Space-Efficient Manifest Contracts

Michael Greenberg
University of Pennsylvania
mgree@seas.upenn.edu

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 [20, 21, 33, 35] 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. [20] and Henglein's work [19], 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.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.2.4 [*Software Engineering*]: Software/Program Verification—Programming by contract

General Terms Languages, Theory

Keywords Contract, refinement type, dynamic checking, precondition, postcondition, space efficiency, coercion, dynamic typing, gradual typing

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. 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 the Siek and Taha's rubric of "gradual typing" [1, 2, 7, 10, 20, 26, 33–35, 37, 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 [3, 11, 14, 16–18, 27, 31]. 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, 25, 30, 39]. Starting with Sage [25] and continuing with Wadler and Findler [39], many languages have expressed the interactions between dynamic and simple and between simple 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".

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 [3, 17, 24, 33, 35, 39]). 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 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.[1]

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.

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 $\langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{prime}\ x\} \rangle$. 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 [3, 24].)

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 from Herman et al. [20]:

```
even : ?→? = ⟨Int→Bool ⇒ ?→?⟩ λx:Int.
  if x = 0 then true else odd (x − 1)
odd : Int→Bool = λx:Int.
  if x = 0 then false else (⟨?→? ⇒ Int→Bool⟩ even) (x − 1)
```

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 1, 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's more, the work is redundant: we must tag and untag true twice. In short, casts have taken an algorithm that 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 [20, 33, 35] factor casts into their constituent *coercions*, adapting Henglein's system [19]. We will take a similar

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

```
    odd 3
⟶ (⟨?→? ⇒ Int→Bool⟩ even) 2
⟶ ⟨? ⇒ Bool⟩ (even (⟨Int ⇒ ?⟩ 2))
⟶ ⟨? ⇒ Bool⟩ (even 2Int!)
⟶ ⟨? ⇒ Bool⟩ ((⟨Int→Bool ⇒ ?→?⟩ (λx:Int. ...)) 2Int!)
⟶ ⟨? ⇒ Bool⟩ ((⟨Bool ⇒ ?⟩ ((λx:Int. ...) (⟨? ⇒ Int⟩ 2Int!)))
⟶ ⟨? ⇒ Bool⟩ ((⟨Bool ⇒ ?⟩ ((λx:Int. ...) 2))
⟶ ⟨? ⇒ Bool⟩ ((⟨Bool ⇒ ?⟩ (odd 1))
⟶ ⟨? ⇒ Bool⟩ ((⟨Bool ⇒ ?⟩ ((⟨?→? ⇒ Int→Bool⟩ even) 0))
⟶ ⟨? ⇒ Bool⟩ ((⟨Bool ⇒ ?⟩ (⟨? ⇒ Bool⟩ (even (⟨Int ⇒ ?⟩ 0))))
⟶ ⟨? ⇒ Bool⟩ ((⟨Bool ⇒ ?⟩ (⟨? ⇒ Bool⟩ (even 0Int!)))
⟶ ⟨? ⇒ Bool⟩ ((⟨Bool ⇒ ?⟩ (⟨? ⇒ Bool⟩
     ((⟨Int→Bool ⇒ ?→?⟩ (λx:Int. ...)) 0Int!)))
⟶ ⟨? ⇒ Bool⟩ ((⟨Bool ⇒ ?⟩ (⟨? ⇒ Bool⟩ (⟨Bool ⇒ ?⟩
     ((λx:Int. ...) (⟨? ⇒ Int⟩ 0Int!)))))
⟶ ⟨? ⇒ Bool⟩ ((⟨Bool ⇒ ?⟩ (⟨? ⇒ Bool⟩ (⟨Bool ⇒ ?⟩
     ((λx:Int. ...) 0))))
⟶ ⟨? ⇒ Bool⟩ ((⟨Bool ⇒ ?⟩ (⟨? ⇒ Bool⟩ (⟨Bool ⇒ ?⟩ true)))
⟶ ⟨? ⇒ Bool⟩ ((⟨Bool ⇒ ?⟩ (⟨? ⇒ Bool⟩ trueBool!))
⟶ ⟨? ⇒ Bool⟩ (⟨Bool ⇒ ?⟩ true)
⟶ ⟨? ⇒ Bool⟩ trueBool!
⟶ true
```

Figure 1. Space-inefficient reduction

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 Bool?; the cast $\langle \mathsf{Bool} \Rightarrow ? \rangle$, which tags a boolean into type dynamic, is written 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 Bool!; Bool? into the no-op coercion **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 [20] 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 **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 3; we show that this means that space-efficient programs fail less often than their naïve counterparts in Section 4.

