al. [32] cover the spectrum and include dependent types, but allow only a constrained set of refinement predicates; Sage [27] covers the entire spectrum and also includes dependent types, but lacks a soundness proof; Wadler and Findler's [42] development covers dynamic types through refinements of base types; Bierman et al. [4] cover the whole spectrum but (also with dependency) only for first-order types.

Coercions

There are many other systems that use coercions to other ends. Henglein gives an excellent summary of work up to 1994 in the related work section of his article [20]. One of the classic uses of coercions is subtyping [5, 30]; more re-

cent work relates subtyping and polymorphism [8]. Work on unboxing [35, 31] confronts similar issues of space efficiency. Many of these examples carefully ensure that coercions are erasable, while our coercions are definitely not. Henglein, whose work is the closest to ours, develops a rich theory of coercions. For our purposes, such a theory is overkill: his general characterization of canonical coercions is beautiful, but our table is all that we need.

Swamy, Hicks, and Bierman [39] study coercion insertion in general, showing that their framework can encode gradual types. We haven't studied coercion insertion at all, though Swamy et al.'s framework would be a natural one to use. We are not aware of work on how coercion insertion algorithms affect space consumption.

10 Future work

The obvious next step is adding blame [12]. Siek and Wadler [38] are the only space-efficient calculus to have blame, which they obtain with some effort—their threesome merging takes place outside in, making it hard to compute which label to blame. We conjecture that our inside-out coercion merge algorithm offers an easy way to compute blame: blame comes from left to right.

Another interesting point of study would relate coercion insert [39] to space efficiency.

Extending the calculus to general refinements, where any type T can be refined to $\{x{:}T \mid e\}$, would be a challenging but important step towards adding polymorphism. (You can't allow refinement of type variables unless *any* type can be refined, since there's no way to know what type will be substituted in for the variable.) It wouldn't be too difficult to add function refinements $\{x{:}(T_1 \to T_2) \mid e\}$ to this calculus, but refinements of refinements seem to break space efficiency: if $\{x{:}B \mid e\}?$ is canonical, so is $\{x{:}B \mid e\}?; \{x{:}\{x{:}B \mid e\} \mid e'\}?$—there are an infinite number of canonical coercions. In a monomorphic calculus, the number of canonical coercions can be bounded by the types in the original program, but not so in a polymorphic calculus. Prior work relating dynamic types and polymorphism will apply here [28, 2].

Other systems have treated failure more eagerly, e.g., $\mathbf{Fail} \mapsto \mathbf{Id} * c \Rightarrow \mathbf{Fail}$. This would further disrupt the connection between the naïve and space-efficient semantics.

Adding dependent functions to the coercion calculus above would complicate matters significantly, but would also add a great deal of expressiveness. Adding the type $(x : T_1) \to T_2$ is straightforward enough: we should be able to prove type soundness using entirely syntactic techniques, adapting work on FH, a polymorphic calculus with manifest contracts and general refinements [3]. Designing the coercions is a challenge, though. Dependency means that in the coercion $(x{:}c_1) \mapsto c_2$, the variable x is bound in c_2. Coercion well formedness now needs a context to keep track of such bound variables:

$$\frac{\Gamma \vdash c_1 : T_{11} \rightsquigarrow T_{21} \qquad \Gamma, x{:}T_{11} \vdash c_2 : T_{12} \rightsquigarrow T_{22}}{\Gamma \vdash (x{:}c_1) \mapsto c_2 : ((x{:}T_{21}) \to T_{12}) \rightsquigarrow ((x{:}T_{11}) \to T_{22})}$$

How do dependent functions and their corresponding coercions affect coercion normalization? The following structural equivalence rule, derived from the for dependent functions looks improbable, though its asymmetry echoes the asymmetry of the dependent function cast rule in FH and other manifest calculi.

$$((x{:}c_1) \mapsto c_2); ((y{:}c_1') \mapsto c_2') = \\ (y{:}(c_1'; c_1)) \mapsto (c_2\{\langle c_1' \rangle\ y/x\}; c_2')$$

The metatheory surrounding dependent functions in coercion calculi will be difficult. In fact, the rule above isn't obvious: trying to use $x{:}T_{21}$ as the binding for x will raise difficulties in typing the equational rule.