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 3). In particular, earlier systems have given either term rewriting systems modulo equational theories, which lack straightfoward implementations [19–21, 33], or they have given complicated algorithms for merging casts [35]. We rework the definition of coercions to admit a straightforward term rewriting system, for which we enumerate

the exact set of canonical coercions. We then define a straightforward merge operator on canonical coercions, offering a simpler and clearer theory.

- 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 2 and Section 3). This new language narrowly edges out Wadler and Findler [39] 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 5).
- 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 4). Of the prior work, only Siek and Wadler [35] prove a similar soundness theorem, showing that their space-efficient gradual typing calculus is (exactly) equivalent to their original, naïve calculus.

2. A naïve coercion calculus

In this section, we define a naïve coercion calculus; 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 [35] establish a more interesting one, relating their space-efficient threesome casts and a coercion calculus.)

Our language adheres to a design philosophy of "simply 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 two 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 [15]); giving operations types that make them total means that our reasoning about runtime errors can entirely revolve around cast (here, coercion) failures.

Before we begin our technical work in earnest, a word about conventions. We are defining two calculi, and they largely share syntax and typing rules (of the form T_NAME), relying on context to differentiate terms. The evaluation rules for the naïve calculus in Section 2 are named F_NAME, using a subscripted arrow $\longrightarrow_n$ for the reduction relation; the space-efficient evaluation rules in Section 3 are named E_NAME using a plain arrow $\longrightarrow$.

2.1 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 values with tags.

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

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

Typing contexts

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

Figure 2. Naïve syntax

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 3.1, 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 7).

Well formedness of types and contexts is defined straightforwardly in Figure 3. It is worth noting, however, that well formedness of refinements refers back to the term typing judgment.

Term typing (also defined in Figure 3) 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 , the rule for constants gives them base types: $\mathsf{ty}(k) = B$. We require that no constant 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

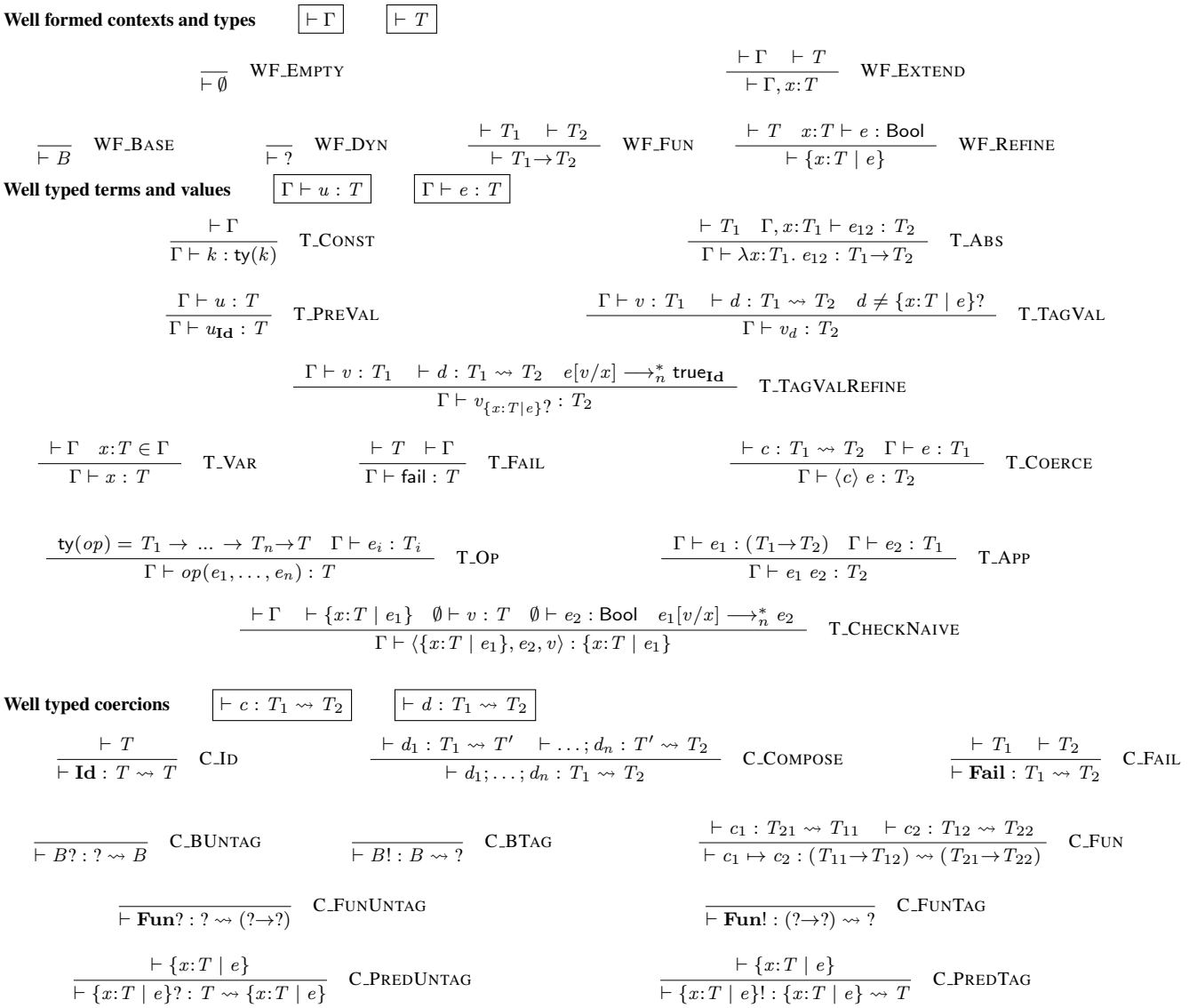

Figure 3. Typing

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 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 proofs.

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 2 splits coercions into three parts: composite coercions c, primitive coercions d, and tags D. The typing rules for coercions are written in Figure 3. 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 reassociation of coercion composition. We compare our coercions to other formulations in related work (Section 6). 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 dynamically typed languages.

We also have refinement tags for both types of refinement, which we write the same as the corresponding types.

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 2.2.) It is worthwhile to contrast our treatment of failure with that of Herman et al. [20]. Whereas Henglein treats mismatched tag/untag operations, such as $B!;\mathbf{Fun}?$, as stuck, we follow Herman et al. [20] 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 prevalues 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 naïve evaluation rule generates **Fail**. Instead, failures arise in the naïve calculus when the other rules with FAIL in their name fire. We include F_FAIL as a technicality for the soundness proof (Theorem 4.6) in Section 4. 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 [23], 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 [2, 3, 6, 9, 11, 13, 17, 18, 22, 33–35, 38, 39].

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 $\mathbf{Fun}$ are used to move values to and from the dynamic type ?. The tagging coercions $B!$ and $\mathbf{Fun}!$ mark a base value (typed B) or functional value (typed ?→?) as having the dynamic type ?. The checking coercions $B?$ and $\mathbf{Fun}?$ 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 $\mathbf{Fun}!$ and $\mathbf{Fun}?$ 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.

2.2 Operational semantics

Our rules are adapted from the evaluation contexts used in Herman et al. [20]. 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 4. 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 F_MERGE helps maintain determinism—if we didn't force F_MERGE to apply first, the number of coercions might grow out of control. 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 ?→?) into type dynamic, using the tagging coercions $B!$ and $\mathbf{Fun}!$, respectively. F_TAGBB and F_TAGFUNFUN apply $B?$ and $\mathbf{Fun}?$ to values that have matching $B!$ and $\mathbf{Fun}!$ 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 naïve calculus, 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 3, 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.

In Figure 5, we translate the cast example from the introduction (Figure 1). 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 $\langle\mathsf{Bool}?\rangle\ e$.

Finally, the naïve calculus is type sound.