Programs with dependent types have a potentially infinite set of types (and, so, coercions) which may appear as the program evaluates. In the type $(x : \mathsf{Real}) \to \{y : \mathsf{Real} \mid \left|x - y^2\right| < \epsilon\}$, there are potentially infinitely many different codomain types: one for each Real value of x. But even with an infinite number of possible coercions, the set of canonical coercions shouldn't change (beyond the addition of dependency to functions).

Set semantics for refinement types extend refinement types to have a set of predicates, rather than a single one. The N_PredSame merge rule skips a check when we would have projected out of and then back into a refinement type. If refinements were sets, we could broaden this optimization to allow the space-efficient calculus to avoid even more redundant checks.(This was, in fact, the original motivation for this work.)

In an extension of the system of Section 5 with so-called "general" refinements [3], the coercion $\{x{:}\mathsf{Int} \mid x > 0\}?; \{x{:}\{x{:}\mathsf{Int} \mid x > 0\} \mid x > 5\}?$ is well typed but $\{x{:}\mathsf{Int} \mid x > 0\}?; \{x{:}\mathsf{Int} \mid x > 5\}?$ isn't. Furthermore, $\{x{:}\{x{:}\mathsf{Int} \mid x > 0\} \mid x > 5\}?$ and $\{x{:}\{x{:}\mathsf{Int} \mid x > 5\} \mid x > 0\}?$ are totally different coercions, even though the underlying predicates in the refinements are the same. Treating a refinement as being flat with a single set of predicates, rather than a tower of separate refinements, would resolve these ordering issues. That is, we could have a single type that represents both orderings of refinement: $\{x{:}\mathsf{Int} \mid \{x > 5, x > 0\}\}$. Casts into and out of refinement types could be simplified to adding and removing refinements from the set. When a value is coerced into a refinement type with a set of predicates, the type system remembers all of the predicates equally, acting as a cache of multiple satisfied contracts. The utility of the set semantics is that helps address the library problem: when writing a list library, what refinements are important—emptiness, sortedness, length? When refinements are treated as sets of predicates, libraries can deal only with their own predicates, treating extra client predicates parametrically.

For example, remembering that a list is both non-empty and sorted might be useful for a sorted-list representation of sets. When the set code needs to take the head of a list (which happens to be a minimal member of the set), it can do so directly. Similarly, when calling the $\mathsf{insertSorted}$ function to add an element to the set, it knows that both its original representation and the extended one are still valid, sorted representations.

Another example where the set semantics allows multiple predicates to interact is a value $v : \{x{:}\mathsf{Int} \mid \{x \neq 0, \mathsf{prime}\ x\}\}$. Since $v \neq 0$, we can use it as the

divisor with $\mathsf{div} : \mathsf{Int} \to \{x{:}\mathsf{Int} \mid x \neq 0\} \to \mathsf{Int}$; since $\mathsf{prime}\ v$, we can use it as half of a private key.

Set semantics was my initial motivation for revisiting space-efficient coercions: we were interested in ways of remembering contract checks on values to reduce redundant checking.

11 Conclusion

Acknowledgments

We thank Stephanie Weirich and Benjamin Pierce for looking over drafts, and also for their excellent advice and support. Brent Yorgey contributed to some discussions. Our work has been supported by the National Science Foundation under grant 0915671 *Contracts for Precise Types*. This material is based upon work supported by the DARPA CRASH program through the United States Air Force Research Laboratory (AFRL) under Contract No. FA8650-10-C-7090. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. Finally, the support of Hannah de Keijzer and the patience of the staff of B2 are greatly appreciated.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Principles of Programming Languages (POPL)*, 1989.