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

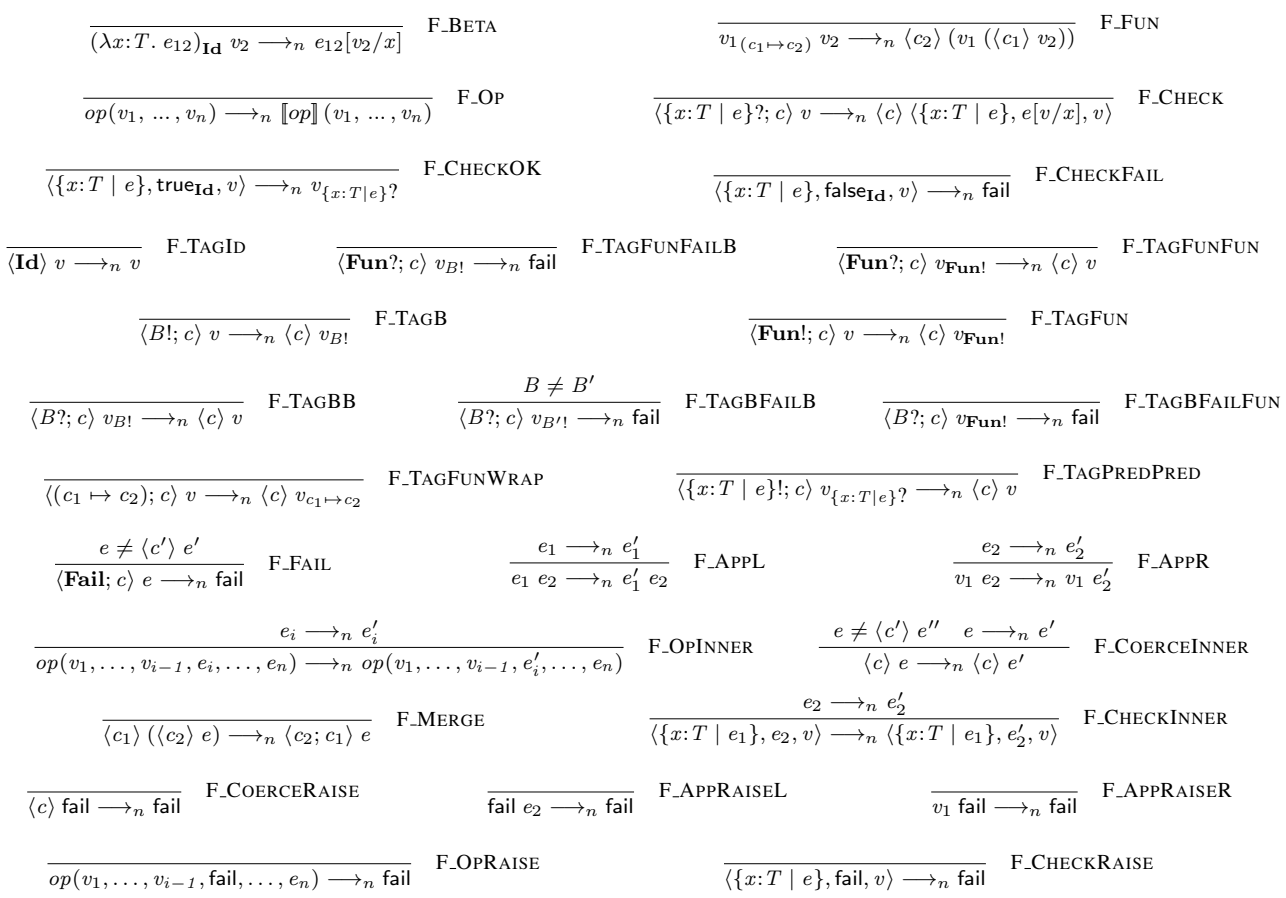

Figure 4. Naïve operational semantics

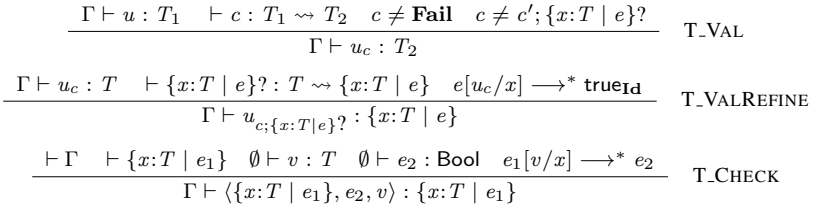

Figure 7. Updated typing rules for the space-efficient calculus

3. A space-efficient coercion calculus

Having developed the naïve semantics for our language, we now turn to space efficiency. There are two loci of inefficiency: coercion merges and function proxies (functional coercions). When F_MERGE applies, it merely concatenates two coercions: $\langle c_1 \rangle\ (\langle c_2 \rangle\ e) \longrightarrow_n \langle c_2; c_1 \rangle\ e$. Our space-efficient 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 the naïve semantics allows an arbitrary stack of tags values, the space-efficient semantics 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 3.1, we discuss changes to the syntax of values and to the typing rules.

We make changes to both the syntax (Figure 6) and typing rules (Figure 7) of the naïve calculus of Section 2; we define an entirely new operational semantics (Figure 10). Throughout the new typing rules, we assume that coercions are canonical (see below).

In the naïve calculus, tagging is stacked: a value is either a pre-value tagged with **Id** or a value tagged with a single primitive coercion. In the space-efficient calculus, 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.

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.

$$
\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 5. Naïve reduction

$$
\begin{array}{lcl}
r & ::= & v \mid \mathsf{fail} \\
v & ::= & u_c \\
u & ::= & k \mid \lambda x{:}T.\ e
\end{array}
$$

Figure 6. Updated syntax for the space-efficient calculus

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.

3.1 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 8. 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, our calculus will keep its coercions in canonical form for which we have a bounded size.

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:

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

We, however, directly define a rewrite system in Figure 8, where we lift $c \longrightarrow c'$ over all possible redexes in a composite coercion $d_1; \ldots; d_n$. Note that many of these rules correspond to reduction rules in the naïve 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 $\{x{:}B \mid e\}?$ that 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 4): we must have a ψ rule if $\{x{:}T \mid e\}!$ is going to untag $v_{\{x:T\mid e\}?}$. But if $\{x{:}T \mid e\}?; \{x{:}T \mid e\}!$ on tags, it must also hold for coercions on the stack if we want space efficiency. 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.

$$
\begin{array}{rcll}
\mathbf{Fail}; c & \longrightarrow & \mathbf{Fail} & (\text{F\_FAIL}) \\
c; \mathbf{Fail} & \longrightarrow & \mathbf{Fail} & \\
(c_{11} \mapsto c_{12}); (c_{21} \mapsto c_{22}) & \longrightarrow & (c_{21}; c_{11}) \mapsto (c_{12}; c_{22}) & \\
B!; B? & \longrightarrow & \mathbf{Id} & (\text{F\_TAGBB}) \\
B!; B'? & \longrightarrow & \mathbf{Fail} \text{ when } B \neq B' & \\
 & & & (\text{F\_TAGBFAILB}) \\
\mathbf{Fun}!; \mathbf{Fun}? & \longrightarrow & \mathbf{Id} & (\text{F\_TAGFUNFUN}) \\
B!; \mathbf{Fun}? & \longrightarrow & \mathbf{Id} & (\text{F\_TAGBFAILFUN}) \\
\mathbf{Fun}!; B? & \longrightarrow & \mathbf{Id} & (\text{F\_TAGFUNFAILB}) \\
\{x{:}T \mid e\}?; \{x{:}T \mid e'\}! & \longrightarrow & \mathbf{Id} & (\text{F\_TAGPREDPRED}) \\
\{x{:}T \mid e\}!; \{x{:}T \mid e\}? & \longrightarrow & \mathbf{Id} &
\end{array}
$$

Figure 8. Rewriting rules

3.1 Lemma: The rewrite system is strongly normalizing on well typed terms.

3.2 Lemma: The canonical coercions are normal.

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

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

Having developed the rewrite rules for our (somewhat relaxed) coercions, we define a merge algorithm in Figure 9 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 6.

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 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 3.6, which shows that merging is an operator on canonical coercions.

3.5 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 $\vdash c_3 : T_1 \rightsquigarrow T_3$.

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

$$\frac{}{\mathbf{Id} * c \Rightarrow c}\ \text{N\_IDL} \qquad \frac{c \neq \mathbf{Id}}{c * \mathbf{Id} \Rightarrow c}\ \text{N\_IDR} \qquad \frac{}{\mathbf{Fail} * c \Rightarrow \mathbf{Fail}}\ \text{N\_FAILL} \qquad \frac{c \neq \mathbf{Fail}}{c * \mathbf{Fail} \Rightarrow \mathbf{Fail}}\ \text{N\_FAILR}$$

$$\frac{c_1 * c_2 \Rightarrow c}{c_1; B! * B?; c_2 \Rightarrow c}\ \text{N\_BB} \qquad \frac{c_1 * c_2 \Rightarrow c}{c_1; \mathbf{Fun}! * \mathbf{Fun}?; c_2 \Rightarrow c}\ \text{N\_FUNFUN}$$

$$\frac{B \neq B'}{c_1; B! * B'?; c_2 \Rightarrow \mathbf{Fail}}\ \text{N\_BFAILB} \qquad \frac{}{c_1; B! * \mathbf{Fun}?; c_2 \Rightarrow c}\ \text{N\_BFAILFUN} \qquad \frac{}{c_1; \mathbf{Fun}! * B?; c_2 \Rightarrow \mathbf{Fail}}\ \text{N\_FUNFAILB}$$