[2] A. Ahmed, R. B. Findler, J. Siek, and P. Wadler. Blame for all. In *Principles of Programming Languages (POPL)*, 2011.

[3] J. F. Belo, M. Greenberg, A. Igarashi, and B. C. Pierce. Polymorphic contracts. In *European Symposium on Programming (ESOP)*, 2011.

[4] G. M. Bierman, A. D. Gordon, C. Hriţcu, and D. Langworthy. Semantic subtyping with an SMT solver. In *International Conference on Functional Programming (ICFP)*, 2010.

[5] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172 – 221, 1991. ISSN 0890-5401. doi:http://dx.doi.org/10.1016/0890-5401(91)90055-7. Selections from 1989 {IEEE} Symposium on Logic in Computer Science.

[6] O. Chitil and F. Huch. Monadic, prompt lazy assertions in haskell. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2007.

[7] R. Chugh, P. M. Rondon, and R. Jhala. Nested refinements: a logic for duck typing. In *Principles of Programming Languages (POPL)*, POPL '12, pages 231–244, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. doi:10.1145/2103656.2103686.

[8] J. Cretin and D. Rémy. On the power of coercion abstraction. In *Principles of Programming Languages (POPL)*, POPL '12, pages 361–372, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. doi:10.1145/2103656.2103699.

[9] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: no more scapegoating. In *Principles of Programming Languages (POPL)*, 2011. doi:10.1145/1926385.1926410.

[10] T. Disney and C. Flanagan. Gradual information flow typing. In *Workshop on Script-to-Program Evolution (STOP)*, 2011.

[11] R. B. Findler. Contracts as pairs of projections. In *Symposium on Logic Programming*, 2006.

[12] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*, 2002.

[13] R. B. Findler, S.-Y. Guo, and A. Rogers. Implementation and application of functional languages. chapter Lazy Contract Checking for Immutable Data Structures, pages 111–128. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-85372-5. doi:10.1007/978-3-540-85373-2_7.

[14] C. Flanagan. Hybrid type checking. In *Principles of Programming Languages (POPL)*, 2006.

[15] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Programming Language Design and Implementation (PLDI)*, 2002.

[16] T. Freeman and F. Pfenning. Refinement types for ML. In *Programming Language Design and Implementation (PLDI)*, June 1991.

[17] A. D. Gordon and C. Fournet. Principles and applications of refinement types. Technical Report MSR-TR-2009-147, 2008.

[18] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *Principles of Programming Languages (POPL)*, 2010.

[19] J. Gronski and C. Flanagan. Unifying hybrid types and contracts. In *Trends in Functional Programming (TFP)*, 2007.

[20] F. Henglein. Dynamic typing: Syntax and proof theory. *Sci. Comput. Program.*, 22(3):197–230, 1994.

[21] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. In *Trends in Functional Programming (TFP)*, pages 404–419, Apr. 2007.

[22] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. *Higher Order Symbol. Comput.*, 23(2):167–189, June 2010. ISSN 1388-3690. doi:10.1007/s10990-011-9066-z.

[23] R. Hinze, J. Jeuring, and A. Löh. Typed contracts for functional programming. In *Functional and Logic Programming (FLOPS)*, 2006.

[24] T. Hoare. Null references: The billion dollar mistake. QCon talk, 2009.

[25] C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your ifcexception are belong to us. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 3–17, 2013. doi:10.1109/SP.2013.10.

[26] K. Knowles and C. Flanagan. Hybrid type checking. To appear in TOPLAS., 2010.

[27] K. Knowles, A. Tomb, J. Gronski, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, 2006.

[28] J. Matthews and A. Ahmed. Parametric polymorphism through run-time sealing or, theorems for low, low prices! In *European Symposium on Programming (ESOP)*, 2008.

[29] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., 1992. ISBN 0-13-247925-7.