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

Figure 9. Merging coercions

Coercion		Type		
$\mathbf{Id}$	:	T	$\rightsquigarrow$	T
$\mathbf{Fail}$	:	T	$\rightsquigarrow$	T'
$\{x{:}? \mid e\}?$	:	$?$	$\rightsquigarrow$	$\{x{:}? \mid e\}$
$B?; c$	:	$?$	$\rightsquigarrow$	T
		$\vdash c : B \rightsquigarrow T$ is a value coercion		
$\mathbf{Fun}?; c$	:	$?$	$\rightsquigarrow$	$?{\to}?$
		$\vdash c : (?{\to}?) \rightsquigarrow T$ is a value coercion		
$B!$	:	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 $e \neq e'$
$\{x{:}? \mid e\}!; B?; c$	:	$\{x{:}? \mid e\}$	$\rightsquigarrow$	T
		$\vdash c : B \rightsquigarrow T$ is a value coercion		
$\{x{:}? \mid e\}!; \mathbf{Fun}?; c$	:	$\{x{:}? \mid e\}$	$\rightsquigarrow$	$?{\to}?$
		$\vdash c : (?{\to}?) \rightsquigarrow T$ is a value coercion		
$\{x{:}B \mid e\}!; c$	:	$\{x{:}B \mid e\}$	$\rightsquigarrow$	T
$\vdash c : B \rightsquigarrow T$ is a value coercion and $c \neq \{x{:}B \mid e\}?$				

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 source type.

Table 1. Canonical coercions

3.6 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 $c_1 * c_2 \Rightarrow c$.

3.2 Operational semantics

We give the relevant rules of the changed operational semantics in Figure 10. 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 simply 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: 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 11 don't grow with the size of the input like the coercions in Figure 5 or the casts in Figure 1. We discuss this claim in more detail in Section 5.

$$\begin{aligned}
& \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{aligned}$$

Figure 11. Space-efficient reduction

3.3 Proofs

Our space-efficient calculus enjoys type soundness; we show as much using standard syntactic methods.

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

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

4. Soundness of the space-efficient calculus

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* the naïve semantics produces a value, the space-efficient calculus will produce a similar one.

$$\frac{}{u_{1\,(c_1 \mapsto c_2)}\ v_2 \longrightarrow \langle c_2 \rangle\ (u_{1\,\mathbf{Id}}\ (\langle c_1 \rangle\ v_2))}\ \text{E\_Fun}$$

$$\frac{}{\langle c_1 \rangle\ (\langle c_2 \rangle\ e) \longrightarrow \langle c_2 \Downarrow c_1 \rangle\ e}\ \text{E\_Merge}$$

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

$$\frac{d_1 \neq \{x{:}T \mid e\}? \quad c * d_1 \Rightarrow \mathbf{Fail}}{\langle d_1; c_2 \rangle\ u_c \longrightarrow \mathsf{fail}}\ \text{E\_TagFail}$$

$$\frac{}{\langle \mathbf{Id} \rangle\ v \longrightarrow v}\ \text{E\_TagId}$$

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

Figure 10. Operational semantics

Pre-value and value rules

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

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

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

Term rules

$$\begin{array}{c} e_1 \simeq e_2 : T \\ \iff \\ e_1 \text{ diverges} \vee e_1 \longrightarrow_n^* \mathsf{fail} \vee \\ (e_1 \longrightarrow_n^* v_1 \wedge e_2 \longrightarrow^* v_2 \wedge v_1 \sim v_2 : T) \end{array}$$

Type rules

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

Closing substitutions

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

Figure 12. Relating the naïve and space-efficient semantics

We adapt the asymmetric logical relations from Greenberg et al. [17] to show that the two calculi behave mostly the same, with the naïve calculus diverging and failing more often. We define the logical relation in Figure 12. The definitions begin by defining a relation $v_1 \sim v_2 : T$ for closed values and a relation $e_1 \simeq e_2 : T$ for closed terms as a fixpoint on types (it is a known bug that this isn't quite right—rather, we ought to use step-indexed logical relations). We lift the definitions to open terms by defining *dual closing value substitutions* δ; if $\Gamma \models \delta$ and $x{:}T \in \Gamma$, then $\delta_1(x) \sim \delta_2(x) : T$.

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 the space-efficient calculus 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.

Calculating canonical coercions

$$\begin{array}{rcl} \mathsf{canonical}(\mathbf{Id}) & = & \mathbf{Id} \\ \mathsf{canonical}(\mathbf{Fail}) & = & \mathbf{Fail} \\ \mathsf{canonical}(d_1; \ldots; d_n) & = & \\ & & \mathsf{canonical}(d_1) \Downarrow \mathsf{canonical}(d_2; \ldots; d_n) \\ & & \\ \mathsf{canonical}(D!) & = & \mathsf{canonical}(D)! \\ \mathsf{canonical}(D?) & = & \mathsf{canonical}(D)? \\ \mathsf{canonical}(c_1 \mapsto c_2) & = & \mathsf{canonical}(c_1) \mapsto \mathsf{canonical}(c_2) \\ & & \\ \mathsf{canonical}(B) & = & B \\ \mathsf{canonical}(\mathbf{Fun}) & = & \mathbf{Fun} \\ \mathsf{canonical}(\{x{:}T \mid e\}) & = & \{x{:}T \mid \mathsf{canonical}(e)\} \\ & & \\ \mathsf{canonical}(u_{\mathbf{Id}}) & = & \mathsf{canonical}(u)_{\mathbf{Id}} \\ \mathsf{canonical}(v_d) & = & u_{c \Downarrow d} \\ & & \text{where } \mathsf{canonical}(v) = u_c \end{array}$$

Figure 13. Canonicalizing naïve terms

The value relation $v_1 \sim v_2 : T$ 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 the case of the space-efficient calculus).

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 canonical in Figure 13, 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 naïve-calculus value and merge them into a single coercion.

Our ultimate goal is soundness: if $\Gamma \vdash e : T$ then $\Gamma \vdash e \simeq \mathsf{canonical}(e) : T$. 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 omit the details of these relations for space, though they are available in the supplementary materials. We then prove lemmas that allow us to easily work with ignorable and failable coercions (Lemma 4.1 and Lemma 4.2, respectively). Then we relate non-canonical coercions to canonical ones (using a separate inductive relation $\vdash c_1 \sim c_2$, defined in Figure 14). 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.

Relating coercions

$$\frac{\vdash c_i' \textbf{ ignorable} \quad \vdash c_i \sim d_i}{\vdash c_0'; c_1; c_1'; c_2; \ldots; c_{n-1}'; c_n; c_n' \sim d_1; \ldots; d_n} \text{ R\_COMPOSITE} \qquad \frac{\vdash c \textbf{ ignorable}}{\vdash c \sim \textbf{Id}} \text{ R\_ID}$$

$$\frac{\begin{array}{l}\vdash c_i' \textbf{ ignorable} \\ \vdash (c_{1\,n\,1}; \ldots; c_{111}) \mapsto (c_{112}; \ldots; c_{1\,n\,2}) \sim c_{21} \mapsto c_{22}\end{array}}{\vdash (c_{111} \mapsto c_{112}); c_1'; \ldots; c_n'; (c_{1\,n\,1} \mapsto c_{1\,n\,2}) \sim c_{21} \mapsto c_{22}} \text{ R\_FUN} \qquad \frac{\vdash c_1 \textbf{ failable}}{\vdash c_1'; c_1; c_2' \sim c_2} \text{ R\_FAIL}$$

$$\frac{\vdash D \sim D}{\vdash D! \sim D!} \text{ R\_TAG} \qquad \frac{\vdash D \sim D}{\vdash D? \sim D?} \text{ R\_CHECK} \qquad \frac{}{\vdash B \sim B} \text{ R\_DB} \qquad \frac{}{\vdash \textbf{Fun} \sim \textbf{Fun}} \text{ R\_DFUN}$$

$$\frac{}{\vdash \{x{:}T \mid e_1\} \sim \{x{:}T \mid \mathsf{canonical}(e_1)\}} \text{ R\_DPREDCANONICAL} \qquad \frac{\forall v_1 \sim v_2 : T.\ e_1[v_1/x] \simeq e_2[v_2/x] : \mathsf{Bool}}{\vdash \{x{:}T \mid e_1\} \sim \{x{:}T \mid e_2\}} \text{ R\_DPREDLR}$$

Figure 14. Relating coercions

Ignorable coercions can be freely added or removed to naïve terms while preserving logical relation to space-efficient terms.

4.1 Lemma: If $\langle c_1; c_2\rangle\ v_1 \simeq e_2 : T$ and $\vdash c_1$ **ignorable** then $\langle c_2\rangle\ v_1 \simeq e_2 : T$.

We prove a similar lemma that *failable* coercions always fail.