[30] Y. Minamide. Runtime behavior of conversion interpretation of subtyping. In *Selected Papers from the 13th International Workshop on Implementation of Functional Languages*, IFL '02, pages 155–167, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43537-9.

[31] Y. Minamide and J. Garrigue. On the runtime complexity of type-directed unboxing. In *International Conference on Functional Programming (ICFP)*, ICFP '98, pages 1–12, New York, NY, USA, 1998. ACM. ISBN 1-58113-024-4. doi:10.1145/289423.289424.

[32] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *IFIP Conference on Theoretical Computer Science (TCS)*, 2004.

[33] A. M. Pitts. Typed operational reasoning. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 7, pages 245–289. The MIT Press, 2005. ISBN 0-262-16228-8.

[34] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Programming Language Design and Implementation (PLDI)*, 2008.

[35] Z. Shao. Flexible representation analysis. In *International Conference on Functional Programming (ICFP)*, pages 85–98, Amsterdam, The Netherlands, June 1997.

[36] J. Siek, R. Garcia, and W. Taha. Exploring the design space of higher-order casts. In G. Castagna, editor, *Programming Languages and Systems*, volume 5502 of *Lecture Notes in Computer Science*, pages 17–31. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-00589-3. doi:10.1007/978-3-642-00590-9_2.

[37] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, September 2006.

[38] J. G. Siek and P. Wadler. Threesomes, with and without blame. In *Principles of Programming Languages (POPL)*, pages 365–376, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9. doi:10.1145/1706299.1706342.

[39] N. Swamy, M. Hicks, and G. M. Bierman. A theory of typed coercions and its applications. In *International Conference on Functional Programming (ICFP)*, pages 329–340, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. doi:10.1145/1596550.1596598.

[40] S. Thatte. Quasi-static typing. In *Principles of Programming Languages (POPL)*, 1990.

[41] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *Principles of Programming Languages (POPL)*, 2008.

[42] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *European Symposium on Programming (ESOP)*, 2009.

Ignorable coercions

$$\frac{}{\vdash \mathbf{Id}\,\mathbf{ignorable}}\ \text{I\_Id} \qquad \frac{}{\vdash \mathbf{Fail}\,\mathbf{ignorable}}\ \text{I\_Fail} \qquad \frac{\vdash c\,\mathbf{ignorable}}{\vdash B!; c; B?\,\mathbf{ignorable}}\ \text{I\_BB}$$

$$\frac{\vdash c\,\mathbf{ignorable}}{\vdash \mathbf{Fun}!; c; \mathbf{Fun}?\,\mathbf{ignorable}}\ \text{I\_FunFun} \qquad \frac{}{\vdash \{x{:}T \mid e\}?; \{x{:}T \mid e\}!\,\mathbf{ignorable}}\ \text{I\_PredPred}$$

$$\frac{\vdash c\,\mathbf{ignorable}}{\vdash \{x{:}T \mid e\}!; c; \{x{:}T \mid e\}?\,\mathbf{ignorable}}\ \text{I\_PredSame} \qquad \frac{\vdash c_1\,\mathbf{ignorable} \quad \vdash c_2\,\mathbf{ignorable}}{\vdash c_1; c_2\,\mathbf{ignorable}}\ \text{I\_Concat}$$

Failable coercions

$$\frac{}{\vdash \mathbf{Fail}\,\mathbf{failable}}\ \text{L\_Fail} \qquad \frac{B \neq B' \quad \vdash c\,\mathbf{ignorable}}{\vdash B!; c; B'?\,\mathbf{failable}}\ \text{L\_BB}$$

$$\frac{\vdash c\,\mathbf{ignorable}}{\vdash B!; c; \mathbf{Fun}?\,\mathbf{failable}}\ \text{L\_BFun} \qquad \frac{\vdash c\,\mathbf{ignorable}}{\vdash \mathbf{Fun}!; c; B?\,\mathbf{failable}}\ \text{L\_FunB}$$

Figure 22: Ignorable and failable coercions