4.2 Lemma: If $\vdash c_1$ **failable**, then $\langle c_1'; c_1; c_2'\rangle\ v_1 \simeq e_2 : T$.

With ignorable and failable coercions, we can characterize *all* non-canonical coercions, relating them to canonical coercions. The relation $\vdash c_1 \sim c_2$ 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 the naïve calculus are related to $\mathsf{canonical}(c)$. 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 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.

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

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

4.4 Lemma [Relating canonical coercions]:
If $v_1 \sim v_2 : T_1$ and $\vdash c_1 : T_1 \rightsquigarrow T_2$ and $\vdash c_1 \sim c_2$ (using R_DPREDLR), then $\langle c_1\rangle\ v_1 \simeq \langle c_2\rangle\ v_2 : T_2$.

In Greenberg et al. [17], 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 4.4. 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 $\vdash c_1 \sim c_2$.

4.5 Lemma [Relating coercions with merges]: If $e_1 \simeq e_2 : T_1$ and $\vdash c_1 : T_1 \rightsquigarrow T_2$ and $\vdash c_1 \sim c_2$ (using R_DPREDLR), then $\langle c_1\rangle\ e_1 \simeq \langle c_2\rangle\ e_2 : T_2$.

Proof: First, we can ignore the cases where $e_1 \longrightarrow_n^* \mathsf{fail}$ or e_1 diverges—those are immediately related. So $e_1 \longrightarrow_n^* v_1$ and $e_2 \longrightarrow^* v_2$. By induction on the length of the evaluation derivations, with a long case analysis on the space-efficient side. □

4.6 Theorem [Soundness]: If $\Gamma \vdash e : T$ then $\Gamma \vdash e \simeq \mathsf{canonical}(e) : T$.

Proof: We generalize the proof to work also on values and terms. By lexicographic induction on the typing derivation and the size of the term (v or e, respectively), using Lemma 4.5 in the T_COERCE case. We use Lemma 4.3 and Lemma 4.4 in the T_TAGVAL and T_TAGVALREFINE cases. □

The definition of $\emptyset \vdash e_1 \simeq e_2 : T$ gives us our approximate observational equivalence: either $e \longrightarrow_n^* \mathsf{fail}$, e diverges, or $e \longrightarrow_n^* v_1$ and $\mathsf{canonical}(e) \longrightarrow^* v_2$ such that $v_1 \sim v_2 : T$. Note that for base values, we have exactly the same result on both sides.

5. 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 $\{x{:}? \mid e\}!; \textbf{Fun}?; c_1 \mapsto c_2; \textbf{Fun}!; \{x{:}? \mid e'\}?$, 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.

5.1 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 $c_1 * c_2 \Rightarrow c_3$. □

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.

6. 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 [34], 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 [20] compiled a language like Siek and Taha's into a calculus with Henglein's coercions [19], 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 [33] 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 [35] study an alternative, cast-based formulation of space efficiency, proving tighter bounds than Herman et al. [20] 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 [21] 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 7).

Before considering other full-spectrum languages, we compare our work to the most closely related work: Herman, Tomb, Flanagan [20, 21] and Henglein [19]. 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 [21], 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.

The work discussed so far consisted of calculi devised expressly for space-efficient gradual typing. Findler et al. [12] 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.

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. [30] cover the spectrum and include dependent types, but allow only a constrained set of refinement predicates; Sage [25] covers the entire spectrum and also includes dependent types, but lacks a soundness proof; Wadler and Findler's [39] 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 [19]. One of the classic uses of coercions is subtyping [5, 28]; more recent work relates subtyping and polymorphism [8]. Work on unboxing [29, 32] confronts similar issues of space efficiency. Many of these examples carefully ensure that coercions are erasable, while our coercions are definitely not.

Swamy, Hicks, and Bierman [36] 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.

7. Future work

The obvious next step is adding blame [11]. Siek and Wadler [35] 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 believe that our inside-out coercion merge algorithm offers a straightforward way to compute blame: we conjecture that blame comes from left to right.

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 that doesn't hold for a polymorphic calculus. Prior work relating dynamic types and polymorphism will apply here, as well [2, 26].

Adding dependent functions to the coercion calculus above would complicate matters significantly, but would also add a great deal of expressiveness. Programs with dependent types have a potentially infinite set of types (and so coercions) as the program evaluates, but the *shape* of canonical coercions remain unchanged.

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.

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 and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*, 2002.

[12] 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.

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

[14] 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.

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

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

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

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

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

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

[21] 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.

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

[23] 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.

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

[25] 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.

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

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

[28] 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.

[29] 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.

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

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

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

[33] 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.

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

[35] 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.

[36] 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.

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

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

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